Institut für Informatik
der Technischen Universität München

# Integriertes Management erweiterbarer verteilter Systeme

*Markus Pizka*

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. U. Baumgarten |
| Prüfer der Dissertation: | |
| 1. | Univ.-Prof. Dr. P. P. Spies |
| 2. | Univ.-Prof. Dr. A. Bode |

Die Dissertation wurde am 24.6.1999 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 29.9.1999 angenommen.

# Kurzfassung

Mit vernetzten Arbeitsplatzrechnern stehen enorme Rechen- und Speicherkapazitäten zur Verfügung, mit deren Hilfe Aufgabenstellungen schneller und mit höherer Qualität als bisher bewältigt werden könnten. Die konzertierte Nutzung physisch verteilter Ressourcen scheitert jedoch bislang an einer geeigneten Infrastruktur für die Programmierung und den Betrieb dieser Anlagen. Zur Verbesserung dieser Situation wurden bereits zahlreiche Lösungsansätze erarbeitet, die sich meist aus Gründen des Aufwands auf spezielle Probleme konzentrieren und verteiltes Rechnen für Sonderfälle ermöglichen. Charakteristisch für diese Speziallösungen ist, daß sie durch ihre mangelnde Integration in eine Gesamtarchitektur entweder unter unbefriedigender Effizienz leiden oder ihre Programmierung nicht praktikabel ist.

Die Alternative dazu ist die Entwicklung neuer Methoden, die es ermöglichen, die ungleich komplexere Aufgabe der gesamtheitlichen Konstruktion verteilter Systeme zu bewältigen. Im Projekt MoDiS wird dieser Ansatz mit dem Ziel verfolgt, verteilte Konfigurationen als qualitativ und quantitativ leistungsfähige verteilte Systeme für parallele und kooperative Problemlösungen einfach und universell nutzen zu können.

In diese Zielsetzung ordnet sich die vorliegende Arbeit ein, in der eine flexible und integrierte verteilte Ressourcenverwaltung für die automatisierte Realisierung paralleler und kooperativer Problemlösungen entworfen wird. Im Sinne der gesamtheitlichen Vorgehensweise steht die Integration der Verwaltungsmaßnahmen zur Vermeidung von Reibungsverlusten im Vordergrund. Da starre Verwaltungsverfahren wegen der großen Leistungsdifferenz zwischen lokalen und entfernten Zugriffen in verteilten Umgebungen ungeeignet sind, wird das Management zusätzlich systematisch flexibilisiert. Um diese Ziele erreichen zu können muß das Management kreativ Klassen von Ressourcen entwerfen und ihren Einsatz neu planen.

Den Ausgangspunkt für die Gewinnung der Information, die zur Steuerung der flexibilisierten Verfahren benötigt wird, liefert der Ein-Programm-Ansatz von MoDiS, der sämtliche Komponenten einer verteilten Berechnung in ein globalstrukturiertes Gesamtsystem integriert, das durch genau ein Programm beschrieben ist. Für die inkrementelle Weiterentwicklung des langlebigen Systems werden Verfahren erarbeitet, die es erlauben, das in Ausführung befindliche Programm dynamisch zu modifizieren und dabei insbesondere auch zu erweitern.

Sowohl für das Management als auch die Erweiterbarkeit werden im ersten Schritt methodische Grundlagen erarbeitet. Die sich anschließende Realisierung des Managements konzentriert sich auf die Beiträge transformatorischer Maßnahmen des Übersetzers und des Binders, an deren Beispiel die flexible Produktion und Bindung von Ressourcen demonstriert wird. Um diese Aufgabe bewältigen zu können, wird existierende Software an die veränderten Anforderungen angepaßt. Die Ergebnisse dieser Vorgehensweise werden abschließend anhand ausgewählter Fallstudien, unter anderem der Verwaltung multipler Keller in dem verteilten Ein-Adreßraum und der Parameterübergabe evaluiert.

# Danksagung

Zu herzlichstem Dank bin ich Herrn Prof. Dr. Peter Paul Spies für die langjährige Betreuung dieser Arbeit und die zahlreichen Denkanstöße in den vergangenen Jahren meiner Ausbildung verpflichtet. Mein Dank gilt des weiteren Herrn Prof. Dr. A. Bode für die Begutachtung dieser Arbeit und Herrn Prof. Dr. U. Baumgarten, für die Bereitschaft, den Vorsitz meiner Prüfungskomission zu übernehmen.

Bedanken möchte ich mich auch bei den Studenten und Kollegen, die im Laufe der Jahre zum Gelingen dieser Arbeit beigetragen haben. Hervorheben möchte ich hierbei meine Kollegin Frau Dr. Claudia Eckert für zahllose Diskussionen und ihre langjährige Hilfestellung, Herrn Dipl.-Inform. Christian Rehn für sein ausdauerndes Interesse an dieser Arbeit als studentische Hilfskraft, Diplomand und Kollege sowie Herrn Dipl.-Inform. Jürgen Rudolph für seine freundschaftliche Unterstützung in vielen Situationen.

Herzlich Danken möchte ich meinen Freunden, die mir in schwierigen Phasen dieser Arbeit mit Rat und Tat beistanden. Dies gilt in ganz besonders hohem Maße für Herrn Dipl-Ing., Dipl-Wirtschafts-Ing. Harald Blusch für seine wiederholte, freundschaftliche Animation zu weiterführenden Arbeiten sowie Frau Katharina Blusch-Reisenbüchler, die in den vergangenen Tagen diese umfangreiche Arbeit unermüdlich gelesen und einen wertvollen Beitrag zur Minimierung der enthaltenen Fehler geleistet hat.

Besonderer Dank gilt meinen Eltern, die mir das Studium ermöglichten und mich auch in der anstrengenden Endphase der vorliegenden Arbeit unterstützt haben.

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

Das große Reservoir an preiswerter Rechenleistung und Speicherfähigkeit moderner vernetzter Rechner bietet die Möglichkeit, ein großes Spektrum an Aufgaben quantitativ und qualitativ besser als mit den bislang dominierenden isolierten Arbeitsumgebungen bewältigen zu können. Fortschritte im Bereich der Hardware in Form neuer Fertigungstechniken und Rechnerarchitekturen [Hwa93] führten in den vergangenen Jahren dazu, daß sich die Rechenleistung moderner Prozessoren alle ein bis zwei Jahre bei gleichzeitiger Reduktion der Kosten verdoppelt hat. Besonders deutlich ist diese Entwicklung am Beispiel der „Personal Computer“ auf Basis der Intel x86 Architektur zu sehen. Wurde die Leistungsfähigkeit des 80386 Prozessors mit 33 MHz Taktfrequenz bei seiner Einführung 1989 noch mit 6.4 Cint92/SPEC [Shn96] angegeben, so erreichte der Prozessor 80486/66 drei Jahre später bereits 39.6 Cint92. Der Ende 1998, zum Zeitpunkt der Niederschrift, aktuelle Vertreter dieser Familie „Pentium II“ mit 400 MHz Taktung erzielt mit 15.8 Cint95 [SPE97] eine Leistung, die in etwa der Rechenleistung eines SUN RISC Ultra II/360 Arbeitsplatzrechners entspricht [Bur98]. Im Bereich der Speichermedien verläuft die Entwicklung ähnlich. Zusätzlich schreitet die Koppelung von Rechnern mittels schneller Netze, wie z. B. 100MBit/s FastEthernet, im hohen Tempo fort. Resultat dieser Entwicklung ist, daß bereits heute in Firmen, Instituten und global durch das Internet physisch verteilte Rechenanlagen mit enormer Leistungsfähigkeit zur Verfügung stehen. Diese technischen Fortschritte motivieren die Nutzung dieses großen Potentials als kosteneffektivere Alternative zu Hochleistungsrechnern und darüber hinaus für die Verwirklichung grundlegend neuer Ziele, die erst durch die wesentlich höhere Flexibilität dieser Konfigurationen möglich werden. So kann die Integration physisch verteilter Prozessoren und Speicher über ein Nachrichtentransportsystem von der Durchführung rechen- und speicherintensiver Berechnungen im technisch-wissenschaftlichen Bereich über die Erhöhung der Zuverlässigkeit bis hin zur Unterstützung neuer Organisationsformen, wie etwa Telearbeit oder rechnergestützte Gruppenarbeit [EGR91] – CSCW –, eingesetzt werden.

Diese vielfältigen intuitiven Nutzungsmöglichkeiten werden nach wie vor durch den Mangel an einer geeigneten Infrastruktur für die Programmierung und den Betrieb verteilter Systeme nahezu egalisiert. Zahlreiche Forschungsarbeiten auf diesem Gebiet lassen vermuten, daß der skizzierte Wandel der Nutzung vernetzter Rechner, den Wechsel zahlreicher Paradigmen voraussetzt. Grobgranulare, statische, isolierte und sequentielle Abläufe sind durch feingranulare, hoch dynamische, kooperative und parallele Verfahren abzulösen. Dies bedeutet, daß neue Algorithmen, Sprachen, Systemarchitekturen, Werkzeuge und Betriebssysteme mit inhärenter Unterstützung für paralleles und verteiltes Rechnen benötigt werden. Um verteiltes Rechnen

auf das Niveau eines Standardangebots moderner Rechensysteme anheben zu können, wird es von zentraler Bedeutung sein, diese Übergänge auf allen Abstraktionsebenen konsequent zu vollziehen.

In den jeweiligen Teilgebieten existieren bereits zahlreiche Lösungen von Detailproblemen. Die Voraussetzungen und Zielsetzungen der einzelnen Arbeiten variieren in der Regel jedoch so stark, daß die Vorstellung, die Fähigkeiten der verschiedenen Lösungen nachträglich kombinieren zu können, nicht tragfähig ist. In der Realität existiert eine Fülle heterogener Sprachen, Bibliotheken und sonstige Erweiterungen für verteiltes Rechnen, die meist nur für spezifische Zwecke geeignet sind und nicht kombinierbar sind. Symptomatisch für viele der separaten Detaillösungen ist, daß ihre Nutzung entweder kompliziert und damit für den allgemeinen Einsatz nicht praktikabel ist oder nur enttäuschende Leistungswerte erzielt werden. Der Bereich „Distributed Shared Memory“ (DSM) ist ein Beispiel für dieses Dilemma [Car98]. Nicht praktikable Verfahren entstehen durch die partielle Betrachtung realisierungsnaher Abstraktionsebenen. Fähigkeiten, die auf diese Weise entwickelt werden, lassen sich später oft nur ungenügend in die Konstruktionskonzepte höherer Abstraktionsebenen inklusive der Programmiersprachen integrieren. Andererseits entsteht unbefriedigende Leistung durch die isolierte Betrachtung von Konzepten auf hohen Abstraktionsebenen. Werden für die Realisierung existierende Mechanismen unreflektiert übernommen, so entstehen Inkompatibilitäten und damit verbunden Reibungs- bzw. Effizienzverluste.

Eine Verbesserung dieser Situation kann auf zwei Wegen erfolgen. Eine Möglichkeit ist die gewohnte Weiterentwicklung von Systemen mit der wie bisher üblichen Integration sukzessiver Detailverbesserungen. Aufgrund der enormen Vielfalt der Detailprobleme und dem Aufwand für Neuentwicklungen sind auf diese Weise auch langfristig nur geringfügige Fortschritte zu erwarten. 20 Jahre nach der Charakterisierung „Verteilter Rechensysteme“ [Ens78] gegen Ende der 70er Jahre sind die auf diese Weise erzielten Fortschritte nach wie vor gering. Die Alternative dazu ist die Entwicklung neuer Methoden, die es ermöglichen, komplexe verteilte Systeme gesamtheitlich zu konstruieren. Dieser Ansatz wird in dem Projekt MoDiS[1] [EW95b, EW95a] verfolgt. Das Ziel ist die Entwicklung von Konzepten und Verfahren für die Spezifikation und automatisierte Realisierung verteilter Anwendungssysteme auf Basis vernetzter Hardwarekonfigurationen. Geeignete Sprachkonzepte für die Konstruktion qualitativ hochwertiger Systeme werden losgelöst von Realisierungsdetails entwickelt. Das Problem mangelnder Praktikabilität entsteht somit nicht. Um gleichzeitig Leistungsfähigkeit zu erzielen, wird das Ressourcenmanagement, das für die verteilte Ausführung von Programmen auf diesem hohen Abstraktionsniveau benötigt wird, gesamtheitlich und angepaßt konstruiert.

In diese Zielsetzung ordnet sich die vorliegende Arbeit ein. Im Kontext von MoDiS wird eine angepaßte und flexible verteilte Ressourcenverwaltung für die automatisierte und effiziente Realisierung paralleler und kooperativer Anwendungssysteme entworfen und realisiert. Im Sinne der gesamtheitlichen Vorgehensweise steht die Integration und Homogenität von Verwaltungskonzepten und ihrer Realisierungen im Zentrum der Überlegungen.

Die ungleich höhere Komplexität dieser gesamtheitlichen Vision gegenüber der Betrachtung von Teilaspekten erfordert eine deutliche Verbesserung der Methodik, mit der Managementsysteme konstruiert werden. Das bisher überwiegend empirisch geprägte Vorgehen birgt die Gefahr, die Tragweite von Entscheidungen in veränderten Umgebungen zu unterschätzen. Bei dem Übergang von zentralen zu verteilten Systemen gibt es zahlreiche Situationen, die sich oberflächlich betrachtet nur geringfügig unterscheiden, durch die Diskrepanz zwischen

[1] Model oriented Distributed Systems

lokalen und verteilten Zugriffen jedoch ein drastisch unterschiedliches Verhalten des Systems bewirken. Bislang existiert keine Systematik, mit der die damit einhergehenden Gefahren erkannt und schwerwiegende Fehler vermieden werden können. Bereits die Kenntnis der zahlreichen Lösungsmöglichkeiten für Verwaltungsaufgaben sowie die Auswahl einer Alternative unterliegt nahezu ausschließlich der individuellen Kreativität des Systementwicklers. D. Knuth charakterisiert dieses Problem folgendermaßen [SL95]:

„Computer programming is an art form, like the creation of poetry or music."

Die Diskrepanz zwischen den hohen qualitativen und quantitativen Anforderungen an verteilte Betriebssysteme und der ungenügenden Methodik ist ein wesentlicher Grund für die nach wie vor unbefriedigende infrastrukturelle Unterstützung für verteiltes Rechnen. Im Zuge der Arbeiten an dem integrierten Ressourcenmanagement im Kontext von MoDiS wird diesem Defizit durch die Erarbeitung einer grundlegend neuen Betrachtung verteilter Managementsysteme und einem daraus resultierendem Ansatz zur Systematisierung der Konstruktionsaufgabe begegnet.

Die Ergebnisse, die auf diese Weise erzielt werden, sind mitnichten auf den Kontext von MoDiS beschränkt. Wenngleich MoDiS den Rahmen der Untersuchungen vorgibt, sind besonders die erarbeiteten methodischen Grundlagen auf verwandte Arbeiten übertragbar. Bei den konkreten Implementierungsarbeiten werden existierende Systemprogramme und Werkzeuge für den Einsatz in verteilten Umgebungen modifiziert. Die so entstehenden Produkte können und werden in gängigen, arbeitsfähigen Systemen eingesetzt.

## 1.1 Problemstellung

Das gesamtheitliche Denken wird in MoDiS mit einer sprachbasierten und top-down orientierten Vorgehensweise kombiniert, um qualitativ und quantitativ hochwertige verteilte Systeme zu konstruieren. Den Ausgangspunkt bildet ein Vorrat an homogenen Sprachkonzepten [RW96], die mit der Syntax der Programmiersprache INSEL[2][SEL+96] operationalisiert sind und es erlauben, parallele und kooperative Berechnungen auf hohem Abstraktionsniveau zu spezifizieren. Die physische Verteilung der Ausführungsumgebung ist auf dieser Ebene völlig opak. Die Transformation eines auf INSEL-Niveau formulierten Problemlösungsverfahrens in ein auf real verteilter Hardware effizient rechnendes System, ist Aufgabe des systemintegrierten Ressourcenmanagements, d. h. des Betriebssystems. Den Entwicklern und Nutzern verteilter Problemlösungen werden vernetzte Rechnerkonfigurationen als einfach nutzbare und leistungsfähige verteilte Systeme für parallele und kooperative Problemlösungen – **V-PK-Systeme** – zur Verfügung gestellt [ECPR97a, ECPR97b]. Die Nutzung verteilter Ressourcen soll nicht länger auf Ausnahmesituationen für besonders rechen- oder speicherintensive Berechnungen beschränkt bleiben, sondern allgemein als Standardfall betrieben werden. Diese *general purpose*[3] Ausrichtung ist ein grundlegender Unterschied von MoDiS und der V-PK-Idee zu anderen ebenso bedeutsamen Arbeiten, die sich der dedizierten Unterstützung spezifischer Domänen widmen.

Eingebettet in die V-PK-Vision, verteilte Rechensysteme universell nutzen zu können, sind in der vorliegenden Arbeit folgende zwei Aufgaben zu bearbeiten:

[2] Integration and Separation Supporting Language
[3] engl.: Allzweck

**A** *Integrierte und flexible Ressourcenverwaltung in V-PK-Systemen*

**B** *Inkrementelle Erweiterbarkeit von V-PK-Systemen*

Im Mittelpunkt des Interesses steht Aufgabe **A**, deren Lösung, wie später gezeigt wird, grundlegende Kenntnis über die Gesamtstruktur der verteilten Berechnungen voraussetzt. Um diese Voraussetzung konstruktiv zu schaffen, wird der unter Punkt **B** formulierten Fragestellung nachgegangen. Für beide Aspekte sind zunächst methodische Grundlagen und Lösungsansätze zu erarbeiten und anschließend Realisierungsarbeiten zur Evaluation und Demonstration der entworfenen Konzepte zu leisten. Eine wichtige Bedeutung kommt dabei der Frage zu, wie der enorme Aufwand für die Realisierung eines völlig neu entworfenen Managementsystems, von der Ebene der Programmiersprache INSEL bis hin zu der Hardware, reduziert werden kann. Der Grund dafür, daß in der Regel die Bearbeitung von Teilaspekten bevorzugt wird, ist selbstverständlich der Aufwand. In der vorliegenden Arbeit wird dennoch versucht, den Faktor Aufwand nicht als dominantes Entscheidungskriterium zu akzeptieren. Im Gegenzug ist zu untersuchen, wie arbeitsfähige Verfahren unter diesen Voraussetzungen realisiert werden können.

In den folgenden Abschnitten werden die unter **A** und **B** genannten Aufgabenstellungen präzisiert.

### 1.1.1 Inkrementelle Erweiterbarkeit von V-PK-Systemen

Die anvisierte Steigerung der Qualität und Beherrschbarkeit verteilter Systeme durch ein hohes Abstraktionsniveau und vollständige Transparenz der Verteilung auf der Ebene der Sprachkonzepte hat zur Folge, daß der Entwickler von der Aufgabe verteilten Managements entbunden wird, um sich auf die abstrakte Problemlösungsstruktur konzentrieren zu können. Das Betriebssystem muß im Gegensatz zu anderen Plattformen für verteiltes Rechnen, wie z. B. PVM [GBD+94] oder MPI [For94], Speicher-, Rechen- und Kommunikationsressourcen selbständig verteilt verwalten und damit eine ungleich schwierigere Aufgabe erfüllen. Die Qualität der Lösung dieser Aufgabe ist von der Existenz ausreichend präziser Information abhängig. Das Wissen über den Algorithmus und die Ausführungsumgebung, das bei anwendungsintegriertem Management vom Programmierer mit der Problemlösungsstruktur vermengt und unmittelbar zur Optimierung von Nachrichtentransporten, etc. genutzt wird, muß im Falle des systemintegrierten Managements durch eine Informationsgrundlage ersetzt werden, die es den Verfahren erlaubt, adäquate Realisierungsentscheidungen automatisiert zu fällen. Durch die enormen Leistungsunterschiede zwischen Entscheidungsalternativen, wie z. B. Faktor $\approx 3.4 * 10^4$ bei lokalem vs. entferntem Zugriff auf Datenobjekte [Win96b], ist Information über globale Abhängigkeiten in verteilten Umgebungen von herausragender Bedeutung.

Eine Grundlage für die Gewinnung dieser Information wird in MoDiS konstruktiv durch den gewählten Ein-Programm-Ansatz [BS90] geschaffen. Die gesamtheitliche Betrachtung von Systemen ist in MoDiS nicht auf das Management limitiert, sondern bezieht sich ebenso auf das zu verwaltende System selbst. Sämtliche auf der verteilten Konfiguration ablaufenden Berechnungen werden als Ausführung genau eines Programmes betrachtet. Das System entwickelt sich ausgehend von einer Hauptkomponente dynamisch durch Erzeugung und Auflösung abhängiger Komponenten. Zu jedem Zeitpunkt $t$ besteht das verteilte System aus einer Menge $K_t$ von partiell geordneten Komponenten. Die strukturellen Abhängigkeiten zwischen den $k \in K_t$ werden durch den Einsatz der INSEL Konzepte etabliert und können von

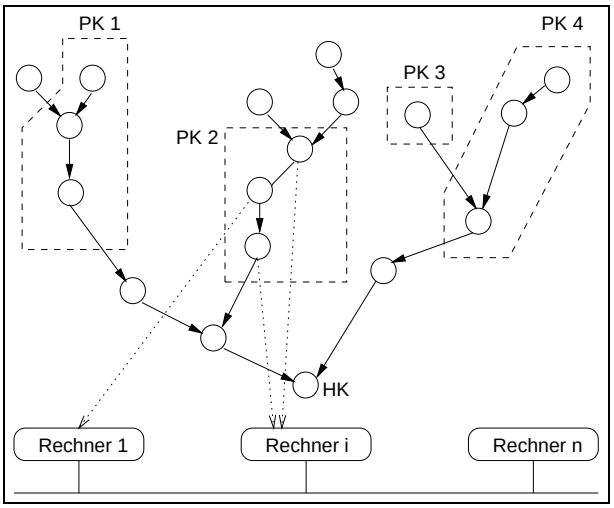


Abbildung 1.1: Gesamtsystemsicht basierend auf dem Ein-Programm-Ansatz

dem systemintegrierten Management weitestgehend automatisiert extrahiert werden. Mit den globalen, strukturellen Abhängigkeiten zwischen den Komponenten steht dem Management erste wichtige Information über lokale und nicht-lokale Zusammenhänge zur Verfügung.

Abbildung 1.1 skizziert das Modell eines verteilten und konzeptionell strukturierten Gesamtsystems. Ausgehend von der Hauptkomponente HK wurden Problemlösungskomplexe PK und andere im Beispiel nicht näher charakterisierte Komponenten erzeugt. Für jede Komponente $k \in K_t$ gilt, daß sie in Relation zu mindestens einer anderen Komponente $k' \in K_t$ steht. Für die Ausführung sind die Komponenten des Systems, wie an PK2 demonstriert, auf die vernetzten Stellenrechner abgebildet.

Der Ein-Programm-Ansatz integriert Problemlösungskomplexe und damit Anwendungen und Betriebssystemfunktionalität in ein Gesamtsystem, dessen Komponenten in einem engen Zusammenhang stehen. Diese System-Sicht stellt neue Anforderungen an die Systemkonstruktion, die im Gegensatz zu der bisher üblichen statischen und grobgranularen Entwicklung kurzlebiger Anwendungssysteme als ein fortschreitender feingranularer Evolutionsprozeß des gesamten Systems zu verstehen ist. Charakterisch für die Weiterentwicklung des Systems ist seine dynamische und inkrementelle *Erweiterung* und *Reduktion*.

**Definition 1.1 (P-Erweiterung)**
*Eine P-Erweiterung des System $S_t$ zum Zeitpunkt $t$ ist eine konsistente Ergänzung des $S_t$-beschreibenden Programmes $P_t$ um zusätzliche Komponenten.*

**Definition 1.2 (P-Reduktion)**
*Bei einer P-Reduktion des Systems $S_t$ zum Zeitpunkt $t$ werden Komponenten aus dem $S_t$-beschreibenden Programm $P_t$ entfernt.*

Neu in das System eingebrachte Beschreibungen von Komponenten erweitern das in Ausführung befindliche Programm. Ebenso können Komponenten dynamisch durch Reduktionsschritte wieder aus dem System entfernt werden. Durch aufeinanderfolgende Erweiterungen und Reduktionen und zusätzlichem Transfer von Zuständen lassen sich dann *Revisionen* realisieren, mit deren Hilfe ein mächtiges Instrumentarium für die flexible Konstruktion

anpassungsfähiger Systeme zur Verfügung stehen wird. Da für die Realisierung dieser Vorstellung bislang grundlegende Konzepte fehlen, werden in dieser Arbeit als erste Schritte auf dem Weg zu gesamtheitlichen, evolutionsfähigen Systemen folgende Aufgaben im Bereich der Erweiterbarkeit bearbeitet:

1. Dynamisierung des Übergangs zwischen Programmierung und Ausführung.
2. Festlegung charakteristischer Freiheitsgrade für konsistente Erweiterungsschritte sowie deren Granularität.
3. Systematische Lockerung von Bezügen zwischen Komponenten und ihren Realisierungen. Dabei sind besonders die Rollen von Übersetzer und Binder anzupassen.

Neben dem bereits erläuterten Aspekt für die Informationsgewinnung sind von der inkrementellen Systemkonstruktion weitere Effekte zu erwarten, die hier erwähnt, im folgenden jedoch nicht exploriert werden. So liefert die gesamtheitliche Systemkonstruktion Ansatzpunkte, um komplexe Systeme besser beschreiben und somit leichter beherrschen zu können. Darüber hinaus wird die Kombination interpretativer und übersetzerbasierter Verfahren gefördert. Partielle Korrekturen und Fehlerbehebungen an einem in Ausführung befindlichen, hoch optimierten und effizienten System werden ohne Sicherung des Zustands des gesamten Systems, Anhalten der Berechnung, statischer Rekonfiguration und Wiederaufsetzen möglich. Für das Ressourcenmanagement des Systems selbst, können die entwickelten Konzepte in einem weiten Rahmen zur Flexibilisierung der Managementmaßnahmen eingesetzt werden. So können durch Erweiterungen und Revisionen von Verwaltungsmaßnahmen leistungssteigernde Anpassungen an neue und veränderte Anforderungen vorgenommen werden.

### 1.1.2 Flexibles und integriertes V-PK-Ressourcenmanagement

Berechnungen, die auf hohem Abstraktionsniveau formuliert werden, sollen auf einer physisch verteilten Hardwarekonfiguration zur Ausführung gebracht werden. Hierfür ist eine Korrespondenz zwischen den abstrakten Einheiten der Rechenvorschrift und den realen Speicher-, Rechen- und Kommunikationseinheiten der Geräte herzustellen. Die Gesamtheit dieser Komponenten zuzüglich der Zwischenprodukte der Gesamttransformation auf dem Pfad zwischen Quelle und Ziel bildet die Menge der zu verwaltenden Ressourcen. Ein Verwaltungssystem – das Management – muß diese Ressourcen des verteilten Systems unter Berücksichtigung qualitativer und quantitativer Anforderungen verteilt verwalten. Die vorliegende Arbeit konzentriert sich auf die quantitative Leistungsfähigkeit, d. h. Optimierung des Raum-/Zeit-Verhaltens.

Die Verwaltung von Ressourcen kann als Teil der Problemlösung selbst oder systemintegriert durch Komponenten, die bereits Teil des Systems sind, vorgenommen werden. Durch die automatisierte Lösung von Verwaltungsaufgaben im systemintegrierten Fall werden Nutzer des Systems von schwierigen, repetitiven oder unmöglichen Aufgaben befreit oder zumindest dabei unterstützt. Um verteiltes Rechnen langfristig zu einem Normalfall der Nutzung moderner vernetzter Rechenanlagen machen zu können, ist die Entwicklung systemintegrierter Verfahren ein zentrales Ziel der Arbeiten im Projekt MoDiS.

**Anforderung 1** *Automatisiertes Ressourcenmanagement*

Universelle, systemintegrierte Verfahren sind bislang von den gestellten Anforderungen jedoch meist überfordert. Bereits im Hinblick auf ihre Effizienz verhalten sie sich eher

enttäuschend. Oft basieren Leistungssteigerungen, die für verteilte Betriebssysteme bei Nutzung verteilter Ressourcen gemessen werden, auf einer unakzeptabel schwachen Leistung relativ zu herkömmlichen Systemen. De facto entstehen dadurch vielmehr Leistungsverluste als -gewinne [LKBT92]. Nutzer verteilter Konfigurationen bevorzugen deshalb nach wie vor anwendungsintegrierte Lösungen [GS96, Gan97], obwohl selbst elementare Parameter wiederholt manuell bestimmt werden müssen [CBZ95][4].

Uniformität ist ein wesentlicher Faktor für die Leistungsdefizite automatisierter Verfahren. Die äußerst divergenten Eigenschaften von lokalen und entfernten Zugriffen verursachen eine enorme Variabilität der Anforderungen in verteilten Umgebungen. Ein elementares Beispiel ist der Transport von Datenobjekten zwischen Stellen. Konkrete Anforderungen können zwischen genau einem Lesezugriff auf ein feinstgranulares Datenobjekt bis hin zu komplexen zusammengesetzten Datenobjekten, die für aufwendige, lesende und schreibende Zugriffe benötigt werden, variieren. Durch die große Diskrepanz zwischen lokalen und entfernten Zugriffszeiten sind uniforme Verwaltungsverfahren aufgrund ihres extrem ungünstigem *worst-case* Verhaltens und evtl. zusätzlich schwachem *average case* unter diesen Voraussetzungen unbrauchbar. Die Möglichkeiten, lokal oder verteilt zu operieren, spannen allerdings auch ein erheblich größeres Spektrum an Realisierungstechniken und -mechanismen als bisher gewohnt auf. In [Win95] wird gezeigt, wie im Fall von Speicherzugriffen das Spektrum von RPC [Blo92] bis hin zu replizierten Datenobjekten eingesetzt werden kann, um anpassungsfähige Verfahren konstruieren und dadurch Effizienz gewährleisten zu können. Diese Forderung nach Flexibilität erhöht die Komplexität des Systementwurfs und den Realisierungsaufwand erheblich. Bei systemintegrierten Verfahren verschärft sich diese Situation durch den Wunsch nach Universalität. Auf Grund der a priori unvollständigen Information über die Bedürfnisse in der Zukunft durchzuführender Berechnungen müssen in diesem Fall weitere Freiheitsgrade vorgesehen werden, die eine dynamische Adaption zum Zeitpunkt der Ausführung erlauben. Bisher übliche Vorgehensweisen sind von der Komplexität dieser Flexibilität überfordert, mit der Konsequenz, daß entweder schwierige Verwaltungsaufgaben als anwendungsintegriertes Management auf die Nutzer verteilter Systeme verlagert werden oder gravierende Ineffizienzen starrer Verfahren unter bestimmten Voraussetzungen akzeptiert werden müssen – Verlust von Praktikabilität oder Qualität. Flexibilität ist somit für erfolgreiches V-PK-Ressourcenmanagement von vitaler Bedeutung. Um dieser Anforderung entsprechen zu können, soll das V-PK-Managements systematisch flexibilisiert werden.

**Anforderung 2** *Systematische Flexibilisierung*

Neben der mangelnden Flexibilität ist ein zweiter wichtiger Faktor für Ineffizienzen in der mangelnden Integrität von Verwaltungmaßnahmen zu beobachten. Der hohe Aufwand, der bei der Implementierung von Betriebssystemen zu leisten ist, führt primär dazu, daß sich entsprechende Arbeiten oft auf einen geringen Ausschnitt der möglichen Instrumentierungen konzentrieren. Gängige Vorgehensweisen sind das Hinzufügen zusätzlicher Schichten in Form von laufzeitunterstützenden Bibliotheken außerhalb des Kerns oder Detailmodifikationen der Kernfunktionalität. Die mangelnde Betrachtung des Kontextes und möglicher Alternativen hat jedoch eine Reihe negativer Auswirkungen. Dazu gehören Redundanzen, Inkonsistenzen, Inkompatibilitäten, unnötige - und unnötig komplizierte Lösungen. Beispiele dafür sind die mangelnde Unterstützung von Erweiterungen für *Multi-Threading* [IEE95] durch die Speicher-

[4]Siehe TSP-C und TSP-F: Eine minimale Abweichung bei der Suchtiefe, ab der sequentiell gerechnet wird, verlangsamt die Ausführung bereits um 50%.

verwaltung, Unklarheiten über die Verwendung von Registern[5] und die oftmals vielfach redundante und aufwendige Verwaltung von Seiten- und Objekttabellen in DSM-Systemen [Rei96]. Die mangelnde analytische Betrachtung der Instrumentierungsmöglichkeiten bzw. des Kontextes der gewählten Instrumentierung ist somit eine entscheidende Quelle für quantitative und qualitative Mängel. Die Forderung für das V-PK-Management, die sich daraus ableitet, ist die Integration von Maßnahmen und ihrer Instrumentierung zu einem gesamtheitlichen, leistungsstarken und qualitativ hochwertigen Managementsystem.

**Anforderung 3** *Integration der Managementmaßnahmen und ihrer Instrumentierungen*

Eine ausgezeichnete Rolle für die Integration kommt neben der Abstimmung von Maßnahmen der Verarbeitung von Wissen über Anforderungen und Angebote innerhalb des Managements zu. Während die geringe Flexibilität herkömmlicher Verfahren verhältnismäßig wenig Information benötigt um Realisierungsentscheidungen zu treffen, muß für die flexibleren Verfahren des V-PK-Managements wesentlich mehr und präzisere Information (siehe auch 1.1.2) erarbeitet und über Abstraktionsniveaus und Managementeinheiten hinweg integriert verwaltet werden.

Der Integrationsbegriff besitzt in MoDiS darüber hinaus durch die Sprachbasierung und die Gesamtsystemsicht zusätzliche Aspekte. Die Instrumentierung des Managements ist selbst als Teil des Gesamtsystems in dieses zu integrieren. Für diesen Zweck werden abgestufte Sprachkonzepte benötigt, die es erlauben, Abläufe auf zunehmend realisierungsnahen Abstraktionsebenen bis hin zu stellenbezogenen Maßnahmen zu beschreiben. Die daraus resultierende Gesamtsystem-Architektur sowie Konsequenzen für die Sprache INSEL sind zu untersuchen.

Mit der systematischen Flexibilisierung und Integration der Ressourcenverwaltung ist eine Grundlage geschaffen, um angepaßte und dynamisch anpaßbare Verfahren zu konstruieren, die universell für die automatisierte Verwaltung von Ressourcen in verteilten Systemen für parallele und kooperative Problemlösungen geeignet sind. Die Vorteile dieser Vorgehensweise sind durch die exemplarische Implementierung des Transformationsinstrumentariums als wichtiger Teil des Gesamtmanagements zu demonstrieren und zu evaluieren.

**Anforderung 4** *Realisierung des Transformationsinstrumentariums*

Der Begriff „Transformationsinstrumentarium“ subsumiert die Maßnahmen des Managements, die im wesentlichen den Text des auszuführenden Programmes transformieren. In herkömmlichen Systemen sind das der Übersetzer und der Binder. Während in zahlreichen Arbeiten an verteilten Betriebssystemen transformatorische Schritte nicht oder nur schwach in das Betriebssystem miteinbezogen werden, konzentriert sich die Realisierung des Managements in der vorliegenden Arbeit auf die Bedeutung dieser Klasse von Verwaltungsmaßnahmen. Transformatorische Maßnahmen definieren den Rahmen für die sich anschließenden Phasen des Managements und sind somit für die Leistungsfähigkeit des Gesamtmanagements von dominanter Bedeutung. Der Übersetzer besitzt darüber hinaus durch seine Fähigkeit der Quellcode-Analyse eine ausgezeichnete Rolle für die automatisierte Extraktion von Information. Die Gewinnung dieser Information und ihre Nutzung für die Flexibilisierung des V-PK-Managements ist auszuarbeiten.

Mit den Anforderungen 1–4 ist die Problemstellung bezüglich des Ressourcenmanagements erklärt. Im Zuge der Bearbeitung dieser Aufgabe ergeben sich zahlreiche neue Sichten

[5] z. B.: Solaris 2.5.1 verwaltet globale Register im Kern anders als in seinen Bibliotheken

zu grundlegenden Fragestellungen, die aufgrund ihres Umfangs in dieser Arbeit nicht abschließend beantwortet werden können. Dennoch werden diese fundamentalen Fragestellungen sowie Ergebnisse, die zu ihrer Beantwortung beitragen können, in dieser Arbeit diskutiert und im folgenden kurz motiviert.

## 1.2 Langfristige Ziele

Der Ursprung der beobachteten Schwierigkeiten, automatisiertes und dennoch leistungsfähiges verteiltes Ressourcenmanagement zur Verfügung zu stellen, ist in den Produktionsfaktoren *Methodik*, *Realisierungsaufwand* und *Kreativität* zu suchen. Die Konstruktion von Betriebssystemen (*OS* für engl.: operating system) wird von diesen Faktoren geprägt.

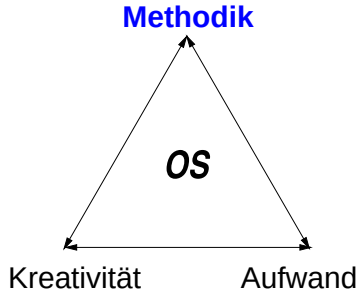


Abbildung 1.2: Fundamentale Einflußfaktoren

- Konzepte und Verfahren müssen angepaßt an die Anforderungen **kreativ** entwickelt werden. Dazu gehört das Schaffen neuer Fähigkeiten, Beheben erkannter Mängel und Finden neuer, besserer Lösungswege. Der hohe Aufwand und die mangelnde Methodik bei dem Entwurf und der Realisierung von Managementverfahren fördert empirisches Vorgehen und behindert Kreativität.

- Der enorme **Aufwand**, der mit den primitiven Methoden der Systemprogrammierung [See74] verbunden ist, führt dazu, daß die Implementierung von Managementkonzepten gravierenden Einschränkungen unterliegt. Die Tragweite eingegangener Kompromisse ist mangels einer fundierten Konstruktionsmethodik in der Regel nicht bekannt. In der Praxis werden mögliche Lösungswege schon früh aufgrund eines vermuteten hohen Realisierungsaufwands nicht weiter verfolgt. Dieses Hemmnis muß u. a. durch die Entwicklung schnellerer Realisierungsverfahren, erhöhter Wiederverwendbarkeit von Software und besseren Werkzeugen abgeschwächt werden.

- Um das große Spektrum bisher bekannter Verfahren, von grundlegenden Architekturfestlegungen bis hin zu Detail-Optimierungen, nutzen sowie notwendige Neukonstruktionen erkennen und initiieren zu können, müssen Lösungswege gekannt und analytisch betrachtet werden. Dem Entwickler eines verteilten Betriebssystems steht für diesen Zweck bislang keine **Methodik** zur Verfügung. Es existiert weder eine systematische Aufbereitung möglicher Entscheidungsalternativen noch sind fundierte Kriterien für die Auswahl zwischen diesen bekannt. Für die erfolgreiche Konstruktion immer komplexerer Systeme muß dieses Defizit durch neue systematische Vorgehensweisen behoben werden.

Durch eine Verbesserung der Methodik kann darüber hinaus der Aufwand reduziert und die Entfaltung kreativer Lösungsansätze unterstützt werden.

Im Vergleich zu anderen Disziplinen, wie z. B. dem Übersetzerbau [SF85, WM96, Muc97] und der Entwicklung grafischer Benutzungsoberflächen, besteht bei den Betriebssystemen gerade in Bezug auf den dominanten Faktor Methodik großer Nachholbedarf. Während Übersetzer, basierend auf den Erkenntnissen aus der Theorie formaler Sprachen, systematisch entworfen und mit generativen Verfahren realisiert werden, werden Betriebssysteme in der Regel nach dem *trial & error* Prinzip entworfen und unter großen Anstrengungen manuell implementiert. Der daraus resultierende Aufwand hat weitreichende Konsequenzen und führt letztendlich zu einer Betrachtung von Betriebssystemen, in der die Ziele des Ressourcenmangements in den Hintergrund treten und von den Eigenschaften künstlich geschaffener Instrumente verdrängt werden. Verloren geht dabei das Bewußtsein, daß es sich bei den vermeintlichen Rahmenbedingungen meist um veränderbare Artefakte handelt. Typische Fragen, die dieses Problem belegen, diskutieren phänomenologisch Aspekte, wie z. B. die Integration eines Dienstes in den Kern contra Auslagerung in Benutzerprozesse. Tatsächlich wäre jedoch die Frage zu stellen, wie die ursprüngliche Zielsetzung unter den Randbedingungen der Physik und den veränderbaren Eigenschaften der Artefakte (Benutzerprozeß und Kern) erreicht werden kann. Der Einfluß des hohen Realisierungsaufwands hemmt jedoch die Bereitschaft, die Eigenschaften der Artefakte selbst in Frage zu stellen und bei Bedarf zu ersetzen.

Andere Konsequenzen aus der unbefriedigenden Methodik und dem hohen Aufwand demonstrieren Bemühungen schlanke Systeme zu konstruieren, wie Oberon [WG92] und Plurix [Sch99]. Motiviert von der Vielfalt anscheinend unnötiger, redundanter und inkompatibler Maßnahmen in existierenden Systemen, werden zunächst Systeme mit ggf. reduzierter Funktionalität von Grund auf neu konstruiert. Spätere Wünsche nach zusätzlicher Funktionalität und besserer Leistung werden allerdings wie gewohnt durch Hinzufügen zusätzlicher Fähigkeiten befriedigt. Für diese Erweiterungen gibt es keine Methodik, mit der die Defizite, die durch den minimalen Ansatz behoben werden sollen, vermieden werden können. Das ursprüngliche Ziel, ein schlankes Betriebssystem zu konstruieren, dessen Verwaltungseinheiten genau die an sie gestellten Anforderungen erfüllen, scheint auch so nicht erreichbar.

Um Betriebssysteme zielgerichtet konstruieren und mit akzeptablem Aufwand realisieren zu können muß die Methodik ihrer Konstruktion deutlich verbessert werden. Die großen Schwierigkeiten, den Übergang von zentralen zu verteilten Systemen zu vollziehen, sind ein drastisches Beispiel dafür, daß mit den bisher eingesetzten Verfahren dieser Anstieg an Komplexität nicht mehr beherrscht wird. Um mit den steigenden Anforderungen schritthalten zu können, muß eine Systematisierung der Konstruktionsmethodik erfolgen. Basierend auf dieser Systematisierung muß der Einfluß des Aufwands durch die Automatisierung der Implementierung von Betriebssystemen abgeschwächt werden. Konträr zu dieser Vision fehlen allerdings im Vergleich mit dem Übersetzerbau und den formalen Sprachen bei den Managementsystemen derzeit bereits Grundlagen zur präzisen Charakterisierung ihrer Eigenschaften.

Der weit gefaßte Rahmen von MoDiS ermöglicht es, kreative Positionen in der Betrachtung von Betriebssystemen einzunehmen und Ansätze für die dringend erforderliche Systematisierung ihrer Konstruktion zu erarbeiten. Unter diesem Aspekt ist auch die umfangreiche Aufgabenstellung der vorliegenden Arbeit zu verstehen. Im Zuge der exemplarischen Ausarbeitung einer flexiblen, integrierten und leistungsfähigen Ressourcenverwaltung für V-PK-Systeme werden die Freiheitsgrade der gesamtheitlichen Vorgehensweise genutzt, um Beiträge zur Verbesserung der Methodik der Konstruktion von Ressourcenverwaltungen zu erarbeiten.

## 1.3 Lösungsweg und Gliederung der Arbeit

Nach der Motivation dieser Arbeit und der Erörterung der Aufgabenstellung in den vorangegangenen Abschnitten erfolgt hier eine Beschreibung der Gliederung dieser Niederschrift.

Die Reihenfolge der Darstellung entspricht nicht der chronologischen Ordnung, in der die Arbeiten durchgeführt wurden. Der Umfang und die große Tragweite einiger Aspekte der Aufgabenstellung erlaubten es nicht, sequentiell Lösungen von Teilproblemen zu erarbeiten. Durch konkrete Implementierungen abstrakt entworfener Konzepte und anschließender Evaluation wurden wertvolle Erfahrungen gesammelt, die ihrerseits Wirkung auf die konzeptionellen Grundlagen hatten und wiederholt zu Verfeinerungen der gesamten Betrachtung geführt haben. Dieses Fortschreiten des Erkenntniszugewinns und der Realisierungsarbeiten entspricht einer spiralförmigen Vorgehensweise mit zahlreichen Iterationen über Entwurfs-, Realisierungs- und Evaluationsphasen. Trotz des inhärent hohen Aufwands des Spiralmodells, der aus der repetitiven Realisierung unvollständiger und somit veränderlicher Konzepte entsteht, wurde diese Vorgehensweise gewählt, um ein möglichst großes Spektrum an Einflußfaktoren und Problemstellungen erkennen und bearbeiten zu können. Die Niederschrift dieser Arbeit ist in zwei Teile gegliedert.

**Teil I** analysiert die Ausgangsbasis, auf deren Grundlage die Aufgabe gestellt und Konzepte und Verfahren zu ihrer Lösung erarbeitet wurden.
Nach der bereits erfolgten Darstellung der Problemstellung und Zielsetzung in **Kapitel 1** werden im **Kapitel 2** wichtige Konzepte aus dem Bereich der verteilten und erweiterbaren Systeme klassifiziert und bezüglich ihrer Eignung für das Erreichen der qualitativen und quantitativen Ziele von V-PK-Systemen untersucht. Entsprechend dem gesamtheitlichen Ansatz werden Rechnerarchitekturen, mögliche Anwendungsgebiete, Methoden der Programmierung und die Verwaltung von Ressourcen in verteilten Umgebungen betrachtet. Im Anschluß an diese Untersuchungen wird in **Kapitel 3** mit den konzeptionellen Grundlagen des Projekts MoDiS und der Sprache INSEL der Rahmen für alle weiteren Entwicklungsschritte in dieser Arbeit festgelegt. Teil I schließt in **Kapitel 4** mit der Betrachtung einiger ausgewählter Repräsentanten verwandter Forschungsprojekte, die ebenfalls Fragestellungen der Erweiterbarkeit, der Flexibilisierung oder der Integration des verteilten Ressourcenmanagements untersuchen. Eine Sonderstellung nimmt bei dieser Recherche die Evaluation vorangegangener prototypischer Implementierungen der MoDiS-Architektur ein, aus der wichtige Anforderungen für das in dieser Arbeit zu konstruierende Management abgeleitet werden.

Basierend auf den Beobachtungen aus Teil I befaßt sich **Teil II** mit den Grundlagen der systematischen Konstruktion einer Ressourcenverwaltung für gesamtheitliche V-PK-Systeme, erörtert die Realisierung eines effizienten V-PK-Managements und demonstriert anhand arbeitsfähiger Verfahren die quantitativen und qualitativen Verbesserungen, die mit den Maßnahmen der Flexibilisierung und Integration erzielt wurden.
**Kapitel 5** erläutert zunächst grundlegende Konzepte, die sowohl für das Ressourcenmanagement als auch die inkrementelle Konstruktion eines strukturierten Gesamtsystems von zentraler Bedeutung sind. Im Mittelpunkt steht die Definition des Begriffs „Ressource“ und die systematische Betrachtung von Bezügen zwischen Ressourcen. **Kapitel 6** kombiniert dieses Fundament mit den bereits existierenden Sprachkonzepten von INSEL und führt neue Konzepte für Komponentenfamilien und die dynamische Vervollständigung von INSEL-Spezifikationen ein. Mit diesem Konzeptevorrat steht ein Instrumentarium zur Verfügung, mit dem gesamtheitlich strukturierte Systeme inkrementell konstruiert werden können. Auf dieser Grundlage wird an-

schließend in **Kapitel 7** die im Projekt MoDiS entworfene Management-Grobarchitektur konkretisiert und ein leistungsfähiges Instrumentarium für die automatisierte Realisierung global strukturierter INSEL-Systeme entworfen. Dabei wird insbesondere das integrierte Transformationsinstrumentarium erläutert, das im Rahmen dieser Arbeit entwickelt und realisiert wurde. Schwerpunkte bilden der eng in das Gesamtmanagement integrierte INSEL-Übersetzer *gic* sowie der flexible Binder *FLink*, der als Paradebeispiel für das flexible Binden von Ressourcen dient. Um diese aufwendigen Implementierungsarbeiten mit moderatem Aufwand durchführen zu können sowie kostspielige „Neuerfindungen des Rades“ zu vermeiden und dennoch nicht die Restriktionen existierender Systemprogramme akzeptieren zu müssen, wurden bestehende Programme wiederverwandt und durch Modifikation an die Anforderungen in V-PK-Systemen angepaßt. Die Vorzüge des verfolgten Ansatzes werden in **Kapitel 8** anhand ausgewählter Beispiele evaluiert. Die Arbeit schließt in **Kapitel 9** mit einer Zusammenfassung der erzielten Ergebnisse und einem Ausblick auf mögliche weitere Arbeitsschritte.

Im **Anhang** können bei Interesse die konkrete und abstrakte Syntax der Sprache INSEL, Quellen von INSEL-Beispielprogrammen sowie Ergebnisse von Leistungsmessungen nachgelesen werden.

# Teil I

# Analyse

# Kapitel 2

# Taxonomie verteilter und erweiterbarer Systeme

Die Ziele, die mit parallelen und verteilten Rechensystemen verfolgt werden, sind breit gefächert. Gleiches gilt für die Methoden, mit deren Hilfe sie programmiert werden, und ähnliches auch für die Rechnerarchitekturen. Um ein universell einsetzbares, integriertes bzw. gesamtheitliches verteiltes System konstruieren zu können, sind grundlegende Kenntnisse der Einflußfaktoren und ihrer Wechselwirkungen erforderlich. In diesem Kapitel wird ein Teil dieses Wissens erarbeitet, indem wesentliche Konzepte aus den Forschungsgebieten der verteilten und parallelen Systeme zusammengetragen und wichtige Begriffe eingeführt werden. Die Ergebnisse dieser Recherche ermöglichen es, die bearbeitete Problemstellung – Konstruktion eines integrierten und flexiblen verteilten Managements – sowie die dabei erzielten Ergebnisse zu diskutieren. Die Darstellung konzentriert sich dabei auf Eigenschaften, die zum Verständnis der Problematik des automatisierten verteilten Managements beitragen.

Die Begriffe *parallel* und *verteilt* wurden bisher verwendet, ohne zuvor eingeführt worden zu sein. In der Literatur werden sie durchaus uneinheitlich gebraucht und lassen somit unterschiedliche Interpretationen zu. Die Begriffe nebenläufig und parallel werden in der vorliegenden Arbeit synonym verwendet um Unabhängigkeit zweier Arbeitsabläufe in der Dimension Zeit zum Ausdruck zu bringen. Die Charakterisierung verteilter Systeme erfolgt meist bottom-up orientiert an den Eigenschaften der physischen Ressourcen. Da einige wichtige Probleme der Ressourcenverwaltung in verteilten Umgebungen in engem Zusammenhang mit dem physisch verteilten Speicher stehen, wird im Bereich der Rechnerarchitektur die Organisation des Speichers als primäres Kriterium für die Klassifikation verteilter Konfigurationen herangezogen. Andere bottom-up geprägte Begriffe die in Zusammenhang mit dieser Sichtweise in der Literatur auftreten sind „lose gekoppelte“ Rechner, „Multicomputer“ oder „Clusters of Workstations“ (COW). In den Bereichen Betriebssysteme und Rechnernetze werden demgegenüber *verteiltes System* und *Rechnernetz* zur gegenseitigen Abgrenzung verwendet. Als Kriterium dient dabei der Grad an Opazität[1]. Eine präzisere Abgrenzung verteilter Systeme, an der

[1] **Transparenz** wird im Kontext verteilter Systeme oft in dem Sinne von Durchsichtigkeit gebraucht um „nicht wahrnehmbar“ auszudrücken. Gemeint ist damit in aller Regel, daß die physische Verteilung nicht zu Tage tritt. Diese Begriffsprägung ist zumindest mißverständlich. Zum einen kann unter Transparenz auch das Gegenteil, das Durchscheinen von Details verstanden werden und zum anderen stellt sich die Frage, selbst wenn man „transparent“ im Sinne von Durchsichtigkeit interpretiert, was durch das „durchsichtige“ Gebilde zu sehen ist. Im Zusammenhang mit einem verteilten Betriebssystem könnte das nichts oder die physisch verteilten Hardwareressourcen sein, d. h. keine Aussage oder genau die, die nicht beabsichtigt war.

sich auch die vorliegende Arbeit orientiert, wurde in [Ens78] mittels folgender fünf Kriterien vorgenommen:

1. *Vielzahl und Vielfalt* physischer und logischer Ressourcen.

2. *Physische Verteilung* der Ressourcen, wobei eine Verbindung über ein Kommunikationsnetz besteht.

3. Ein *Betriebssystem* auf hohem Abstraktionsniveau, das die Kontrolle über die verteilten Ressourcen unifiziert und integriert. Individuelle Prozessoren führen ein eigenes ggf. einzigartiges lokales Betriebssystem aus.

4. *Opazität* von Komponenten und Diensten bezüglich deren Lokalisation.

5. *Kooperative Autonomie* kennzeichnet die Abläufe und Interaktion sowohl der physischen als auch der logischen Ressourcen.

Der dort verwendete Begriff der „logischen“ Ressourcen ist unpräzise. Gemeint sind die Betriebsmittel des Systems, die nicht unmittelbar Komponenten der Hardware, sondern Resultat einer Abstraktionsbildung sind. Um diesen Sachverhalt auszudrücken, werden die oben genannten „logischen“ Ressourcen im weiteren als abstrakte- oder virtuelle – im Sinne von nicht in der Wirklichkeit existierende [Dud97] – Ressourcen bezeichnet.

Im Laufe der Arbeiten an dem integrierten und flexiblen V-PK-Management hat sich darüber hinaus Heterogenität der Kooperation in verteilten Systemen als besonders bedeutsam herausgestellt. Wenngleich diese Beobachtung keine abschließende Definition ermöglicht, ist es nützlich, verteilte Systeme auf diese Weise zu betrachten.

**Behauptung 2.1 (Verteiltes System)**
*Ein verteiltes Rechensystem ist durch die enorme Vielfalt und Variabilität der Kosten für Interaktion zwischen den Komponenten des Systems charakterisiert.*

Die speziellen Anforderungen verteilter Systeme an das Ressourcenmanagement kommen in dieser Sichtweise gut zum Ausdruck, was für das Verständnis der Methoden, die in dieser Arbeit entwickelt werden, hilfreich ist. Die u. U. dynamische Veränderlichkeit der Kosten für die Kooperation zwischen nebenläufigen Berechnungen bzw. Objekten des Systems um Größenordnungen erfordert eine Vielzahl flexibler Verfahren und präzise Information für deren Steuerung. Sie ist daher verantwortlich für viele Detailprobleme, wie z. B. Ermittlung der aktuellen Lastsituation [BASK95] oder Notwendigkeit der Replikation mit der Folge von Konsistenzproblemen, et cetera. Die dynamische Variabilität und die großen Leistungsdifferenzen beruhen ihrerseits selbstverständlich auf den technischen Eigenschaften der Rechenanlagen. In den folgenden Abschnitten werden deshalb – entgegen der ansonsten in MoDiS top-down orientierten Vorgehensweise – zunächst die technischen Grundlagen verteilter Rechensysteme erläutert, bevor mögliche Nutzungsziele, Programmiermodelle und Ansätze für das Ressourcenmanagement diskutiert werden.

Aus diesem Grund wird in dieser Arbeit der Begriff „**opak**“ (lat.: undurchsichtig) und substantiviert „**Opazität**“ verwendet. Opak bringt vermutlich weniger mißverständlich die gewünschte Aussage zum Ausdruck. Setzt ein verteiltes Betriebssystem Opazität durch, so bleibt dem Nutzer dieses Systems die physische Verteilung verborgen.

## 2.1 Technische Grundlagen

Die technischen Eigenschaften der Zielarchitektur, auf der parallele und kooperative Berechnungen mit variierender Granularität zur Ausführung kommen sollen, sind invariante Angebote, die dem Management zur Erfüllung seiner Aufgabe zur Verfügung stehen. Invariant drückt aus, daß davon ausgegangen wird, daß sich ihre Eigenschaften (Zahl, Fähigkeiten) während der Ausführung des Systems nicht verändern. Wie bereits in der Einleitung erwähnt wurde, werden Netze von Arbeitsplatzrechnern wegen ihres günstigen Preis-/Leistungsverhältnis und ihrer großen Verbreitung als Zielarchitekturen für die Realisierung von V-PK-Systemen angestrebt. Zur Vereinfachung der Aufgabenstellung werden im weiteren lediglich homogene Netze betrachtet, wobei sich die Homogenität auf die Prozessoren der verteilten Rechnerarchitektur bezieht. Die Rechen- und Speicherfähigkeiten solcher Anlagen werden im folgenden näher erläutert.

Der dritte Aspekt einer verteilten Konfiguration, das Kommunikationsnetz, muß hier nicht detailliert behandelt werden, da davon ausgegangen wird, daß ein gewöhnliches lokales Netz eingesetzt wird; z. B. 10 MBit/s Ethernet oder 100 MBit/s FastEthernet. Die bedeutsamste Eigenschaft dieser Verbindungsart ist ihre um bis zu $10^{-6}$ (siehe Tab. 2.1) schwächere quantitative Leistung als Kommunikationsmedium gegenüber Kooperation mittels einem physisch gemeinsamen Speicher. Diese inhärente Diskrepanz zwischen lokaler und entfernter Kooperation ist der treibende Faktor für die notwendige Flexibilisierung des Managements in verteilten Umgebungen.

### 2.1.1 Rechenfähigkeit

Die wohl bekannteste Klassifikation paralleler Rechnerarchitekturen basiert auf der Betrachtung von Instruktions- und Datenströmen [Fly72, FR96]. Ein Strom ist eine Sequenz von Elementen. Die Elemente der Ströme sind je nach Strom Sequenzen von Daten oder Sequenzen von Instruktionen. Ströme sind voneinander unabhängig. Für die Verarbeitung von Strömen existieren in diesem Modell vier verschiedene Kombinationsmöglichkeiten:

**SISD** — single instruction, single data stream
: SISD entspricht der Verarbeitung in gewöhnlichen Uniprozessoren. Zu jedem Zeitpunkt wird genau ein Element aus dem Instruktions- und Datenstrom verarbeitet. Nebenläufigkeit wird in dieser Klasse durch Fließbandverarbeitung[2] – nebenläufige Verarbeitung verschiedener Phasen der Instruktion – und Parallelität auf Instruktionsebene (ILP: Instruction Level Paralellism) erzielt.

**SIMD** — single instruction, multiple data streams
: Repräsentanten dieser Klasse sind Feldrechner wie die CM-200 von Thinking Machines Corporation und Vektorrechner (z. B. Fujitsu VPP 700 mit $\leq$ 256 Prozessoren). Alle Prozessoren führen genau eine Instruktion aus, wobei die beteiligten Prozessoren oder Rechenwerke jedoch auf unterschiedlichen Daten operieren können. Dieses Modell ist speziell für die Bearbeitung regulärer Datenstrukturen wie Felder und Matrizen geeignet.

[2] engl.: pipelining

**MISD** — multiple instructions, single data stream
Abstrakt betrachtet besteht eine MISD Architektur aus einer Anzahl funktionaler Einheiten, die einen Datenstrom fließbandartig verarbeiten. Im Gegensatz zur Fließbandverarbeitung bei Vektorrechnern sind die parallel verarbeiteten Operationen vollständige Instruktionen anstelle von Fragmenten einer Instruktion. Obwohl das Prinzip der parallelen Verarbeitung mehrerer Instruktionen auf genau einem Datum leicht vorstellbar ist und der Entwurf entsprechender Prozessoren möglich erscheint, ist das Interesse an diesem Ausführungsmodell gering. Der Grund hierfür ist der Mangel an geeigneten Konzepten der Programmierung.

**MIMD** — multiple instructions, multiple data streams
Bei einer MIMD Architektur agieren die beteiligten Prozessoren weitestgehend unabhängig. Jeder Prozessor verarbeitet einen eigenen Instruktions- und Datenstrom parallel. Dieses Modell entspricht auf sehr natürliche Weise dem Konzept der Dekomposition eines Programms in parallele Teilberechnungen. Neben der gewonnenen Flexibilität entstehen durch die vollständige Unabhängigkeit der Verarbeitung von Strömen zahlreiche neue Probleme, wie etwa die Festlegung und Durchsetzung eines Konsistenzmodells und Kohärenz der Caches falls über einen gemeinsamen Speicher kommuniziert wird (s. u.).

MIMD Konfigurationen werden *homogen* genannt, sofern alle Prozessoren identische Eigenschaften besitzen, anderenfalls *heterogen*. Die als Zielplattform für V-PK-Systeme genannten, homogenen Rechnernetze ordnen sich in dieses Schema als homogene MIMD-Architekturen ein.

### 2.1.2 Speicherfähigkeit

MIMD Konfigurationen existieren in vielen Varianten, die sich durch den Zugriff auf den Speicher und das Verbindungsnetz zwischen den Prozessoren unterscheiden. Da die Speicherorganisation sowohl für das Management als auch das Programmiermodell von besonders großer Bedeutung ist, dient es als primäres Merkmal für die weitere Differenzierung von MIMD-Architekturen. Die Kriterien, die zur Kategorisierung herangezogen werden, orientieren sich sowohl an qualitativen als auch an quantitativen Aspekten des Speichersystems.

#### 2.1.2.1 Quantitative Merkmale

Wichtige quantitative Attribute von Speicher betreffen ihre Kapazität, Datenbreite und Zugriffszeiten. Verschiedene Typen von Parallelrechner werden in erster Linie anhand der Geschwindigkeit von Speicherzugriffen unterschieden. Die beiden wichtigen Eckdaten für die Beurteilung des Speicherzugriffsverhaltens sind Latenz und Bandbreite.

**Definition 2.2 (Latenz)**
*Die Latenz L eines Kommunikationssystems ist die minimale Zeitspanne, die zur Übermittlung eines Datums benötigt wird, inklusive dem Aufwand, der für die Initiierung des Transfers zu leisten ist.*

Bei der Betrachtung des Speichers handelt es sich bei den betrachteten Kommunikationspartnern um Prozessoren und Speicherworte. Die Latenz gibt in diesem Fall die minimale Zeit an, die ein Prozessor benötigt, um auf ein elementares Speicherwort zuzugreifen.

Abstrahiert von dem physischen Speicher, ist Latenz für paralleles - und verteiltes Rechnen generell eine wichtige Größe, da sie eine untere Schranke für die aus Leistungsgesichtspunkten sinnvolle Granularität separierbarer Berechnungen festlegt. Dies gilt primär für Transporte von Speicherobjekten und davon abhängig sekundär auch für die mögliche Verteilung paralleler Berechnungen inklusive ihrer Code- und Datensegmente. Wenn eine Berechnungsvorschrift lokal schneller ausgeführt werden kann als der Transport des zugehörigen Programmtextes und der Datenobjekte der Berechnung dauern würde, so ist die lokal nebenläufige bzw. sequentielle Ausführung zu bevorzugen. Unter der günstigen Voraussetzung, daß sowohl der Programmtext als auch die Datenobjekte geringes Volumen besitzen, bedeutet dies, daß die Dauer der Berechnung $t > 2 * L$ sein muß, um eine Ausführung auf einem entfernten Prozessor zu rechtfertigen[3].

**Definition 2.3 (Bandbreite)**
*Die Bandbreite B eines Kommunikationssystems ist die maximale Menge an Daten, die innerhalb einer Zeiteinheit übertragen werden kann.*

Zeiten für die Initiierung und Terminierung der Kommunikation gehen nicht in die Bandbreite, sondern in die Latenz ein. Hohe Bandbreiten werden für die effiziente Übertragung großer Nachrichten sowie kontinuierlicher Datenströme mit großem Volumen benötigt. Bei Übertragung individueller, feingranularer Datenobjekte ist die Bandbreite jedoch von untergeordneter Rolle. In diesem Fall ist die Latenz des Kanals dominant. Bei der Klassifikation von Parallelrechnern besitzt die Bandbreite gegenüber der Latenz ebenfalls eine untergeordnete Bedeutung.

#### 2.1.2.2 Qualitative Merkmale

Parallelrechner, die über einen gemeinsamen Speicher verfügen, werden *eng gekoppelte* Rechner, oder *Multiprozessorsysteme* genannt. Bei Abwesenheit eines gemeinsamen Speichers spricht man dahingegen von einer *losen Koppelung* oder *Multicomputersysteme* [Tan92, Hwa93]. Zu klären ist allerdings die Frage, was unter einem gemeinsamen Speicher zu verstehen ist. In der Regel ist damit die Existenz eines gemeinsamen Namens- bzw. Adreßraumes $A = \{a_0, a_1, \ldots, a_{n-1}\}, a_i \in \mathbb{N}$ gemeint, mit dem die Speicherworte $\{s_0, s_1, \ldots, s_{n-1}\}$ des Speichers $S$ von den Prozessoren $p \in P$ identifiziert werden können. Charakteristisch für einen gemeinsamen Speicher ist:

**globale Eindeutigkeit**
: Die Abbildung $m : A \rightarrow S$ von Adressen auf Speicherworte ist unabhängig von den Prozessoren.

**globale Erreichbarkeit**
: Die Abbildung $m : A \rightarrow S$ von Adressen auf Speicherworte ist auf allen Prozessoren für alle $a \in A$ definiert.

Gilt jedoch $\exists s \in S$, die nicht von allen $p \in P$ erreichbar sind oder besitzen die $a \in A$ auf den verschiedenen Prozessoren unterschiedliche Interpretationen, so existiert auf dieser Betrachtungsebene kein gemeinsamer, sondern ein verteilter Speicher. Die Grenze zwischen gemeinsam und verteilt verläuft in der Praxis fließend, da die strengen Forderungen nach globaler

[3] notwendig, aber nicht hinreichend

Eindeutigkeit und Erreichbarkeit oft abgeschwächt werden. Ein global eindeutiger Adreßraum wird auch *Ein-Adreßraum* genannt. Voraussetzung für einen Ein-Adreßraum ist, daß er mächtig genug ist, um alle Speicherzellen des Systems adressieren zu können. 64Bit weite Adreßräume, die von immer mehr gängigen Architekturen, wie der SUN SPARC V9 [WG94], unterstützt werden, erfüllen diese Voraussetzung. Der Vorteil eines global eindeutigen Adreßraumes ist seine einfache Nutzbarkeit, d. h. ein sehr einfaches Programmiermodell, das sich aus der Uniformität des Namensraums ergibt. Gleichzeitig bietet ein Ein-Adreßraum auch Effizienzvorteile, da Identifikatoren nicht transformiert werden müssen, wenn sie zwischen den Prozessoren ausgetauscht werden. Existiert kein Ein-Adreßraum, so muß auf einer höheren Abstraktionsebene ein zusätzlicher Namensraum inklusive Operationen für die Transformation von Identifikatoren zwischen den Namensräumen konstruiert werden, um die ursprüngliche Separation zu überwinden. Der Vorteil einer solche Separation ist auf der anderen Seite die sehr einfache und flexible Erweiterbarkeit der Konfiguration.

Verfügt eine Konfiguration über einen gemeinsamen Adreßraum $A$, so ist weiter zu untersuchen, wie er gebildet wird. Häufig wird der Speicher $S$ *partitioniert*, um separate Module getrennt realisieren zu können. Diese Zerlegung bewirkt, daß Adressen $a \in A$ an bestimmte Prozessoren gebunden sind. Diese Partitionierung kann *statisch*, d. h. einmalig, oder *dynamisch*, d. h. variabel über die Zeit, erfolgen. Ein statisch partitionierter Adreßraum ist leicht zu realisieren, schränkt jedoch die Mobilität der Daten stark ein, bzw. erschwert sie. Um die Realisierung von Datenobjekten im Speicher an die Anforderungen der Nutzer anpassen zu können, müssen Datenobjekte im Speicher migrieren können. Deshalb ist statisches Partitionieren zu vermeiden.

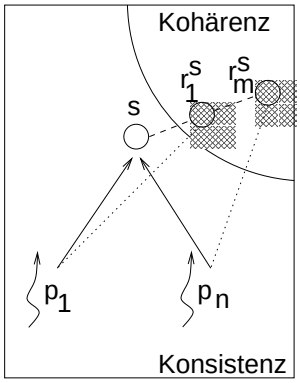


Abbildung 2.1: Konsistenz und Kohärenz

Die beiden letzten Merkmale, die hier zur Klassifikation von verteilten - und parallelen Architekturen eingeführt werden und auch für die weitere Betrachtung verteilter Systeme benötigt werden, sind die wichtigen Begriffe *Konsistenz* und *Kohärenz*. In der Literatur werden Konsistenz und Kohärenz unterschiedlich verwendet und teilweise nicht voneinander getrennt. Abbildung 2.1 illustriert die Abgrenzung der beiden Begriffe voneinander, so wie sie in dieser Arbeit zu verstehen ist. $p_1$ bis $p_n$ sind Prozessoren, die auf einem gemeinsamen Datum $s$ operieren. Das Konsistenzmodell legt fest, welchen Zustand $s$ aus der Sicht der unterschiedlichen $p_i$ besitzt als Eigenschaft von $s$ fest. Das Datum $s$ kann seinerseits aus Effizienzgründen in unterschiedlichen Speichern $r_1^s, \ldots, r_m^s$, z. B. einer Speicherhierarchie, realisiert sein. Mit der Kohärenz ist festgelegt, wie sich die unterschiedlichen Repräsentanten $r_i^s$ von $s$ zueinan-

der verhalten. Indem die Prozessoren unterschiedliche Repräsentanten von $s$ für ihre Zwecke nutzen, sind Konsistenz und Kohärenz voneinander abhängig.

**Konsistenz**
Ein Konsistenzmodell definiert Bedingungen für Widerspruchsfreiheit. In Bezug auf den Speicher bezieht sich die Widerspruchsfreiheit auf den Zustand der Daten aus der Sicht der Prozessoren, wobei je nach Abstraktionsebene der Betrachtung abstrakte Berechnungen an die Stelle der physischen Prozessoren treten. Beispiele für Konsistenzmodelle sind:

- Sequentielle Konsistenz (*sequential consistency*) [Lam79]: Ein Speichersystem heißt sequentiell konsistent, wenn das Resultat einer beliebigen Programmausführung das selbe ist, als würden die Operationen aller Prozessoren sequentiell ausgeführt werden, wobei sich die Operationen eines Prozessors in dieser Sequenz in der selben Reihenfolge befinden müssen, wie sie durch das Programm für diesen Prozessor festgelegt sind.
  Dieses Modell entspricht in etwa der Sicht auf den Speicher, die man von SISD-Architekturen gewohnt ist. Für den Programmierer ist es deshalb einfach zu handhaben. Auf der anderen Seite ist die inhärent schwache Effizienz dieses Modells problematisch [LS88].

- Schwache Konsistenz (*weak consistency*) [DSB86]: Diese Form der Konsistenz schwächt die verhältnismäßig strenge Forderung der sequentiellen Konsistenz ab. Zusätzlich zu Schreib- und Leseoperationen werden Synchronisationsoperationen und -variablen in die Betrachtung miteinbezogen, um die Bedingung formulieren zu können, daß der Zustand des Speichers nur an Synchronisationspunkten widerspruchsfrei sein muß. Im einzelnen bedeutet das:

  1. Zugriffe auf Synchronisationsvariablen sind sequentiell konsistent.
  2. Auf Synchronisationsvariablen darf erst dann zugegriffen werden, wenn alle vorhergehenden Schreibzugriffe abgeschlossen sind.
  3. Lese- und Schreibzugriffe auf Daten sind erst dann erlaubt, wenn alle vorhergehenden Zugriffe auf Synchronisationsvariablen abgeschlossen wurden.

Darüber hinaus existieren zahlreiche weitere Konsistenzmodelle, wie *atomic consistency*, *causal consistency*, *processor consistency*, *release consisency*, *lazy release consisency* und *entry consistency*, mit denen auf unterschiedlichen Wegen versucht wird, eine Balance zwischen der Qualität des Programmiermodells und einer möglichen effizienten Realisierung zu finden. In [de 99] werden diese Modelle und die oben informell beschriebenen präzise erläutert und gegenübergestellt.

**Kohärenz**
Von einem kohärenten Speicher wird erwartet, daß der von einer Leseoperation erhaltene Wert dem letzten geschriebenen Wert entspricht [Ber98]. Auch von dieser Forderung gibt es entsprechend den unterschiedlichen Anforderungen der Konsistenzmodelle Abschwächungen. Mit der Kohärenz werden demnach Forderungen und Verfahren festgelegt um ein Konsistenzmodell, das aus Sicht des Programmiermodells definiert wurde, durchzusetzen. Kohärenzprobleme treten dann auf, wenn Daten aus Effizienzgründen in Caches repliziert werden. Um die geforderte Konsistenz in diesem Fall durchzusetzen, müssen Schreiboperationen nach definierten

Gesetzmäßigkeiten auch auf Replikate wirken, bevor nachfolgende Lese- oder Schreiboperationen passieren dürfen. Charakteristisch für Koheränzprotokolle ist die Festlegung einer Granularität von Speicherblöcken, die als Einheiten der Kohärenz behandelt werden und in der Regel nicht der Granularität von Datenobjekten der Programme entsprechen.

#### 2.1.2.3 Architektur-Klassen

Mit den angegebenen Kriterien können nun Parallelrechner-Architekturen fundiert unterschieden werden. Im folgenden werden die wichtigsten Klassen und ihre Eigenschaften vorgestellt:

**UMA** — Uniform Memory Access Multiprocessor (Abb. 2.2 a)
Konfigurationen dieser Klasse besitzen einen physisch gemeinsamen Speicher, auf den von allen Prozessoren mit gleicher Latenz zugegriffen werden kann. Der physische Speicher ist global eindeutig adressierbar, global erreichbar und nicht anhand der Prozessoren partitioniert. Kommerzielle Vertreter dieser Klasse sind die Vektorrechner Cray T90 [Cra95] und S90 sowie die Sequent Symmetry S81. Das Problem bei dieser Architektur ist, daß das Verbindungssystem – meist Bus – bei einer großen Zahl von Prozessoren zu einem Engpaß wird.

**cc-UMA** — Cache Coherent UMA (Abb. 2.2 b)
Rechner dieser Klasse besitzen zusätzlich kohärente Caches pro Prozessor, durch den die Anzahl der Speicherzugriffe auf den gemeinsamen Speicher verringert werden. Vertreter dieser Klasse sind die SUN E10000 [Cha97] und PCs nach der Intel Multiprozessor Spezifikation [Int97]. Die limitierende Wirkung des gemeinsamen Verbindungssystems zum Speicher ist gegenüber UMA stark reduziert, wenn auch nicht eliminiert. Zusätzlich steigt der Aufwand der für die Kohärenz zu erbringen ist, mit der Anzahl an Prozessoren.

**NUMA** — Non-Uniform Memory Access Multiprocessor (Abb. 2.2 c)
NUMA-Rechner verfügen nach wie vor über einen global eindeutigen und erreichbaren Speicher, der jedoch in Partitionen unterteilt ist, die statisch an Prozessoren gebunden sind. Während auf lokale Speichermodule schnell zugegriffen werden kann, erfolgt der Zugriff auf entfernte Speichermodule mit deutlich höherer Latenzzeit. Verfügen die Prozessoren über schnelle Caches, so werden lokal angelegte Replikate entfernter Daten nicht automatisch kohärent gehalten. Ein Beispiel für diesen Typ Parallelrechner ist die IBM RP3 [PBG+85]. Zu beachten ist, daß die statische Partitionierung und der große Unterschied der Latenzzeit zwischen lokalem und entferntem Speicher bei einer NUMA-Architektur eine sorgfältige Aufteilung der Speicherobjekte auf die Partitionen erfordert.

**cc-NUMA** — Cache Coherent NUMA (Abb. 2.2 d)
Wegen der fehlenden Kohärenz bei NUMA-Rechnern wurden cc-NUMA-Architekturen entwickelt, die kohärente lokale Caches besitzen, um Speicherzugriffe wirksam zu beschleunigen. Speicheradressen sind jedoch weiterhin fest an Prozessoren gebunden, wodurch Daten nicht zwischen den lokalen Speichern der Prozessoren migrieren können, sondern lediglich in dem Cache des zugreifenden Prozessors repliziert werden. Eine Maschine dieser Klasse ist die DASH [LLG+89].

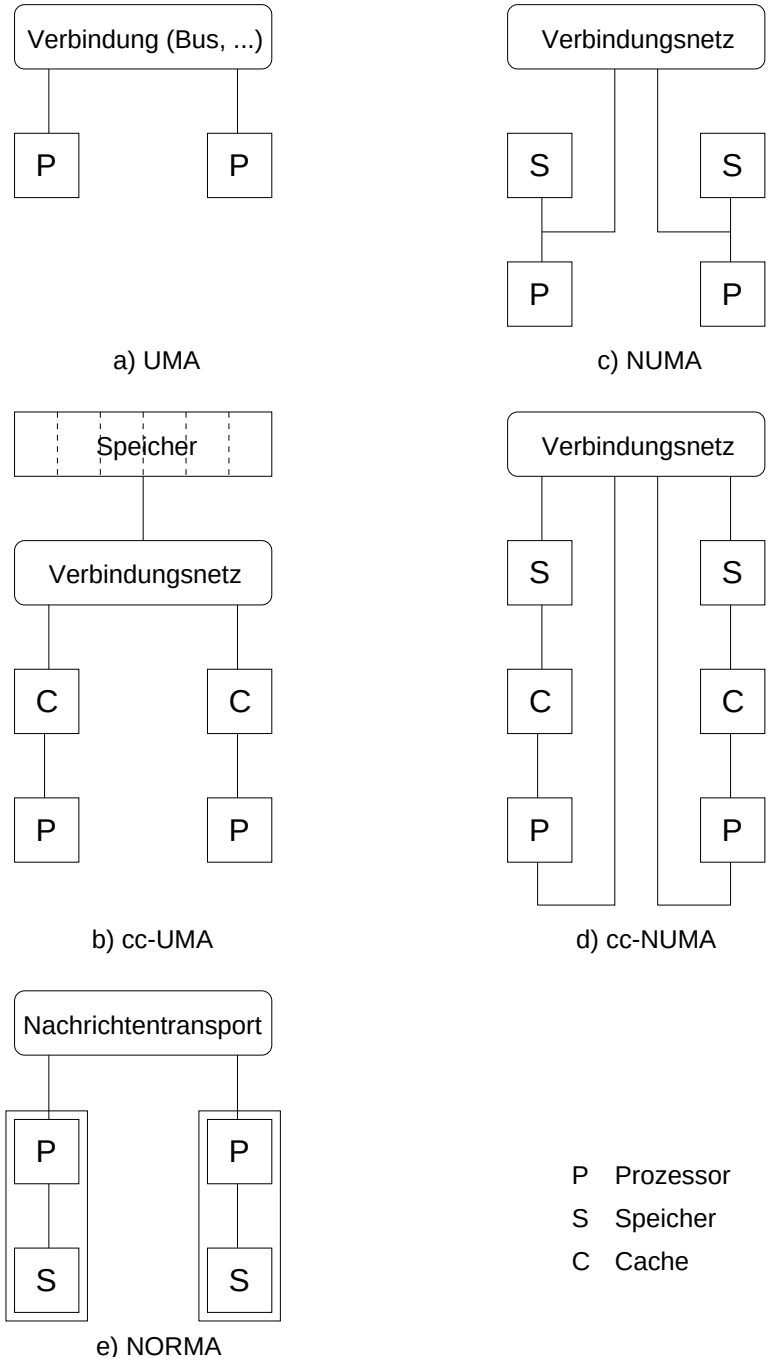


Abbildung 2.2: Architekturmodelle

**COMA** — Cache-Only Memory Access Multiprocessor
Die Leistung von cc-NUMA Architekturen hängt stark von der Größe der Caches ab. Bei einer COMA Architektur handelt es sich um eine cc-NUMA-artige Architektur, bei der die lokalen Speicher durch sehr große und teure Caches ersetzt sind. Der gesamte Speicher ergibt sich aus der Summe der Caches. Die Partitionierung des Speichers wird durch die Caches dynamisiert. Im Unterschied zu NUMA werden Zugriffe auf entfernte Speicher nicht über das Verbindungsnetz geleitet, sondern durch den Transfer von kohärenten Speicherblöcken in den lokalen Cache befriedigt. Die initiale Verteilung der Daten auf die Prozessoren ist daher weniger kritisch, da Speicherobjekte an ihren Nutzungsort migrieren und repliziert werden. Der Erfolg dieser Flexibilisierung hängt davon ab, inwiefern die Einheiten der Kohärenz mit den Granularität der benutzten Daten korrespondiert. Ein Vertreter dieser Klasse ist die KSR-1 von Kendall Square Research [BFKR92].

Alle bis hierhin beschriebenen Architekturen verfügen über einen gemeinsamen Speicher und werden dementsprechend als Multiprozessorsysteme eingeordnet. Parallelrechner, deren Speicher verteilt organisiert ist, werden Multicomputer genannt. Unter einem Multicomputer ist eine Menge weitestgehend autonomer Rechner, die auch *Stellen* oder *Knoten* genannt werden, zu verstehen, die über ein separates Nachrichtentransportsystem miteinander verbunden sind. Jeder einzelne Knoten besitzt mindestens einen Prozessor sowie einen eigenen Hauptspeicher. Zur dieser Familie von Rechnerarchitekturen gehören auch die vernetzten Arbeitsplatzrechner, die in dieser Arbeit zur Realisierung von V-PK-Systemen eingesetzt werden:

**NORMA** — No Remote Memory Access Multiprocessor (Abb. 2.2 e)
Bei einer NORMA-Architektur ist weder der Adreßraum global eindeutig noch der Speicher von den Prozessoren global erreichbar. Zugriffe auf entfernte Speichermodule sind nur mittelbar möglich, indem Nachrichten über das Verbindungsnetzwerk an andere Prozessoren gesendet werden, die ihrerseits ggf. das gewünschte Datum in einer Antwortnachricht liefern. Betrachtet man den gesamten Speicher der Konfiguration, so ist er statisch partitioniert. Etwaige Caches der Prozessoren werden wie bei NUMA von der Hardware nicht kohärent gehalten. Erhältliche NORMA-Architekturen mit schnellen Transportsystemen sind die Intel Paragon XP/S [BBD+94], CM-5 [TMC91] von Thinking Machines Corporation und IBM SP2 [AMM+95].
Der Vorteil des NORMA-Modells ist ohne Zweifel die Möglichkeit, extrem große Konfigurationen konstruieren zu können, die durch Verlagerung der Probleme auf die Nutzer der Konfiguration erreicht wird. Programme für NORMA-Architekturen müssen selbst Daten partitionieren, Caching in lokalen Speichermodulen implementieren, Kohärenz der Softwarecaches gewährleisten, um das gewünschte Konsistenzmodell durchzusetzen, Identifikatoren transformieren und entfernte Zugriffe durch den Austausch von Nachrichten – *message passing* – realisieren. Das Programmiermodell einer NORMA-Architektur ist daher äußerst kompliziert.

**NOWs/COWs** — Network/Clusters of Workstations
Arbeitsplatzrechner, die durch ein Verbindungsnetz miteinander verbunden sind, können ebenfalls als Parallelrechner vom Typ NORMA eingeordnet werden. Die Autonomie der einzelnen Stellen ist in diesem Falle jedoch wesentlich weitreichender. Die Knoten verfügen zusätzlich zu Speicher und Prozessoren über eigene Peripheriegeräte, auf jeder Stelle wird ein separates Knotenbetriebssystem ausgeführt und die Gestalt

der Gesamtkonfiguration unterliegt deutlich höherer Dynamik. Hinzu kommt, daß das Transportsystem in der Regel von niedrigerer Qualität als bei den bisher genannten Architekturen ist. Das Ungleichgewicht der Leistungsfähigkeit (Latenz und Bandbreite) von lokalen zu entfernten Speicherzugriffen erhält dadurch eine neue Größenordnung. Die resultierende hohe Variabilität der Interaktionskosten aus Sicht der Nutzer von NOW-Architekturen prägt das Programmiermodell und wurde deshalb in 2.1 zur Charakterisierung von verteilten Systemen verwendet.
Die beiden wichtigsten Vorteile dieser Architektur gegenüber allen anderen sind zum einen das sehr günstige Preis-/Leistungsverhältnis und zum anderen ihre nahezu ubiquitäre Verfügbarkeit in Form vernetzter Personal-Computer und Arbeitsplatzrechner. Trotz aller Nachteile, die die Defizite von NUMA und NORMA vereinen und neue schwerwiegende hinzufügen, ist dieser Architekturtyp auf Grund seiner Verbreitung für verteiltes Rechnen von größter praktischer Relevanz.

#### 2.1.2.4 Distributed Shared Memory

Insgesamt betrachtet ergibt sich folgendes Dilemma zwischen der Einfachheit der Nutzung und der Leistungsfähigkeit von Parallelrechnerarchitekturen. Während das Programmiermodell von Multiprozessoren mit gemeinsamen Speicher sehr einfach erscheint, ist deren technisch machbare und ökonomisch vertretbare Vergrößerung stark limitiert. Multicomputer unterliegen dieser Beschränkung nicht und können zu wesentlich leistungsfähigeren Konfigurationen ausgebaut werden. Nachteilig ist jedoch deren komplizierte Nutzung, die u. a. den expliziten Austausch von Nachrichten erfordert. Eine Kombination der Vorteile aus beiden Ansätzen, das einfache Programmiermodell des gemeinsamen Speichers und das Leistungspotential einer verteilten Organisation erscheint wünschenswert und wird in verschiedenen Arbeiten mit dem Einsatz spezieller Hardware oder Software angestrebt. Erweiterungen dieser Art für NORMA- und NOW-Architekturen etablieren einen gemeinsam benutzbaren Speicher mit Hilfe der physisch verteilten Speicher, Prozessoren und dem Nachrichtensystem. Abbildung 2.3 illustriert

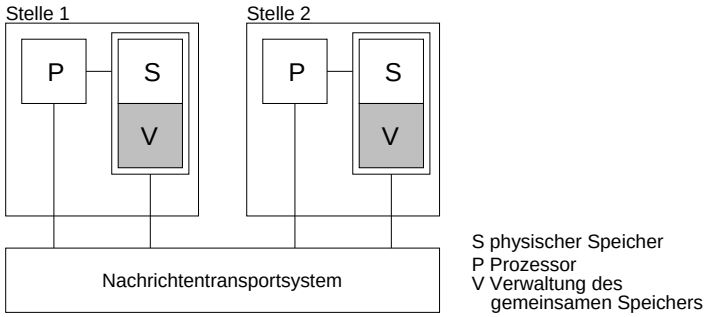


Abbildung 2.3: Distributed Shared Memory

den grundsätzlichen Aufbau eines DSM-Systems (Distributed Shared Memory). Neu in dieser Architektur, relativ zu NORMA/NOW, sind Verwaltungskomponenten `V`, die unter Nutzung des Nachrichtentransportsystems für die Prozessoren einen virtuellen gemeinsamen Speicher aufspannen, Konsistenz durchsetzen und für Kohärenz der Caches sorgen. Der Nutzer dieser Architektur wird von der Notwendigkeit des expliziten Nachrichtenaustausches und der Transformation von Identifikatoren zwischen Namensräumen entbunden.

**Hardware**
Ein Vertreter des Hardware-Ansatzes ist das SCI [Gus92] (Scalable Coherent Interface). Ähnlich wie bei einer COMA-Architektur fungieren die Hauptspeicher der einzelnen Stellen als kohärente Caches. Der Austausch von kohärenten Speicherblöcken erfolgt mittels eines dediziert optimierten Protokolls, das geringere Latenzzeiten und höhere Bandbreiten als gewöhnliche Transportsysteme zwischen Arbeitsplatzrechnern erzielt. Charakteristisch für SCI und ähnliche Hardware-Lösungen, die sich auf das Transportsystem konzentrieren, ist allerdings zum einen die Notwendigkeit spezieller Geräte, was die Motivation für die Nutzung von NOW als Parallelrechner in Frage stellt, und zum anderen der mangelhafte Bezug der Angebote zu den Anforderungen der Programme. Speicherseiten, die,wie im Falle von SCI [KLS99], als Einheiten der Kohärenz verwendet werden, sind in aller Regel nicht geeignet, um feingranularen Datenaustausch, wie er von den parallelen Ausführungsfäden einer Berechnung möglicherweise angefordert wird, zu realisieren. Das Phänomen, das durch die inadäquate Wahl von physischen Speichereinheiten als Kohärenzeinheiten entsteht, ist *false sharing*, dessen schwerwiegende Konsequenzen auf die Leistungsfähigkeit von der besseren Latenz und Bandbreite des optimierten Transportsystems nicht aufgewogen werden können. Der Nutzer eines solchen Systems ist demzufolge gezwungen, die Objekte angepaßt an die Transporteinheiten der Hardware zu partitionieren. Dies widerspricht dem ursprünglichen Ziel der Vereinfachung. Wirksam verhindert werden kann dieses Problem nur durch eine Veränderung der Speicherlogik der Hardware, die nicht Seiten sondern individuelle Objekte mit ihren Rechten verwalten muß. In dem Projekt MONADS wurde konsequent eine Architektur entwickelt und in Hardware realisiert, die in der Lage ist, Objekte mit unterschiedlicher Granularität in einem großen $\geq$ 64Bit gemeinsamen Adreßraum individuell zu verwalten [Abr82, AK85]. Aktuelle Arbeiten an verteilten Ein-Adreßraum-Systemen greifen die dort erzielten Ergebnisse wieder auf [AS96]. Nach wie vor scheitern diese Bemühungen jedoch an der geringen Verbreitung entsprechender Geräte, was auf das mangelnde Interesse der Industrie zurückzuführen ist.

**Software**
Sind die Verwalter `V` einer DSM–Architektur mit Programmen realisiert, die selbst auf den Prozessoren `P` ausgeführt werden, so spricht man von einem Software DSM-System. Wenngleich bei dieser Vorgehensweise ein enormer Aufwand seitens der Software zu leisten ist und die quantitative Leistung zunächst deutlich schwächer als bei Hardwarelösungen erscheint, ermöglicht die größere Flexibilität einer Softwarelösung dynamische Adaption an die Anforderungen, die sowohl in der Granularität der gemeinsam benutzten Datenobjekte als auch der Nutzungsfrequenz in der Praxis stark variieren. Die oben beschriebenen Probleme von Hardwarelösungen können auf diese Weise vermieden werden. Sowohl in der Forschung als auch für den kommerziellen Einsatz wurden Software-DSMs in zahlreichen Varianten entwickelt, die zusammen einen großen Teil des gesamten kombinatorischen Spektrums der weiter oben beschriebenen quantitativen und qualitativen Eigenschaften der Speicherorganisation abdecken. Die Unterschiede der hier nicht im Detail betrachteten Systeme erstrecken sich von der Partitionierung des virtuellen Adreßraumes über Fähigkeiten zur Migration und Replikation von Objekten bis zum Ausführungsmodell, das für die Spezifikation paralleler Teilberechnungen vorgesehen ist.

Ein zentrales Kriterium, mit dem DSM-Systeme im allgemeinen und besonders Software-DSMs einzuordnen sind, ist die Granularität der Speicherblöcke, die als Einheiten der Kohärenz dienen. Dieses Kriterium dividiert DSM-Systeme in *seitenbasierte* - und *objekt-*

*basierte*[4] Systeme. Seitenbasierte Systeme betrachten den gemeinsamen Speicher als eine Menge virtueller Seiten, die vielfache von physischen Speicherseiten sind und darüber hinaus keine weitere Struktur besitzen. Wegen ihrer Nähe zur gewöhnlichen virtuellen Speicherverwaltung werden seitenbasierte Systeme als SVM[5]-Systeme bezeichnet. Dieses sehr einfache Verfahren ist mit der seitenbasierten Verwaltung gängiger Hardware einfach zu realisieren, leidet grundsätzlich jedoch unter dem gleichen Problem des *false sharing*, wie oben für SCI beschrieben. Vertreter dieser Klasse von Systemen sind IVY [Li86], JIAJIA [E+99], *Mirage* [FP89, FHJ94], *Quarks* [Kha96] und *Treadmarks* [ACD+96].

Die Alternative dazu ist die DSOM[6]-Variante von DSM-Systemen. Anstelle einer Einteilung in Seiten, wird der gemeinsame Speicher als Container für eine strukturierte Menge von Datenobjekten betrachtet. Die Granularität der Datenobjekte ist variabel und ihre Anzahl wächst und schrumpft dynamisch. Individuelle Datenobjekte dienen als Transporteinheiten, an die darüber hinaus in einigen Systemen separat ein angepaßtes Konsistenzmodell und Kohärenzprotokoll gebunden werden kann. Dieser Ansatz nutzt die größere Flexibilität einer Software-Lösung gegenüber Hardware und eliminiert wirksam negative Effekte seitenbasierter Verfahren, wie etwa false sharing. Beispiele für objektbasierte verteilte gemeinsame Speicher sind *Linda* [Gel85], *Orca* [BK93], CRL [JKW95a] und *Midway* [BZS93]. DSOM-Systeme leiden allerdings in der Regel unter unbefriedigender Leistung, die aus dem Aufwand resultiert, der für die Verwaltung individueller Datenobjekte geringer Granularität ohne Unterstützung durch die Hardware zu leisten ist. Darüber hinaus wurde in [BK98] gezeigt, daß SVM-Systeme durch den Transport ganzer Seiten eine *prefetching*-Charakteristik aufweisen, die für Programme mit hoher Lokalität gegenüber DSOM-Systemen zu bevorzugen ist.

Am Beispiel der verschiedenen DSM-Varianten ist bereits auf diesem niedrigen Abstraktionsniveau deutlich zu sehen, daß die großen Differenzen der Latenz und Bandbreite zwischen lokalen und verteilten Zugriffen es nicht erlauben, ein uniformes Verfahren anzugeben, das die verschiedenen möglichen Nutzungsszenarien zufriedenstellend abdeckt. Statt dessen müssen wichtige Anforderungsklassen gefunden werden, für die flexibel einsetzbare Alternativen vorzubereiten sind [ACD+99]. Im Falle des verteilten Speichers gilt das in einem besonders weiten Rahmen; von der Granularität der zu verwaltenden Objekte über die Verwaltungsstrategien und -mechanismen bis hin zu den Konsistenzmodellen.

### 2.1.3 Konsequenzen für das V-PK-Management

Wegen ihrer weiten Verbreitung und ihres günstigen Kosten-/Leistungsverhältnisses sind NOW-Architekturen ohne spezielle Hardware-Erweiterungen von größter praktischer Relevanz und werden deshalb als Zielarchitektur für V-PK-Systeme gewählt. Die Angebote einer solchen Konfiguration sind jedoch weit entfernt von den qualitativen und quantitativen Erwartungen, die auf höchster Abstraktionsebene mit der Konstruktion hochwertiger paralleler und kooperativer Problemlösungen assoziiert sind.

Seitens der Hardware existiert weder ein gemeinsamer Namensraum noch ist der Speicher global erreichbar, wodurch das Programmiermodell auf dieser Ebene von der Heterogenität gemischter Speicherzugriffe und aufwendiger Nachrichtentransporte negativ bestimmt ist. Weiter erfolgen keine Maßnahmen um Kohärenz der Stellen-Caches zu gewährleisten und ein bestimmtes Konsistenzmodell durchzusetzen. Die Kommunikationsleistung zwischen den

[4] in der englischsprachigen Literatur auch „variablenbasiert“
[5] Shared Virtual Memory
[6] Distributed Shared Object Memory

Stellen ist um Größenordnungen geringer als lokal (Tab. 2.1), wodurch das Management bei der Realisierung kooperativer Berechnungen Differenzierungen vornehmen muß, um entfernte Kommunikation gesondert behandeln und ggf. minimieren zu können. Hinzu kommt, daß die einzelnen Stellen der NOW-Konfiguration selbst komplexe Systeme mit eigenen Speicherhierarchien, etc. sind, woraus ein enormes Spektrum an Realisierungsalternativen resultiert, für deren adäquate Nutzung eine Vielzahl von Entscheidungen zu treffen ist. Die Entscheidungsfindung ist allerdings äußerst schwierig, da die hierfür benötigte Information über das System verteilt ist und lokal nur unvollständige bzw. unpräzise Information zur Verfügung steht. Sollen beispielsweise Seiten aus dem Arbeitsspeicher einer Stelle verdrängt werden, so kann, je nach aktueller Auslastung des gesamten Systems, eine Verlagerung auf den lokalen Hintergrundspeicher (Plattenspeicher) oder über das Netz in den Arbeitsspeicher einer entfernten Stelle günstiger sein.

| CPU | **Latenz** (nanosec.) | | | | **Bandbreite** (Megabyte/s) | | | |
|---|---|---|---|---|---|---|---|---|
| | L1 cache | Mem | TCP local | TCP remote | Mem write | Disk write | TCP local | TCP remote |
| SUN V9 - 167 | 12 | 273 | $184*10^3$ | $2.5*10^6$ | 150 | 5.08 | 44 | 9 |
| Pentium II - 400 | 7 | 152 | $129*10^3$ | — | 149 | — | 55 | — |

Tabelle 2.1: Leistungsspektrum typischer Arbeitsplatzrechner

Dem zu konstruierenden V-PK-Management kommt die komplexe Aufgabe zu, diese große Lücke, die zwischen den Anforderungen des abstrakten Programmiermodells von MoDiS (siehe S. 71 ff.) und den Angeboten der NOW-Konfiguration klafft, zu schließen. Die physische Verteilung soll auf Ebene der Programmierung von V-PK-Systemen opak sein. Nutzer und Programmierer sollen nicht mit den Details und Konsequenzen der physischen Verteilung belastet werden, sondern sich einzig und allein auf die abstrakten Eigenschaften der parallelen Berechnung konzentrieren können. Das bedeutet, daß das V-PK-Management die Prozessoren des Systems einschließlich der Balancierung der Last selbständig verwalten muß. Ähnlich wie bei SVM-/DSOM-Systemen muß das V-PK-Management einen global eindeutigen Namensraum zur Verfügung stellen und globale Erreichbarkeit von Datenobjekten gewährleisten, wobei das gewünschte Konsistenzmodell effizient mit kohärentem Caching durchzusetzen ist. Weiter müssen Leistungsunterschiede automatisch nivelliert werden, um der Forderung nach der Opazität tatsächlich gerecht werden zu können. Neben diesen Forderungen, die dem Zweck dienen, eine NOW-Konfiguration für den Nutzer ähnlich einfach handhabbar zu machen, wie einen flexibel rekonfigurierbaren cc-UMA-Multiprozessor, wird selbstverständlich zusätzlich erwartet, daß sich aus der konzertierten Nutzung vernetzter Rechner de facto Vorteile, wie z. B. verkürzte Bedienzeiten, ergeben.

Diese Ziele können nur dann erreicht werden, wenn das Management in der Lage ist, die unterschiedlichen Fähigkeiten der Hardware-Komponenten angepaßt an die Anforderungen und den aktuellen Systemzustand flexibel einzusetzen. Variiert z. B. die Lokalität des Zugriffs über die Zeit, so muß das Management in der Lage sein, die Strategie von Datentransporten dynamisch von Vorgriff (*prefetching*) auf möglichst spät (*lazy*) zu verändern, um teure Nachrichtentransporte über das Verbindungsnetz zu minimieren [BK98]. Veränderungen der Anforderungen dieser Art kann durch einfache Fallunterscheidungen, nicht im erforderlichen Umfang begegnet werden. Während das prefetching von Datenobjekten gut mit der seitenba-

sierten Unterstützung der Standard-Hardware realisiert werden kann, ist das lazy-Verhalten für feingranulare Datenobjekte besser in Software inklusive dem Übersetzer [DLC+99] zu realisieren. **Flexibilität** muß gemeinhin durch Einbußen erkauft werden, die mit Verzweigungen zu Alternativen sowie der mangelhaften Angepaßtheit universeller Verfahren an konkrete Anforderungen unweigerlich einhergehen. Die Balance von Kosten und Nutzen definiert eine Grenze für die praktisch sinnvolle Flexibilisierung. Um den großen Spielraum zu erhalten, den das V-PK-Management benötigt, muß diese Grenze so weit als möglich nach oben verlagert werden. Möglich ist das durch die enge **Integration** der einzelnen Komponenten des Managements zu einem gesamtheitlichen Managementsystem. Unter Integration ist zu verstehen, daß

a) die Maßnahmen der einzelnen Komponenten eng aufeinander abgestimmt sind, um Reibungsverluste zu minimieren, und

b) möglichst präzise Information über Anforderungen und Angebote von den Managementkomponenten erarbeitet und zwischen ihnen bei Bedarf ausgetauscht werden kann.

Von dieser Überlegung ist die weitere Konstruktion einer flexiblen und integrierten Ressourcenverwaltung für V-PK-Systeme geleitet.

## 2.2 Nutzungsklassen

Das vorrangige Ziel paralleler Rechnerarchitekturen ist, Rechenanlagen mit hoher Leistungsfähigkeit zu entwickeln, um Probleme mit hohem Bedarf an Speicherkapazität und Rechenleistung lösen zu können. Anforderungen dieser Art finden sich im technisch-wissenschaftlichen Bereich ebenso wie in spezifischen Domänen, z. B. der Verarbeitung multimedialer Daten. Verteiltes Rechnen ist häufig ähnlich motiviert. Darüber hinaus gibt es jedoch zahlreiche andere und neue Beweggründe für die Nutzung physisch verteilter Ressourcen. Die teilweise konträren Facetten von verteiltem Rechnen machen eine Präzisierung der beabsichtigten Nutzung unerläßlich.

Mit dem im Projekt MoDiS verfolgten Allzweck-Ansatz wird beabsichtigt, verteiltes Rechnen, charakterisiert durch den Begriff V-PK-Systeme, auf möglichst universelle Art zu unterstützen. Wenngleich sich die vorliegende Arbeit in ihrem weiteren Verlauf auf die Aspekte der quantitativen Leistung, d. h. der *Effizienz*, konzentriert, ist das Verständnis der insgesamt allgemeinen Ausrichtung für die Einordnung der gesamten Vorgehensweise wichtig. Bei einer isolierten Betrachtung einer spezifischen Domäne für verteiltes Rechnen würden sich andere Anforderungen und andere Ergebnisse ergeben. Im folgenden werden zunächst ausgewählte Anwendungsbereiche für verteiltes Rechnen, die nicht von der Rechenleistung verteilter Systeme motiviert sind, aufgezeigt und mit der V-PK-Zielsetzung in Zusammenhang gebracht. Anschließend erfolgt im Abschnitt 2.2.2 eine analytische Betrachtung des quantitativen Leistungsvermögens verteilter Rechensysteme. Mit den Ergebnissen dieser stark vereinfachten Betrachtung sind wichtige Anforderungen an die Architektur des zu entwickelnden effizienten V-PK-Managements formuliert.

### 2.2.1 Qualitative Beweggründe

Verteilte Systeme erlauben es, physische und virtuelle Ressourcen überall und in nahezu beliebiger Menge zu nutzen. Besonders seltene bzw. teure Komponenten können, unabhängig von

den räumlichen Begebenheiten, auch über große Entfernungen von verschiedenen Punkten aus genutzt werden, wodurch Kosten sowohl für Geräte als auch für die abstrakten Ressourcen – Programme, Nutzer, etc. – durch eine verbesserte Auslastung eingespart werden können. Die **Ubiquität** der Ressourcen ist darüber hinaus eine wichtige Voraussetzung, um die **Mobilität** der Komponenten des Systems einschließlich ihrer Nutzer zu erhöhen, aus der sich ein erhebliches Potential für die Dynamisierung sowohl der physischen Konfiguration als auch der Arbeitsabläufe ergibt. Daraus resultierende Nutzungsvorteile können im Bereich *electronic commerce* anhand weltweit nutzbarer Informationsdienste und der Informationsgewinnung aus dem Internet mittels *http* [FGM+98] studiert werden. Mobilität setzt ortsunabhängige Identifikation der Komponenten voraus. Im Falle des *http* muß sich die Mobilität auf den Transport von Dokument-Replikaten beschränken [DJD99], da die Dokumente selbst bereits ortsgebundene Namen besitzen. Dauerhafte konsistente[7] Migration von Dokumenten ist nur unter großen Anstrengungen zu bewerkstelligen. Um das Potential zur Flexibilisierung tatsächlich ausschöpfen zu können, müssen Details der physischen Verteilung so weit wie möglich eliminiert und auf niedrige Abstraktionsebenen beschränkt werden. MoDiS priorisiert Ubiquität und Mobilität gegenüber der größeren Effizienz, die dann erzielt werden kann, wenn Festlegungen betreffend der physischen Verteilung als Invarianten oder nur geringfügig Veränderliche getroffen werden. Von einer geeigneten Programmiersprache und vordefinierten Komponenten, die zur Formulierung von Problemlösungen bereitgestellt werden, wird deshalb postuliert, daß die physische Verteilung verborgen bleibt. Problemlösungen, die unter dieser Maxime konstruiert werden, profitieren zusätzlich durch erhöhte **Portabilität** auf Konfigurationen, die in ihrer Größe und Gestalt variieren können und **skalieren** besser mit der Hinzunahme zusätzlicher Ressourcen.

Eine weitere wichtige Möglichkeit der Nutzung verteilter Systeme ist die Unterstützung neuer **sozialer Organisationsformen**. Den großen Stellenwert dieser Fähigkeit zeigen u. a. die Forschungsgebiete Workflow-Management, rechnergestützte Gruppenarbeit (CSCW[8]) [EGR91, BS98] und Telearbeit [SRKM96]. Die physische Verteilung wird in Projekten wie dem Mehrbenutzer-Editor IRIS [Koc96] genutzt, um der Tatsache Rechnung zu tragen, daß Dokumente in zunehmendem Maße von mehr als einem Autor erstellt und bearbeitet werden, wobei die beteiligten Personen räumlich weit voneinander entfernt sein können. Die technischen Probleme, die sich bei der Realisierung von Vorstellungen dieser Art ergeben, konzentrieren sich auf andere Fragestellungen als bei Versuchen, maximale Effizienz zu erzielen. Aufgrund der Trägheit, die mit Benutzer-Interaktion einhergeht und der Tatsache, daß bereits ein sehr schwaches Konsistenzmodell mit entsprechend loser Kohärenz der Dokumentreplikate für diesen Zweck akzeptabel ist, rücken Fragen der effizienten Verwaltung der Prozessoren und des Speichers in den Hintergrund. Im Vordergrund stehen demgegenüber Fragen, wie z. B. die adäquate Verbreitung von Information über den Zustand des gemeinsamen Dokuments und Aktivitäten der Coautoren für spezifische Dokumenttypen. Bei der Implementierung dieser Konzepte muß bereits zur Überwindung der Stellengrenzen wiederholt großer Aufwand getrieben werden,weil existierende Plattformen mit primitiven Konzepten, wie Sockets [Pac92] und RPC [Blo92], keine adäquate Unterstützung bieten. Dieser vielfach und redundant zu leistende Aufwand soll in V-PK-Systemen durch ein geeigneteres Programmiermodell sowie der automatisierten Überwindung der Stellengrenzen durch das Management eliminiert werden.

[7] bezogen auf existierende Referenzen auf das Dokument
[8] Computer Supported Cooperative Works

Ähnliches gilt für die Unterstützung **inhärent verteilter** Aufgaben, wie etwa komplexe Produktions- und Fertigungsanlagen. Beispiele hierfür sind dezentrale Steuerungssysteme in Fahrzeugen, Aufzugsteuerungen und Fertigungsstraßen auf Basis des CAN-Bus [Piz94]. Auch hier führt das niedrige Abstraktionsniveau existierender Sprachen und Betriebssysteme dazu, daß bereits erheblicher Aufwand nur für das Anheben des Niveaus auf das erforderliche Maß investiert werden muß. Zu berücksichtigen sind in diesem Umfeld zusätzliche neue Probleme, die über den Rahmen des automatisierten Allzweck-Managements in V-PK-Systemen hinausgehen.

Ein in der Praxis sehr wichtiger Einsatz verteilter Systeme ist die Erhöhung der **Zuverlässigkeit und Fehlertoleranz** in sicherheitskritischen Bereichen. Die Vielfachheit physischer und logischer Ressourcen wird in diesem Fall genutzt, um mit gezielter Redundanz „*single points of failure*“ zu eliminieren. Die Spanne reicht von redundanten Speicherverfahren (RAID[9]) über replizierte Diensterbringer bis hin zu vollständig replizierten Rechensystemen. Für die Konstruktion von V-PK-Systemen ist es wichtig festzuhalten, daß Zielsetzungen dieser Art Wünschen, wie Effizienz, widersprechen. Um Interessenskonflikte dieser Art lösen zu können, sind zwei Voraussetzungen notwendig:

1. Es muß eine integrative Basis existieren, auf deren Grundlage der Konflikt erkannt sowie eine Lösung gefunden und durchgesetzt werden kann.

2. Die widerstrebenden Forderungen müssen genügend Freiheitsgrade für eine Anpassung an einen Kompromiß besitzen.

Im Kontext von MoDiS wird Voraussetzung 1 durch die gesamtheitliche Systemkonstruktion geschaffen. Die zweite Forderung unterstreicht die Notwendigkeit des hohen Abstraktionsniveaus, das für die Spezifikation flexibel realisierbarer V-PK-Systeme im Projekt MoDiS angestrebt wird.

### 2.2.2 Quantitative Leistung

NOW-Konfigurationen werden wegen ihrem rechnerisch günstigen Preis-/Leistungsverhältnis und ihrer weiten Verbreitung als geeignete Ausgangsbasis für die Verkürzung von Ausführungszeiten und damit verbundenen Wartezeiten der Benutzer betrachtet. Mit den Überlegungen der folgenden Abschnitte werden die Anforderungen an ein in diesem Sinne quantitativ leistungsfähiges verteiltes Managementsystem präzisiert.

In zahlreichen Arbeiten wird demonstriert, daß die simultane Ausführung von Teilberechnungen auf verschiedenen Stellen vielversprechende Beschleunigungen der Gesamtberechnung ermöglicht. Als Vergleichsgrundlage der Messungen wird dabei in der Regel die Ausführung des selben Programms inklusive seiner auf physische Verteilung vorbereiteten Managementumgebung auf einem Rechner herangezogen. Entsprechend der Terminologie im Umfeld paralleler Programme wird mit dieser Methode die *relative Beschleunigung* bestimmt.

**Definition 2.4 (Relative Beschleunigung)**
*Sei $t_1$ die Ausführungszeit eines parallelisierten Programms $P$ auf einem Prozessor und $t_n$ die Ausführungszeit für $P$ auf $n > 1$ Prozessoren.*

$$\textit{Relative Beschleunigung } S_r = t_1/t_n$$

[9] Redundancy Array of Inexpensive Disks

Mit Hilfe der relativen Beschleunigung wird in vielen Fällen suggeriert, daß die entworfenen Managementverfahren tauglich sind, um bei einer verteilten Hardwarekonfiguration eine Verkürzung von Ausführungszeiten herbei zu führen. In der Realität haften dieser Vorgehensweise Mängel an:

1. Aussagen dieser Art sind im Hinblick auf die ursprüngliche Zielsetzung, Wartezeiten durch die Nutzung verteilter Ressourcen zu reduzieren, wegen der gewählten Vergleichsgrundlage zumindest irreführend.

2. Beobachtungen des Verhaltens bei einer geringen Anzahl von Stellen sind für die Extrapolation auf eine größere Anzahl nicht ohne weiteres geeignet.

Um sinnvolle Aussagen über Leistungsgewinne durch paralleles bzw. verteiltes Rechnen machen zu können, muß die Dauer der Ausführung einer verteilten Problemlösung und einem verteilten Managementsystem mit der Ausführungszeit eines semantisch äquivalenten lokalen Algorithmus und effizienten lokalem Management verglichen werden. Bei der Untersuchung paralleler Programme wird diesem Sachverhalt mit dem Begriff der *absoluten Beschleunigung* Rechnung getragen, bei der als Vergleichsgrundlage die Ausführungszeit des schnellsten sequentiellen Algorithmus herangezogen wird.

**Definition 2.5 (Absolute Beschleunigung)**
*Sei $t_{seq}$ die Ausführungszeit des schnellsten sequentiellen Programmes $P'$ für das Problem $\mathcal{P}$ auf einem Prozessor und $t_n$ die Ausführungszeit eines parallelen Programmes $P$ zur Berechnung von $\mathcal{P}$ auf $n > 1$ Prozessoren.*

$$\textit{Absolute Beschleunigung } S_a = t_{seq}/t_n$$

$S_a$ macht den Einfluß einer großen Menge zusätzlicher Faktoren deutlich. Unter anderem wird die Leistung des parallelen Algorithmus mit der des sequentiellen Pendants verglichen, Unterschiede in dem Optimierungsverhalten der gewählten Übersetzer berücksichtigt und die Anzahl der zusätzlich zu überwindenden Schichten des Managementsystems mit einbezogen. Da in der Praxis allerdings oft ein schnellstes sequentielles Programm fehlt, wird häufig auf eine Betrachtung von $S_a$ verzichtet und lediglich die relative Beschleunigung untersucht.

**Definition 2.6 (Effizienz)**
*Effizienz* $E = S/n$; Je nach Beschleunigungsmaß $S = S_r$ oder $S = S_a$ wird zwischen der relativen Effizienz $E_r$ bzw. absoluten Effizienz $E_a$ unterschieden.

Wenngleich die exakte Bestimmung von $t_{seq}$ und damit $S_a$ schwierig bzw. unmöglich ist, so ist eine Annäherung der quantitativen Betrachtungen an die absolute Beschleunigung für die Beurteilung verteilter Managementverfahren sinnvoll. Vergleicht man etwa die verteilte Ausführung mit einer parallelen Ausführung auf einer Multiprozessormaschine, so wird der enorme organisatorische Aufwand für die Verteilung des parallelen Programmes sowie die Schwierigkeiten und Mängel der im verteilten Fall eingesetzten Verwaltungsverfahren deutlich.

Die in Tafel 2.2 dargestellten Werte sind zur Illustration dieser Problematik aus [LKBT92] entnommen. Sämtliche Messungen wurden auf MC68020 Prozessoren vorgenommen. Für die verteilte Ausführung mit dem Laufzeitsystem *Orca Multicast* wurde eine 10MBit/s Ethernet Verbindung gewählt, während für die Multiprozessorversuche eine VME-Bus Lösung mit insgesamt 8MB gemeinsamen Speicher eingesetzt wurde. In diesem Beispiel ist die Leistung der

verteilten Ausführung mit nur einer Stelle etwa um den Faktor 4.5 langsamer. Ein Effekt, der ähnlich auch in anderen Realisierungen verteilter Ausführungsumgebungen beobachtet werden kann. Obwohl die relative Beschleunigung $S_r$ im verteilten Fall nahezu perfekt ist ($0.986 \leq E_r \leq 0.953$), werden bereits fünf Rechner benötigt, um die Ausführungszeit zu erreichen, die mit der parallelen Variante auf einem Prozessor erzielt wird. Da darüber hinaus die Effizienz mit wachsender Zahl beteiligter Rechner abnimmt, sind den Leistungsgewinnen auch bei Einsatz großer Mengen an Ressourcen enge Grenzen gesetzt. Die negative Relation von Kosten und Nutzen widerspricht offenkundig dem Ziel, vernetzte Arbeitsplatzrechner als preiswerte Plattform für ressourcenintensive Berechnungen einzusetzen.

| **Matrixmultiplikation (Ausführungszeiten in Sekunden)** | | | | | |
|---|---|---|---|---|---|
| Anzahl der Prozessoren bzw. Stellen | 1 | 2 | 3 | 4 | 5 |
| Orca Multicast | 810.3 | 410.6 | 279.1 | 209.8 | 169.9 |
| Shared Mem. Multiproz. | 180.5 | 90.5 | 60.8 | 45.9 | 36.6 |

Tabelle 2.2: Performance einer Matrixmultiplikation im Projekt ORCA

In der Tat erscheint das Erzielen absoluter Leistungssteigerungen durch die Nutzung verteilter Ressourcen in der Praxis wesentlich schwieriger als vermutet. Um Vergleichbarkeit mit in der Literatur durchgeführten Messungen zu erzielen, wird zunächst von folgenden Annahmen ausgegangen:

**A1:** Die Problemgröße ist fest.

**A2:** Das Verhältnis zwischen sequentiellen $t_s$ und parallelen Programmteilen $t_p$ ist bekannt.

Unter diesen Voraussetzungen gibt das Gesetz von Amdahl [Amd67] eine obere Schranke für die relative Beschleunigung eines parallelen Programmes in Abhängigkeit von dem Anteil an sequentieller Arbeit an.

**Satz 2.7 (Gesetz von Amdahl)**
*Sei* $t_s + t_p = 1$

$$S_r = \frac{t_s + t_p}{t_s + t_p/n} = \frac{n}{1 + (n-1) * t_s}$$

Setzt man in das Amdahl-Gesetz verschiedene Werte für $t_s$ und $n$ ein, so erhält man, wie in Abbildung 2.4 dargestellt, eine Schar von Kurven, die mit wachsender Anzahl an Prozessoren steiler abfallen.

Bei dem Übergang von der parallelen auf eine verteilte Ausführung ist zusätzlicher organisatorischer Aufwand zu leisten. Die Einflußfaktoren der Verteilung auf die Leistung des Systems können in vier Klassen eingeteilt werden:

**$t_f$:** Fixkosten für die Vor- und Nachbereitung der verteilten Ausführung.
$t_f$ ist abhängig von der Anzahl der Rechner $n$ und beinhaltet Kosten wie etwa die Zeit für die Verteilung des Programmtextes auf die beteiligten Rechner.

**$t_a$:** Kosten für asynchrone rechnerübergreifende Maßnahmen.
Das Attribut *asynchron* bezieht sich darauf, daß die in $t_a$ enthaltenen Kommunikationskosten nicht in unmittelbarem Zusammenhang mit dem Fortschritt der Berechnung

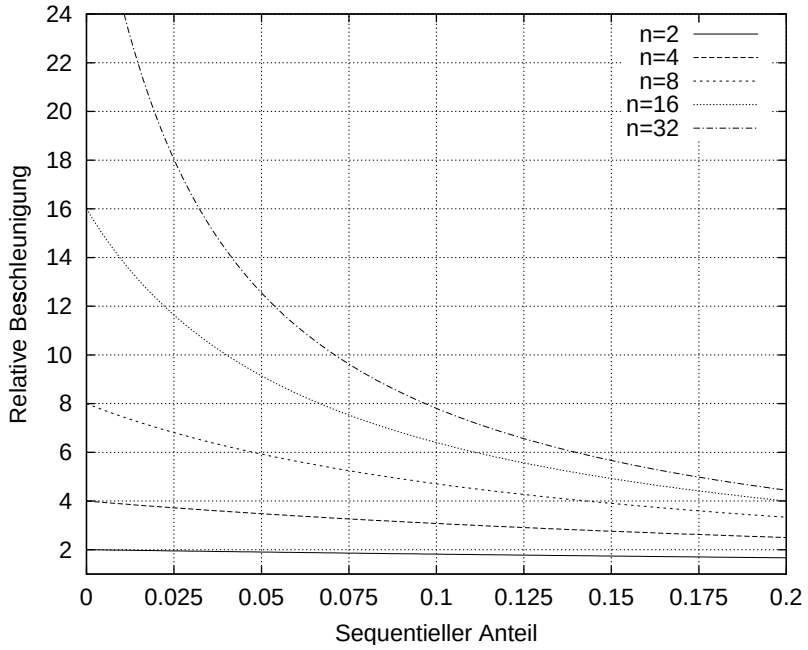


Abbildung 2.4: Amdahl-Gesetz

stehen, wie z. B. bei periodischem Austausch von Lastindizes. $t_a$ ist proportional zu $n$ und der Dauer der Berechnung.

**$t_y$:** Synchrone Kommunikationskosten bzw. Mehraufwand für die Überwindung von Stellengrenzen in unmittelbarer Folge von Kooperationsabhängigkeiten zwischen den parallelen Teilberechnungen. U. a. gehört hierzu der Mehraufwand für die ggf. verteilte Ausführung von Unterprogrammen oder die Beschaffung eines entfernt realisierten Datums.

**$t_l$:** Stellenlokaler Mehraufwand, der durch die Vorbereitung des Systems auf verteilte Ausführung mittels zusätzlichen Abstraktionen und Schichten des Laufzeitmanagements entsteht und bereits bei Ausführung auf $n = 1$ Rechner zum tragen kommt.
$t_l$ wird durch ein weites Spektrum an Veränderungen der Systemarchitektur von einem evtl. veränderten Programmiermodell über die Optimierungsstufen des Übersetzers bis hin zu veränderten Basismechanismen des Kerns geprägt.

Unter der Annahme, daß $t_f$ und $t_a$ durch geeignete Maßnahmen, wie Überlappung der Kommunikation mit Berechnungen oder *piggy-packing*, weitestgehend eliminiert werden können, wird an dieser Stelle lediglich der Einfluß von $t_y$ und $t_l$ weiter untersucht. Zur weiteren Vereinfachung sei $y$ entsprechend $t_y$ der Faktor der Verlangsamung des parallelen Anteils $t_p$ der Berechnung und $l$ der Faktor für den stellenlokalen Mehraufwand entsprechend $t_l$. $l$ sei während der Gesamtdauer der Berechnung zu leisten. Dann ist $t_n$, die Dauer der Berechnung auf $n$ Rechnern, durch

$$t_n = (t_s + \frac{t_p * y}{n}) * l$$

gegeben. Setzt man dieses Ergebnis als Dividend in das Amdahl-Gesetz ein, so erhält man eine erste Abschätzung für die obere Schranke der Beschleunigung, die mit einer verteilten

Ausführungsumgebung bei konstant gehaltener Problemgröße erzielt werden kann.

**Korollar 2.8 (Verteilte Beschleunigung 1)**
*Sei* $t_s + t_p = 1$

$$S_v(l, y) = \frac{t_s + t_p}{t_n} = \frac{n}{(t_s * (n - y) + y) * l}$$

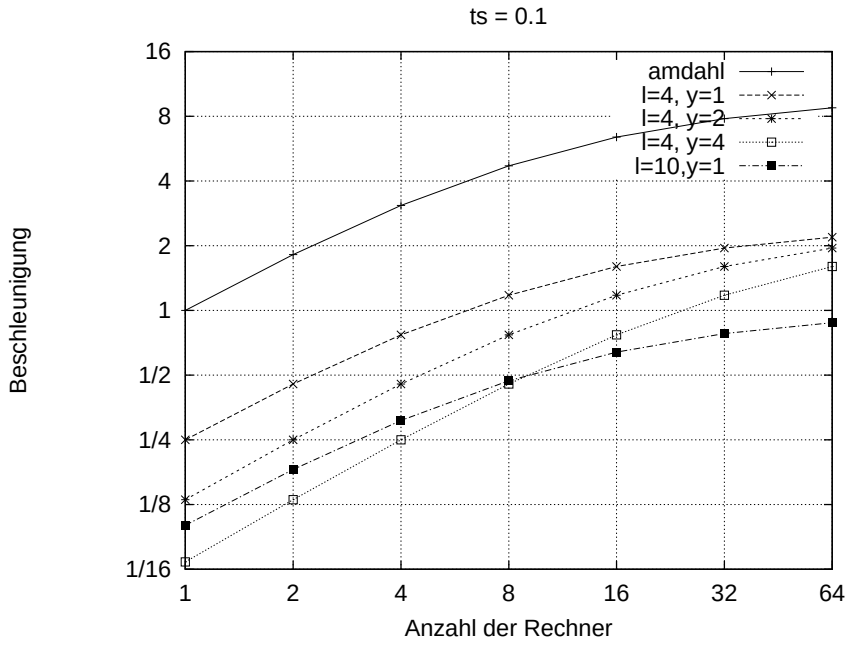


Abbildung 2.5: Leistungspotential verteilten Managements

Offensichtlich gilt für $l, y \geq 1 : S_a \leq S_v \leq S_r$. In Abbildung 2.5 sind Kurven für $t_s = 0.1$ und verschiedene Werte von $l$ und $y$ in einer doppelt logarithmischen Darstellung eingetragen. Besonders deutlich wird hierbei die dominante Bedeutung des Faktors $l$ und damit die Wichtigkeit der Leistungsfähigkeit des stellenlokalen Managements. Für den Fall $l = 4$, der in etwa mit dem Beispiel von *Orca Multicast* ($\approx 4.5$) übereinstimmt, werden bereits 32 Rechner benötigt, um eine Beschleunigung um den Faktor zwei zu erreichen. Darüber hinaus kann auch durch Hinzunahme weiterer Rechner keine bessere Beschleunigung als 2.5 erreicht werden. Des weiteren wachsen in diesem stark vereinfachten Modell die Steigung und Krümmung der Kurven mit den Kosten für das verteilte Management, ausgedrückt durch den Faktor $y$. Gleichzeitig sinkt jedoch mit wachsendem $y$ der Ursprung der Kurven bezüglich der Ordinate. Besondere Vorsicht ist deshalb bei der Interpretation von Meßergebnissen geboten, die bei konstant gehaltener Problemgröße über eine große Anzahl von Prozessoren hinweg perfekte Beschleunigungswerte mit einer Effizienz nahe eins bieten.

**Satz 2.9 (Gustafson-Barsis-Gesetz)**
*Sei* $t_s + t_p = 1$

$$S_r = \frac{t_s + n * t_p}{t_s + t_p} = 1 + (1 - n) * t_s$$

Bei Verzicht auf Annahme **A1** kann die maximale Beschleunigung für ein paralleles Programm mit dem Gustafson-Barsis-Gesetz [Gus88] abgeschätzt werden. Dieser Ansatz trägt der Tatsache Rechnung, daß durch paralleles Rechnen und dem damit verbundenen Zuwachs an Ressourcen Probleme größerer Dimension berechnet werden können. Im Gegensatz zu Amdahl fallen die Kurven bei G.-B. bei wachsendem $t_s$ linear ab. Korollar 2.10 gibt die verteilte Beschleunigung unter diesen Voraussetzungen und den Vereinfachungen wie oben an.

**Korollar 2.10 (Verteilte Beschleunigung 2)**
*Sei* $t_s + t_p = 1$

$$S_{v'}(l, y) = \frac{t_s + n * t_p}{(t_s + y * t_p) * l} = \frac{t_s * (1 - n) + n}{(t_s * (1 - y) + y) * l}$$

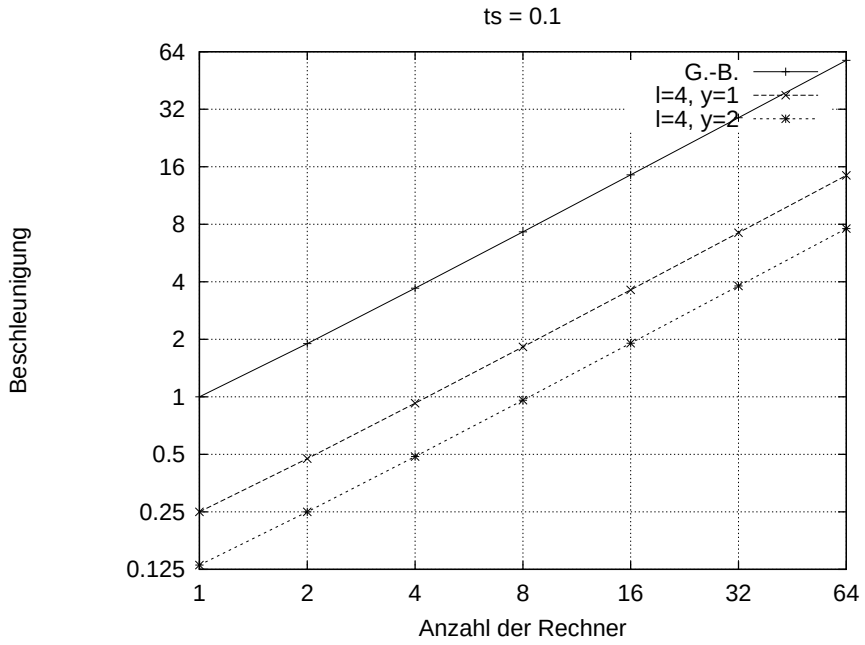


Abbildung 2.6: Gustafson-Barsis-Gesetz

Abbildung 2.6 macht deutlich, daß das Potential zu Verkürzungen von Ausführungszeiten unter diesen Bedingungen durch den Mehraufwand für stellenlokales Management nicht generell beschränkt wird. Allerdings ist auch in diesem Fall eine große Menge an Ressourcen einzusetzen, um tatsächliche Gewinne erzielen zu können. Für $l = 4$ müssen zur Halbierung der Ausführungszeit bereits mindestens acht Rechner eingesetzt werden.

Als Ergebnis dieser grundlegenden quantitativen Überlegungen ist festzuhalten, daß bei der Konstruktion und Realisierung des V-PK-Managements großes Augenmerk auf die Reduktion des Faktors $l$ gelegt werden muß. Anderenfalls kann entweder das ursprüngliche Ziel Ausführungszeiten zu verkürzen nicht bzw. nur mit unattraktiver Effizienz erreicht werden. Obwohl diese Forderung intuitiv erscheint, entspricht sie nicht der gängigen Vorgehensweise. Für gewöhnlich werden große Anstrengungen für die Entwicklung effizienter verteilter Verfahren aufgebracht, wobei deren Wechselwirkung mit dem stellenlokalen Management zunächst

kaum Berücksichtigung findet. Erst bei der experimentellen Ermittlung der Gesamtleistung werden dann Defizite der Gesamtarchitektur sichtbar. Zu diesem späten Zeitpunkt ist der Aufwand für die notwendigen, tiefgreifenden Korrekturen jedoch in aller Regel nicht mehr akzeptabel. Die Ursache hierfür liegt in der unerwartet großen Vielfalt an Einflußfaktoren auf $l$, die sich über die gesamte Architektur eines Rechensystems erstrecken.

**Programmiermodell:** Bereits eine geringfügige Modifikation des Programmiermodells für verteiltes Rechnen kann erhebliche Auswirkungen auf die Leistungsfähigkeit nach sich ziehen. Als Beispiel hierfür kann die *copy-in/-out* Semantik der Parameterübergabe in INSEL dienen. Die Vorzüge dieser Semantik in einer verteilten Umgebung gegenüber dem gängigeren *call-by-reference* sind offensichtlich. Eine naheliegende Implementierung mittels Kopiervorgängen im Speicher führt je nach Beschaffenheit der Parameter allerdings zu deutlichen Leistungsverlusten[10].

**Algorithmus:** Die Eigenschaften des Problemlösungsverfahrens selbst besitzen starken Einfluß auf $l$, so daß $l$ in Abhängigkeit des zu beobachtenden Problemlösungsverfahrens betrachtet werden muß. Im Beispiel der Parameterübergabe ist deren Einfluß auf $l$ von den Parametern selbst und der Häufigkeit von Unterprogrammaufrufen abhängig.

**Transformationsinstrumentarium:** Die Analyse- und Optimierungsfähigkeiten der eingesetzten Werkzeuge bestimmen das spätere Laufzeitverhalten maßgeblich. Als Bindeglied zwischen Sprache und laufzeitbegleitendem Management, werden ihre Fähigkeiten sowohl von den Eigenschaften der Sprachkonzepte als auch von der Integration in das Gesamtmanagement geprägt. Häufig sind bereits einfache Optimierungsschritte des Übersetzers für Leistungsgewinne/-verluste in der Größenordnung $l = 3$ [Muc97] verantwortlich.

**Laufzeitbegleitende Maßnahmen:** Die Integration universeller Verfahren, die den stellenlokalen und den entfernten Fall uniform abdecken, in das eng an den Berechnungsfortschritt gekoppelte Management, hat in aller Regel eine Verlangsamung des stellenlokalen Berechnungsfortschritts zur Konsequenz. Hierzu gehören bereits geringfügige Indirektionen wie im Falle von *Shadow* [GPR97], die lediglich der Erkennung verteilter Speicherzugriffe dienen, bis hin zur einheitlichen Realisierung von Methodenaufrufen über eine Vielzahl von Schichten wie im Beispiel von CORBA [YD96, OMG97].

**Basismechanismen:** Effekte der verteilten Verwaltung physischer Ressourcen seitens des Betriebssystemkerns wie z. B. die verteilte Zuteilung von Prozessoren [BASK95] bzw. das Aufspannen eines stellenübergreifenden Ein-Adreßraumes [HERV93, Win96b] sind zu berücksichtigen.

Insbesondere Top-Down orientierte und sprachbasierte Ansätze sind gefährdet, ihre ursprünglichen quantitativen Ziele im Verlauf der Konstruktion zu verfehlen, da die Festlegung neuer Konzepte auf hoher Abstraktionsebene, Auswirkungen auf niedrigere Ebenen und somit auch auf die Eigenschaften der Realisierung nach sich ziehen, die nur schwer kalkulierbar sind. Ein prominentes Beispiel für diese Problematik ist das gescheiterte Projekt *Spring* der Firma SUN Microsystems [HK93, RHKP95]. Im Gegenzug erklärt diese Betrachtung den Erfolg verhältnismäßig einfacher Erweiterungen bestehender Systeme. Diese Art der Unterstützung

[10] Bei Experimenten mit einem Programm für das Problem des Handelsreisenden bewirkte allein der Mehraufwand für die Übergabe von $13 \times 13$ Matrizen eine Reduktion der Gesamtleistung um den Faktor zwei.

für verteiltes Rechnen ist zwar in der Lage kurzfristig eine schnelle Verbesserung zu erzielen, stößt jedoch mittel- und langfristig bei wachsenden Anforderungen an die gleichen fundamentalen Grenzen, mit dem zusätzlichen negativen Effekt, daß durch die Aufschichtung und das Hinzufügen von Funktionalität schwer beherrschbare Abhängigkeiten entstehen, die weitere Entwicklungsschritte hemmen.

In der vorliegenden Arbeit werden Techniken und Verfahren demonstriert, die es erlauben, im Kontext eines Top-Down orientierten Ansatzes, Ressourcen effizient zu verwalten und die oben diskutierten Leistungsdefizite zu minimieren. Das von MoDiS verwendete Programmiermodell dient der Spezifikation paralleler und kooperativer Algorithmen mit einem homogenen und einfach zu nutzenden Konzepte-Repertoire, das auf dieser Ebene nicht im Hinblick auf effiziente Ausführung auf NOW-Konfigurationen optimiert ist. Um Leistungsverluste wie bei der oben beschriebenen Parameterübergabe zu vermeiden, muß das Management konsequent zwischen abstrakten - und realisierungsbedingten Eigenschaften differenzieren, um einen konstanten lokalen Mehraufwand $l$ zu vermeiden, d. h. die sorgfältige Separation von Abstraktionsniveaus ist von zentraler Bedeutung. Eine abstrakte *copy-in/copy-out* Semantik bedeutet nicht, daß tatsächlich Kopien im Speicher anzulegen sind! Um diese und ähnliche Schwierigkeiten zu überwinden, müssen Ressourcen mit geeigneten, anstelle gewohnter Eigenschaften definiert und von einem integrierten Instrumentarium durch angepaßte transformatorische - und laufzeitbegleitende Maßnahmen verwaltet werden. Wegen der Dynamik der Anforderungen auf der einen und der Heterogenität der Hardwareangebote auf der anderen Seite, bedeutet das, daß uniforme und dadurch häufig ineffiziente Verwaltungsmaßnahmen durch Spektren flexibel einsetzbarer Alternativen ersetzt werden müssen, um die jeweils besser geeigneten Ressourcen im erforderlichen Umfang nutzen zu können.

## 2.3 Ressourcenmanagement

Die oben erläuterten technischen Voraussetzungen bilden zusammen mit der angestrebten Nutzung verteilter Systeme die äußeren Klammern der Gesamtarchitektur eines verteilten Rechensystems. Das Innenleben dieser Klammer läßt sich grob in die Bereiche Algorithmen, Programmiersprachen und dem Management einteilen. Der Begriff des *Ressourcenmanagements* oder kurz *Management* wird in dieser Arbeit anstelle dem vorbelegten Begriff Betriebssystem eingeführt, um neue Sichtweisen möglichst frei von vorgeprägten Vorstellungen entwickeln zu können. Die Definition von Management basiert zunächst losgelöst von konkreten Aufgaben, wie Prozessor- und Speichermanagement, auf der Betrachtung von Ressourcen, Entscheidungen und Maßnahmen.

**Definition 2.11 (Managemententscheidung)**
*Eine Managemententscheidung ist eine Festlegung über die Erzeugung, Nutzung oder Auflösung von Ressourcen.*

**Definition 2.12 (Managementmaßnahme)**
*Eine Managementmaßnahme ist eine Aktion, die der Durchsetzung von Managemententscheidungen dient.*

**Definition 2.13 (Ressourcenmanagement)**
*Das Ressourcenmanagement eines Systems umfaßt sämtliche Entscheidungen und Maßnahmen, die der Realisierung einer Problemlösung, die durch ein abstraktes Programm formal spezifiziert ist, auf einer physischen Hardwarekonfiguration dienen.*

Der Begriff der Ressource wird in Kapitel 5 präzisiert. Für das Verständnis der folgenden Diskussion wichtiger Eigenschaften von Managementsystemen genügt es zu wissen, daß der Begriff Ressourcen sowohl physische Ressourcen wie Register, Arbeitsspeicher und Prozessoren als auch abstrakte bzw. virtuelle Ressourcen, wie Syntaxbäume, Binärcode, virtuelle Adressen, etc. einschließt.

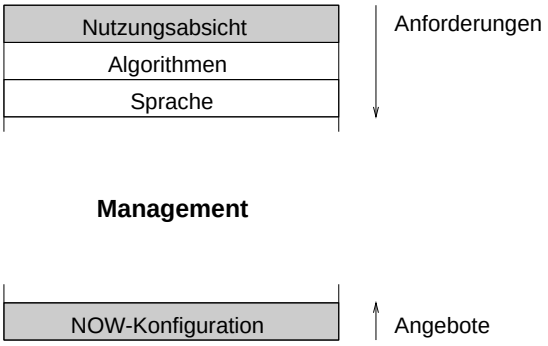


Abbildung 2.7: Position des Managements

Die Komponenten des Rechensystems, die Managemententscheidungen treffen oder entsprechenden Maßnahmen ausführen, bilden das *Managementsystem*[11] des Rechensystems. Aufgabe des Managementsystems ist, die Anforderungen, die aus der Nutzungsabsicht resultieren und durch ein Programm spezifiziert sind, auf Basis der physischen Angebote der Hardware zu erfüllen. Diese sehr allgemeine Betrachtung des Managements hilft Defizite zu vermeiden, die auf einer mangelnden Berücksichtigung von Zusammenhängen beruhen und durch eine a priori auf existierende Verfahren beschränkte Sichtweise entstehen können.

**Definition 2.14 (Betriebssystem nach DIN 44300)**
*Das Betriebssystem wird gebildet durch die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.*

Entgegen allgemein gehaltenen Definitionen von Betriebssystemen, siehe 2.14, entwickelt sich das Verständnis von Betriebssystemen zunehmend anhand von Implementierungstechniken und existierenden Verfahren. In der Literatur beschränkt sich die Darstellung immer mehr auf Betriebssystemkerne und die von ihnen verwalteten Ressourcen [Wec89, Tan92]. Der Grund für diese enge Betrachtung ist wieder der Aufwand, der für eine umfassende Betrachtung von Verwaltungssystemen beträchtlicher Komplexität erforderlich wäre. Durch das mangelnde Verständnis der Gesamtaufgabe entstehen ungenügend integrierte Lösungen an deren Übergänge qualitative und quantitative Verluste entstehen. Diesem Trend wird hier durch eine gesamtheitliche Betrachtung von Managementsystemen entgegengetreten.

[11] kurz: das Management

**Behauptung 2.15 (Betriebssystem)**
*Das Betriebssystem eines Rechensystems ist das Management des Rechensystems.*

Die folgenden Abschnitte beschäftigen sich mit grundlegenden Aspekten von Managementsystemen, die für das zentrale Thema dieser Arbeit, Konzepte und Verfahren für den systematischen Entwurf und der Realisierung eines V-PK-Managementsystems, von Bedeutung sind. Einzelne Verfahren und Mechanismen verteilter Betriebssysteme [Nut92, Tan95, Gos91, SPG91] sind in Bezug auf den methodischen Hintergrund der Aufgabenstellung weniger relevant und werden aus Platzgründen hier nicht im Detail erläutert.

### 2.3.1 Einordnung und Abgrenzung gegenüber der Sprache

Wenngleich parallele Algorithmen (z. B. [B$^+$92, GR88]) und Sprachen hier nicht im Detail diskutiert werden sollen, ist zu berücksichtigen, daß Abbildung 2.7 eine idealisierte Einteilung in die Bereiche Algorithmen, Sprache und Management darstellt. In der Praxis fällt die Abgrenzung des Managements in Richtung der beabsichtigten Nutzung wesentlich schwieriger, da das niedrige Abstraktionsniveau einiger Sprachen dazu führt, daß bereits bei der Formulierung paralleler Algorithmen Entscheidungen getroffen werden müssen, die nicht Teil der abstrakten Problemlösung sind. Das heißt, das Management beginnt bereits auf Ebene der Formulierung von Problemlösungen. Besitzen die Konzepte einer Programmiersprache realisierungsnahe Eigenschaften, so ist konsequenterweise gegebenenfalls bereits die Festlegung des Programmiermodells als Teil des Managements zu betrachten.

Management auf Ebene der Programmierung bedeutet auf der einen Seite einen Verlust an Praktikabilität, auf der anderen Seite jedoch auch möglicherweise größere Effizienz, sofern der Programmierer in der Lage ist, mit dem Wissen, das er besitzt, bessere Entscheidungen zu treffen als ein automatisiertes Management, das kein Detailwissen über die Problemlösung besitzt. Die Frage nach der Abgrenzung zwischen Sprache und Management ist somit eine Frage der Balance zwischen Praktikabilität und Leistung. Im folgenden werden wichtige Merkmale von Sprachen und ihre Auswirkungen auf die Praktikabilität und das Management betrachtet. Für eine umfassende Klassifikation verteilter und paralleler Sprachen sei auf [BST89] und [GG96] verwiesen.

#### 2.3.1.1 Sprachkonzepte und vordefinierte Komponenten

Die Verklammerung des Managements, wie sie in Abbildung 2.7 dargestellt ist, veranschaulicht, daß die Nutzungssprache alle Ziele des Managements festlegt und die Sprache der Hardware alle Angebote bzw. realen Möglichkeiten des Managements definiert. Die Nutzungssprache ist dementsprechend ausschlaggebend für die Fähigkeiten das gesamten Systems und muß deshalb über eine hohe Ausdruckskraft verfügen. Bei dem Entwurf dieser Sprache ist zwischen konzeptionellen Eigenschaften der Sprache und von außen hinzugenommenen und meist schwach integrierten Fähigkeiten zu trennen. Letztere müssen sich entweder den Einschränkungen der Nutzungssprache unterordnen oder führen zu Heterogenität und den damit verbundenen, unerwünschten Reibungsverlusten.

**Bemerkung 2.16 (Programm)**
*Ein Programm $W$ ist ein Wort einer formalen Sprache $Q$.*

In der Grammatik $G_Q = (N, T, P, S)$ der Sprache $Q$ existieren in aller Regel Produktionen, die sich auf das Nichtterminal *Identifikator* beziehen. Mit Hilfe dieser Produktionen und dem

Identifikator-Nichtterminal kann $Q$ um die Signaturen *vordefinierter Komponenten* $K_Q$ und ihrer Identifikatoren $I_{K_Q}$ zu $Q'$ mit $G_{Q'} = (N, T', P', S)$ expandiert werden. Die Menge $I_{K_Q}$ ergänzt $T$ zu $T' = T \cup I_{K_Q}$ mit entsprechender Erweiterung der Produktionen $P$ zu $P'$. In der Praxis wird diese Möglichkeit genutzt, um Sprachen mit zusätzlichen Fähigkeiten auszustatten, die zur Formulierung von Algorithmen benötigt werden oder diese erleichtern. In der Regel gilt $|T| \ll |T'|$, d. h. die Funktionalität der Konstrukte einer Sprache $Q'$ wird überwiegend von den vordefinierten Komponenten bestimmt. Beispiele sind die Sprache C [KR88] mit Bibliotheken u. a. für Ein-/Ausgabe, C++ [Str91] mit seinen zahlreichen Klassenbibliotheken sowie Java [Sun95a] und das dort eingesetzte Interface-Konzept.

Die Hinzunahme vordefinierter Komponenten ermöglicht es anscheinend, bereits existierende Sprachen in einem gewissen Rahmen nachträglich an neue Anforderungen anzupassen. Im Zusammenhang mit paralleler und verteilter Programmierung ist zu beobachten, wie versucht wird, das ursprünglich sequentielle Programmiermodell bestehender Sprachen mit Thread-Bibliotheken [IEE95] um Nebenläufigkeit zu erweitern oder verteilte Kommunikation mit Bibliotheken wie z. B. ACE [Sch94] zu unterstützen.

Um mit diesen Erweiterungen Nutzungsmöglichkeiten schaffen zu können, die mit $Q$ noch nicht gegeben sind, wird davon ausgegangen, daß $k \in K_Q$ kein Element aus $Q \cup Q'$ sein muß. Sowohl vordefinierte Komponenten für Ein-/Ausgabe als auch Multi-Threading, die als Ergänzung der Sprache C zur Verfügung gestellt werden, sind partiell in anderen Sprachen formuliert. Der Vorteil dieser primitiven Erweiterung ist der geringe Aufwand, der dafür zu leisten ist. Existierende Programme, Übersetzer und andere Werkzeuge, die einen unmittelbaren Bezug zur Sprache besitzen, müssen nicht angepaßt werden. Tatsächlich werden der Sprache auf diese Weise allerdings keine neuen Konzepte hinzugefügt, die ihre Ausdrucksfähigkeit erhöhen, sondern die schwache Semantik der Identifikatoren wird benutzt, um eine weitestgehend separate Ergänzungssprache aufzubauen, die Konkretisierungen der $Q$-Konzepte und dessen Management nutzbar macht.

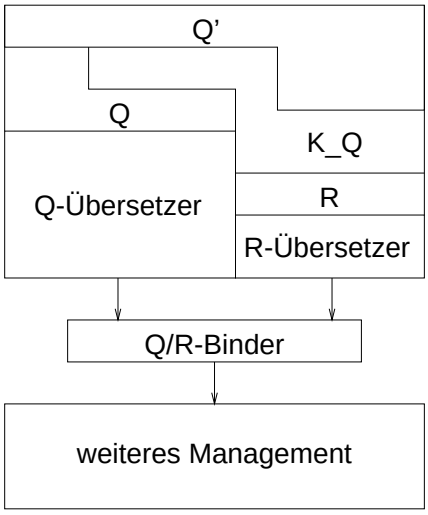


Abbildung 2.8: Heterogene Spracherweiterung mittels vordefinierter Komponenten

Die Nachteile dieser Vorgehensweise werden oft übersehen. Heterogene Erweiterungen dieser Art, wie in Abbildung 2.8 am Beispiel eines übersetzerbasierten Managements skizziert,

führen zu einem mangelhaft integrierten Management. Existieren $k \in K_Q$, die in einer Sprache $R$ verfaßt sind, für die weder $R \subseteq Q$ noch $R \supseteq Q$, so verläuft das Management für $Q$-Programme zunächst separat von dem für $R$-Programme. Die neue Nutzungssprache $Q'$ beinhaltet $k \in K_Q$ die bezogen auf $Q$ als Managemententscheidungen einzuordnen sind. Weder diese Entscheidungen noch das $R$-Management sind auf das $Q$-Management abgestimmt, denn das würde zumindest eine Veränderung des $Q$-Übersetzers erfordern. Doch genau solche Eingriffe werden bei dieser Vorgehensweise zur Reduktion des Aufwands abgelehnt. Das Resultat der mangelnden Integration ist Heterogenität, die sich angefangen von den Konzepten der $Q'$-Programmierung über das System erstreckt und in Form von Inkompatibilitäten bzw. Ineffizienzen zu Tage tritt. Im Zusammenhang mit Thread-Erweiterungen gibt es dafür zahlreiche Beispiele. Unter anderem werden die multiplen Keller der Ausführungsfäden nicht von der Speicherverwaltung unterstützt [Piz99a] oder Optimierungen des Übersetzers, z. B. Verschiebung schleifen-invarianter Berechnungen, führen zu inkonsistenten Ergebnissen.

Neben der mangelnden Integration in das Gesamtmanagement besteht bei dieser Vorgehensweise somit auch die Gefahr, daß die Black-Box Sicht auf die Konzepte von $Q$ aufgegeben werden muß, d. h. das Abstraktionsniveau von $Q$ fällt in Richtung der Realisierungseigenschaften ab.

Bei der Betrachtung von Programmiermodellen ist deshalb zu differenzieren, ob es sich bei den angebotenen Konzepten um Eigenschaften der betrachteten Sprache $Q$ oder um hinzugenommene Funktionalität handelt. Im letzteren Fall ist dann weiter zu unterscheiden, ob die Komponenten, um welche $Q$ erweitert wurde, selbst vollständig in $Q$ spezifiziert sind. Ist dies der Fall, so entstehen die oben genannten Probleme nicht. Der häufigere Fall ist jedoch die Erweiterung mit Mitteln, die nicht von $Q$ zur Verfügung gestellt werden.

#### 2.3.1.2 Explizit und implizit parallel

Anhand des Kriteriums, ob eine Sprache ein Konzept für die ausdrückliche Festlegung von Parallelität besitzt oder nicht, lassen sich explizit - und implizit parallele Sprachen abgrenzen. Diese beiden Klassen unterscheiden sich gravierend in der Fähigkeit, die korrekte und vollständige Spezifikation der Gesetzmäßigkeiten einer parallelen Problemlösung zu ermöglichen.

Zur Klasse der implizit parallelen Sprachen gehören überwiegend funktionale -, Datenfluß- und logische Sprachen. Ebenso sind imperative Sprachen hinzuzurechnen, die kein Aktivitätsträgerkonzept besitzen. Die Vorteile einer implizit parallelen Sprache sind das einfache Programmiermodell und u. U. die Möglichkeit, im Quelltext existierende sequentielle Programme mit geringem Aufwand auf parallele Rechnerarchitekturen portieren zu können. Dem Management und dabei überwiegend dem parallelisierenden Übersetzer kommt die Aufgabe zu, parallel ausführbare Teilberechnungen zu finden und geeignet zu realisieren.

Bei einem explizit parallelen Programmiermodell, das überwiegend bei imperativen Sprachen eingesetzt wird, spezifiziert der Programmierer die Gesetzmäßigkeiten der parallelen Berechnungen gezielt. Auf dieser Grundlage ist das Management von der Aufgabe, Parallelität zu finden, entledigt und kann mit dem Wissen über die Problemlösung operieren. Die quantitative Leistung, die mit explizit parallelen Sprachen erzielt wird, ist in der Regel dementsprechend höher als bei impliziter Parallelität. Das Gewicht des Nachteils explizit paralleler Sprachen, Parallelität angeben zu müssen, hängt vor allem davon ab, in welchem Maße an das Aktivitätsträgerkonzept realisierungsnahe Eigenschaften gebunden sind. Befindet sich dieses Konzept auf einem ausreichend hohem Abstraktionsniveau, so ist der Unterschied bezüglich

der Praktikabilität zu implizit parallelen Sprachen gering.

#### 2.3.1.3 Task- und Daten-Parallelität

Bei explizit parallelen Programmiermodellen ist zu unterscheiden, ob Berechnungen oder Daten im Vordergrund des Programmiermodells stehen. Daten-Parallelismus basiert auf der parallelen Applikation genau einer Operation auf unterschiedlichen Elementen eines Datums, ähnlich der Verarbeitung von Daten- und Instruktionsströmen in SIMD-Architekturen. In der Regel führen alle Prozessoren des Systems genau ein Programm aus (SPMD). In Sprachen wie *High Performance Fortran* (HPF) [Ric93] spezifiziert der Programmierer zusätzlich zur Berechnungsvorschrift die Partitionierung der Daten. Der Übersetzer analysiert diese Information und trifft weitere Entscheidungen über die Erzeugung von Ausführungsfäden, deren Kommunikation und Synchronisation. In der Phase der Codeerzeugung manifestieren sich anschließend die Entscheidungen des Übersetzers in konkreten Maßnahmen. Deutlich zu sehen ist hierbei der nahtlose Übergang von den Entscheidungen des Programmierers an das System, was auf dieser Ebene einer engen Integration des Managements entspricht, bei einem gleichzeitig verhältnismäßig einfachen Programmiermodell. Allerdings ist das einfache Modell der Daten-Parallelität nur dann adäquat, wenn die Strukturierung der Daten tatsächlich mit den parallelen Berechnungen des Systems korrespondiert. In vielen Fällen trifft dies nicht zu. Beispiele sind irreguläre Datenstrukturen oder Anwendungen jenseits technisch-wissenschaftlicher Zwecke, wie langlebige parallele Diensterbringer in Client-/Server-Architekturen.

Stehen Berechnungen im Vordergrund des Modells, so spricht man von Task-Parallelität. Der Programmierer definiert in diesem Fall unterschiedliche Typen von Prozessen[12], deren Instanzen nach ebenfalls festzulegenden Gesetzmäßigkeiten kommunizieren und ihre Ausführungen synchronisieren. Abgesehen von Synchronisationspunkten treiben die Prozesse ihre Berechnungen nebenläufig voran und operieren dabei auf lokalen und ggf. auch gemeinsamen Daten. Dieses Programmiermodell wird wegen seiner Universalität in vielen verteilten und parallelen Sprachen [BST89] verwendet. In der Praxis haftet den Realisierungen dieses Modells Defizite an, die auf der oberflächlich betrachtet einfacheren Implementierung dieses Konzepts beruhen. Während es bei Daten-Parallelen Sprachen selbstverständlich ist, daß der Übersetzer aufwendige Analysen unternimmt und flexibel die abstrakt spezifizierten, feingranularen paralleler Berechnungseinheiten zu sinnvollen größeren Einheiten kombiniert, wird dieses Problem im Falle task-paralleler Ansätze meist ignoriert und lediglich ein Realisierungskonzept uniform an das Parallelitätskonzept der Sprache gebunden. Das Resultat ist, daß eine große Lücke zwischen den gewünschten Eigenschaften der parallelen Algorithmen und ihrer programmiersprachlichen Spezifikation klafft. Aus [GR88] stammt folgendes Beispiel eines effizienten parallelen Algorithmus, der die kürzesten Pfade zwischen allen Knotenpaaren in einem gewichteten Graph in $O(\log^2 n)$ bei Verwendung von $n^3$ Prozessoren bestimmt.

**Beispiel 2.17 (Paralleler Algorithmus – all pairs shortest path)**
*Der Algorithmus erhält als Eingabe die Eingabematrix $M$ der Kantengewichte des Graphen und liefert das Ergebnis in $M'$, mit*

$$\begin{aligned} M'(i,j) &= 0, & i &= j \\ M'(i,j) &= min\{M(i_0,i_1) + M(i_1,i_2) + \cdots + M(i_{k-1},i_k)\}, & i &\neq j \end{aligned}$$

*$m_{ij}$ und $q_{ijk}$ sind lokale Hilfsmatrizen des Algorithmus mit $1 \leq i,j,k \leq n$*

[12] nicht zu verwechseln mit dem Prozeßkonzept von Betriebssystemen

1. ***for*** $all\ i,j$ ***in parallel do*** $m(i,j) = M(i,j)$

2. ***repeat*** $log_2\ n$ ***times***
   ***begin***
   ***for*** $all\ i,j,k$ ***in parallel do*** $q(i,j,k) = m(i,j) + m(j,k)$
   ***for*** $all\ i,j$ ***in parallel do*** $m(i,j) = min\{m(i,j), q(i,1,j), q(i,2,j), \ldots, q(i,nj)\}$
   ***end***

3. ***for*** $all\ i,j$ ***in parallel do***
   ***if*** $i \neq j$ ***then*** $M'(i,j) = m(i,j)$ ***else*** $M'(i,j) = 0$

Während Parallelität auf dieser abstrakten Ebene, wie in Zeile 1, mit dem Wort **parallel** für Berechnungsschritte minimaler Granularität festgelegt wird, muß der Programmierer bei der Formulierung dieses Algorithmus mit Sprachen wie SR [And82], Ada [Ada83] und Orca [BKT92] Realisierungseigenschaften berücksichtigen, die invariant an das Aktivitätsträgerkonzept der Sprache gebunden sind. Im Beispiel von Ada wird das abstrakte TASK-Konzept meist durch Threads realisiert, während im Falle von SR UNIX-Prozesse eingesetzt werden. Aus Effizienzgründen ist es untragbar Berechnungseinheiten von solch geringer Granularität wie im Beispiel unmittelbar auf die entsprechenden Sprachkonstrukte abzubilden. Ein Programmierer ist deshalb dazu gezwungen, die Parallelität des Algorithmus bei der Formulierung in einem Programm selbst zu partitionieren. Diese Partitionierung ist nicht Teil des Problemlösungsverfahrens, sondern dient der Realisierung und ist somit Management. Wenngleich dieser Umstand in der Praxis akzeptiert wird, hat diese Vorgehensweise negative Auswirkungen sowohl bezüglich der Praktikabilität als auch der Effizienz:

1. Die ohnehin schwierige und fehlerträchtige Formulierung paralleler Programme wird erheblich erschwert und der Aufwand wird erhöht. Die Konzentration des Programmierers verlagert sich von den algorithmischen Eigenschaften des Problemlösungsverfahrens auf seine Realisierungseigenschaften.

2. Managementmaßnahmen müssen redundant pro Problemlösung durchgeführt werden.

3. Die Information, auf deren Grundlage der Programmierer einige seiner Entscheidungen stützen muß, ist ungeeignet. Eigenschaften, wie die Anzahl der Stellen, Lastsituation und Realisierungsmechanismen, unterliegen Veränderungen und können zum Zeitpunkt der Programmierung nicht im erforderlichen Umfang berücksichtigt werden.

Managemententscheidungen, die auf Grundlage unpräziser Information irreversibel getroffen werden, leiden generell unter ihrer schlechten Integration in das Gesamtmanagement. Dies trifft in diesem Fall für den Übergang von der Programmierung auf den Übersetzer und das Laufzeitsystem zu. Die Ursache dafür ist das mangelnde Abstraktionsniveau der Sprachkonzepte auf der einen Seite und auf der anderen Seite die ungenügende Flexibilität des Managementsystems, das nicht im ausreichenden Maße zwischen abstrakten- und Realisierungseigenschaften der Parallelität unterscheidet.

In der Tat ähnelt die Formulierung des Beispiel-Algorithmus eher einem daten-parallelen Stil. Diese Form wurde gewählt, um in diesem konkreten Beispiel die maximale Parallelität des Verfahrens im Abstrakten prägnant ausdrücken zu können. An der diskutierten Problematik ändert sich auch dann nichts, wenn der Algorithmus in einem task-parallelen Stil mit seiner maximal möglichen Parallelität angegeben wäre. Ein Programmierer, der eine der gängigen

task-parallelen Sprachen nutzt, müßte dennoch bei der Niederschrift des Algorithmus Managemententscheidung treffen und die Berechnungsvorschrift in einem für das System geeigneten Maß partitionieren.

Neben deutlich abgrenzbaren task- und daten-parallen Sprachen existieren Mischformen und Ansätze, die sich darum bemühen, die positiven Eigenschaften beider Ansätze zu vereinen. Ein Beispiel für eine Mischform ist die objektorientierte Sprache ODL [SL94] aus dem Projekt *diamonds* [BCSL93], bei der ein Programmierer Methodenaufrufe und Synchronisationseigenschaften explizit spezifiziert. Anschließend analysieren der Übersetzer und das Laufzeitsystem das Programm und entscheiden über die Gruppierung von Objekten und die Anzahl tatsächlicher Ausführungsfäden. Die flache Struktur der Objekte und Abwesenheit weiterer Konzepte zur präzisen Trennung von sequentiellen und parallelen Berechnung limitieren im Falle von ODL allerdings die Leistungsfähigkeit dieses Ansatzes, ohne erkennbare Vorteile hinsichtlich der Praktikabilität. Ein Versuch der Kombination von Daten- und Task-Parallelismus stellt die Sprache *Fx* [SY97] dar, in der HPF um task-parallele Direktiven erweitert wird, die der Festlegung von Prozessen und ihrer Zuordnung an Prozessor-Gruppen dienen. Zur Laufzeit beschränkt sich die Integration im wesentlichen auf das Wechseln zwischen daten-paralleler und task-paralleler Ausführung an im Programm festgelegten Punkten. Mit diesem Ansatz ist eine hohe Effizienz der Ausführung möglich bei jedoch hohem Preis bezüglich der Einfachheit bzw. Qualität der Programmierung. Im Falle von *Data Parallel Orca* [HB96] wurde der umgekehrte Weg gewählt; die Erweiterung einer task-parallelen Sprache um daten-parallele Konstrukte. Objekte können explizit partitioniert werden, um im Anschluß Operationen auf den Elementen parallel durchzuführen. Die Verteilung der Elemente auf die physischen Prozessoren muß der Programmierer selbständig vornehmen. Wenngleich dieser Ansatz es erlaubt, task- und daten-parallelen Konzepte gemeinsam zu nutzen, gelingt es auf diese Weise dennoch nicht, deren Vorteile zu vereinen und dadurch ihre Nachteile zu reduzieren. Das niedrige Abstraktionsniveau der daten-parallelen Erweiterungen führt dazu, daß der Programmierer einen Großteil der Managementlast ohne Unterstützung des Systems tragen muß. Zusätzlich wurden die heterogenen Konzepte von Task-/Daten-Parallelismus unreflektiert einer Sprache hinzugefügt, ohne daß ihre abstrakten Eigenschaften aufeinander abgestimmt wurden und zu neuen homogenen Konzepten verschmolzen wurden. Das Ziel, mehr Leistung mit einfacheren Mitteln zu ermöglichen, wird auf diese Weise verfehlt.

Dennoch machen diese Bemühungen deutlich, daß eine Annäherung der beiden Parallelisierungsarten wünschenswert ist. Um eine geeignete Balance zwischen der Einfachheit der Programmierung und Leistungsfähigkeit der Ausführung zu erzielen, dürfen jedoch nicht heterogene Konzepte lediglich addiert werden. Vielmehr ist folgende Vorgehensweise anzustreben:

1. Formulierung von Problemlösungen mit expliziter Spezifikation der Parallelität und Kooperation, über den der abstrakte Algorithmus verfügt, losgelöst von Realisierungseigenschaften.

2. Möglichst effiziente Realisierung des Programms durch das Managementsystem, das selbständig und flexibel anhand den aktuellen Angeboten über Realisierungseigenschaften entscheidet.

#### 2.3.1.4 Kooperation

Der dritte wichtige Faktor, der das Wechselspiel zwischen paralleler Sprache und parallelem bzw. verteiltem Management maßgeblich prägt, sind die Konzepte, die zur Festlegung

der Kooperation zwischen den nebenläufigen Teilberechnungen zur Verfügung stehen. Da die Spezifikation von Kooperation die Kenntnis der Ausführungen voraussetzt, sind Kooperationskonzepte eine Domäne explizit paralleler Sprachen, was deren größere Universalität erklärt.

Das Spektrum an Möglichkeiten, Kooperation bzw. Interaktion festzulegen, ist groß und es existieren unterschiedliche qualitative und quantitative Kriterien mit deren Hilfe sie klassifiziert werden, wie z. B. synchron/asynchron oder anhand der Bandbreite in schmalbandig/breitbandig, et cetera. Besonders stark werden Programmiermodelle und Managementsysteme in verteilten Umgebungen von der Unterscheidung zwischen folgenden Paradigmen beeinflußt:

**message passing (Nachrichtenaustausch):** Eine Nachricht wird von einer Berechnung konstruiert und mit einer Operation *send* explizit über einen Kommunikationskanal versendet. Nach Erhalt wird die Nachricht von den Empfängern mit der Operation *read* gelesen und verarbeitet.
*Quantitative Eigenschaften:* Wenngleich die Latenz des Nachrichtenaustausches hoch sein kann, gibt es in der Regel wenige Einschränkungen, wieviel Information eine Nachricht tragen kann. Message Passing kann hohe Bandbreiten bieten, wodurch es ein sehr effizientes Verfahren für die Übertragung großer Datenmengen sein kann. Die Nähe zu den physischen Fähigkeiten von NOW-Konfigurationen, die keinen gemeinsamen Speicher besitzen, erlaubt es, dieses Kooperationsmodell mit geringem zusätzlichem Managementaufwand sehr effizient auf die verteilte Hardware abzubilden.
*Qualitative Eigenschaften:* Um die Anzahl der Nachrichten minimieren und ihre Größe maximieren zu können, müssen die Daten einer parallelen Berechnungsvorschrift so partitioniert werden, daß die Daten, die von einer Berechnung benötigt werden, ihr zugeordnet sind. Diese Management-Aufgabe ist äußerst kompliziert und zieht komplexe Umgestaltungen des Algorithmus nach sich, die von dem Programmierer durchzuführen sind.

**shared memory (gemeinsamer Speicher):** Bei Kooperation über einen gemeinsamen Speicher werden von den Berechnungen gemeinsam zugreifbare Speicherobjekte genutzt. Geschriebene Werte können von Mengen von Berechnungen gelesen werden.
*Quantitative Eigenschaften:* Physisch gemeinsame Speicher bieten sehr hohe Bandbreiten mit niedriger Latenz. In einem verteilten System muß der gemeinsame Speicher jedoch letztendlich durch den Austausch von Nachrichten realisiert werden (s. DSM in 2.1.2.4). Das Verbindungsnetz besitzt eine höhere Latenz und geringere Bandbreite und stellt das Management vor die Herausforderung, die oben bei Nachrichtenaustausch erwähnte Partitionierung automatisch durchzuführen, was aufgrund der bereits festgelegten Berechnungseinheiten äußerst schwierig ist. In der Praxis ist die mit dieser Vorgehensweise erzielte Effizienz bislang unbefriedigend, sofern die Aufgabe nicht doch in Form zusätzlicher Direktiven partiell auf die Programmierung verlagert wird.
*Qualitative Eigenschaften:* Das Modell des gemeinsamen Speichers entspricht in vielen Fällen wesentlich besser den Eigenschaften paralleler Algorithmen auf abstrakter Ebene und wird deshalb als leichter handhabbar betrachtet. Bei der Programmierung ist in diesem Fall weitaus weniger Management zu betreiben. Auf diese Weise spezifizierte Programme besitzen deshalb größeres Potential für Skalierung mit wachsenden Hardwarekonfigurationen, erhöhte Portabilität und verbesserte Wartbarkeit.

Die starre Trennung von *message passing* und *shared memory* ist stark von der bottom-up betriebenen Virtualisierung der Eigenschaften der Hardware geprägt. Genauso wie bei Multiprozessorsystemen der physisch gemeinsame Speicher auf Ebene der Programmierung mit entsprechenden Konzepten verfügbar gemacht wird, spiegelt sich bei Multicomputern der Mangel des gemeinsamen Speichers meist auch in den Programmiersprachen wider. Um eine Balance zwischen Einfachheit und Effizienz finden zu können, die geeignet ist, um die Vorstellung qualitativ und quantitativ hochwertiger V-PK-Systeme zu verwirklichen, ist die starre Trennlinie zwischen Nachrichtenaustausch und Kooperation via gemeinsamen Speicher ebenso wenig hilfreich, wie die oben diskutierte Trennung von Daten- und Task-Parallelität. Auch in diesem Falle müssen zusätzliche Ebenen zwischen der Hardware und der Sprache eingezogen werden, die es erlauben auf Ebene der Sprache abstrahiert von den physischen Eigenschaften adäquate Konzepte anbieten zu können. Für die Formulierung paralleler Algorithmen werden unabhängig von den Fähigkeiten der Hardware, angepaßt an die Anforderungen der Algorithmen, von Fall zu Fall beide Formen der Kooperation benötigt. Das Management ist um die Realisierung der hinzugefügten Konzepte mit den Angeboten der Hardware zu expandieren.

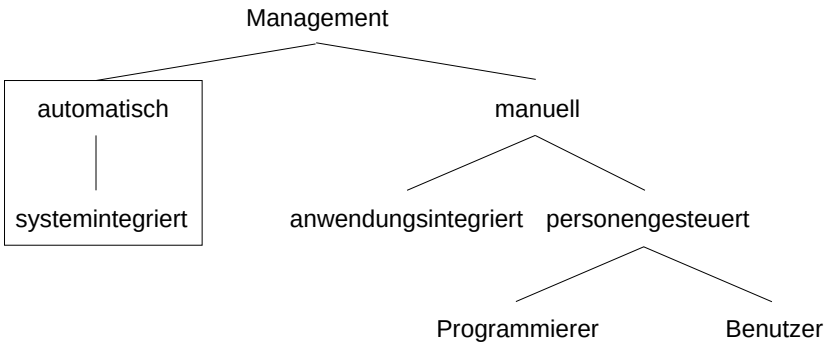


Abbildung 2.9: Zuordnung von Managementaufgaben

Insgesamt ist somit auch diese Frage ein Problem der Zuordnung von Managementaufgaben (s. Abb. 2.9). Relativ zur Position der abstrakten Problemlösung werden automatisierte Managementmaßnahmen ohne Eingriffe von Benutzern des Systems exklusiv von vorbereiteten Komponenten des Systems vorgenommen, die nicht Teil der Problemlösung sind. Manuelles Management erfordert, daß Maßnahmen entweder von dem Programm der Problemlösung, d. h. durch den Programmierer, oder von Nutzern der Problemlösung durchgeführt werden. Bei automatisierten Maßnahmen spricht man auch von *systemintegriertem* Management, während manuelle Maßnahmen *anwendungsintegriertes* Management genannt werden. Die kurze Diskussion der Sprachkonzepte hier zeigt, daß die Trennlinie schwierig zu ziehen ist, da intensive Abhängigkeiten zwischen dem Programmiermodell und dem Management sowie der Einfachheit und Leistungsfähigkeit bestehen.

### 2.3.2 Konstruktionsmethodik

Managementsystemen kommt die Aufgabe zu, eine Verbindung zwischen den Konzepten der Sprache und den Angeboten der physischen Hardware herzustellen. Diese Aufgabe ist in jedem Fall umfangreich und bedarf einer Vielzahl von Entscheidungen, sowohl bezüglich grundlegender architektureller Festlegungen, als auch feinstgranularer Details. In verteilten Systemen

ist der zu überbrückende Weg besonders weit, da zusätzlich zu den üblichen Aufgaben die physische Verteilung überwunden werden muß. Wünsche nach Opazität der Verteilung auf Ebene der Programmierung, Automatisierung des Managements und Erhöhung des Abstraktionsniveaus aus Gründen wie sie in Abschnitt 2.2.1 skizziert wurden, verlängern die Distanz zwischen den Anforderungen und den Angeboten zusätzlich, wodurch sich die Komplexität der Managementaufgabe als Produkt der Anzahl zu treffender Entscheidungen und deren Tragweite weiter erhöht. Das Problem der Entscheidungsfindung ist in dieser Art für den Bereich der Management- bzw. Betriebssysteme spezifisch.

- Im Gegensatz zu Problemlösungen oder *Anwendungen*, die auf einem Managementsystem zur Ausführung gebracht werden, erstrecken sich Managementsysteme über eine wesentlich größere Anzahl an Abstraktionsebenen. Hinzu kommen insbesondere technische Randbedingungen gerätenaher Ebenen.

- Managementsysteme müssen in der Lage sein, das gesamte Spektrum der qualitativen - und quantitativer Anforderungen, die von seinen Nutzern – den Anwendungen – gestellt werden können, zu befriedigen. Als gemeinsame Basis für die Ausführung unterschiedlicher Berechnungen muß das Management sowohl die Maxima als auch die Minima potentieller Anforderungen erfüllen und dynamisch die Balance zwischen Qualität und Quantität variieren können.

Wegen der hohen Anforderungen rückt die Frage nach der Art und Weise *wie* Konzepte realisiert sind bei Managementsystemen wesentlich stärker in den Vordergrund als das sonst der Fall ist. Geeignete Entscheidungen, wie z. B. Wahl zwischen B-Bäumen und Hashing oder Bevorzugung früher gegenüber verzögerter Maßnahmen (*prefetching*, *on-demand/lazy evaluation*), sind für die Eignung eines Managementsystems entscheidend. Ungeachtet der großen Bedeutung der Entscheidungsfindung sind aus dem Bereich der Betriebssysteme keine Verfahren bekannt, die es erlauben würden, die notwendigen Entscheidungen anhand fundierter Kriterien zu finden. In der Praxis überwiegen einfache, intuitive Vorgehensweisen:

**Empirie:** Entscheidungen werden anhand von Erfahrungen in der Vergangenheit getroffen. Die präzisen Eigenschaften der aktuellen Anforderungen, d. h. der Gesamtkontext spiegelt sich in diesen Erfahrungen aufgrund ihrer hohen Dynamik nicht wider.

**ad hoc:** Zur Entscheidungsfindung werden ausschließlich Alternativen und Konsequenzen betrachtet, die in einem engen lokalen Zusammenhang mit der Problemstellung stehen oder bereits bekannt sind.

**trial & error:** Bei dieser Vorgehensweise werden u. U. noch weniger Alternativen und Konsequenzen berücksichtigt als dies bei einer ad hoc Konstruktion der Fall ist. Dabei werden Fehlschläge und wiederholte bzw. nachträgliche Neukonstruktionen sowie Nachbesserungen eingeplant.

Obwohl diese Methoden vom Standpunkt der Systematik betrachtet völlig unzulänglich erscheinen, besitzen sie auch einige Vorteile. *Ad hoc* Konstruktion erlaubt es, Konzepte bzw. Verfahren mit geringem Aufwand zu entwickeln und schnell auf neue Anforderungen zu reagieren. Das typische Explorieren möglicher Lösungen in der *trial & error* Methode kann oft schon in kurzer Zeit zu akzeptablen Ergebnissen führen, bringt Erkenntniszugewinn und entspricht auf natürliche Weise langfristig angelegten Entwicklungs- und Revisionsschritten, wie sie bei unpräziser Kenntnis der Anforderungen unausweichlich sind.

#### 2.3.2.1 Konstruktionsrichtung

Die Aufgabe, Management zu konstruieren, kann vereinfacht als die Konstruktion einer Abbildung von den Quell- auf die Zielressourcen betrachtet werden.

**Bemerkung 2.18 (Management-Konstruktion)**
*Sei $\mathcal{Q}$ die Menge aller möglichen Anforderungsklassen und $\mathcal{Z}$ die Menge aller möglichen Angebotsklassen. $Q \in \mathcal{Q}$ und $Z \in \mathcal{Z}$. Der Prozeß der Konstruktion eines Managements ist die Konstruktorfunktion $\Omega$.*

$$\Omega : \mathcal{Q} \times \mathcal{Z} \rightarrow (Q \rightarrow Z)$$

$\Omega$ liefert die injektive Managementfunktion $m : Q \rightarrow Z$, die Anforderungen $q \in Q$ auf die Angebote $z \in Z$ abbildet. $m$ ist aus einer Vielzahl elementarer Transformationssschritte komponiert, d. h.

$$\begin{array}{ll} m & : Q \rightarrow Z \qquad m = m_1 \circ m_2 \circ \ldots \circ m_n \\ m_i & : R_{i-1} \rightarrow R_i \quad R_0 = Q, R_n = Z \end{array}$$

Anhand der Reihenfolge in der die $R_i$ und $m_i$ von $\Omega$ konstruiert werden, ist zwischen bottom-up bzw. top-down gerichteter Vorgehensweise zu unterscheiden.

**Angebotsorientiert: Bottom-Up Virtualisierung**
Bei der bottom-up orientierten Vorgehensweise werden die Angebote (z. B. Hardware) als Startpunkt der Konstruktion gewählt. $\Omega$ beginnt mit der Festlegung von $(R_{n-1}, m_n)$. Der Konstruktionsvorgang orientiert sich somit an vorhandenen Betriebsmitteln und schafft durch Virtualisierung bzw. Abstraktion neue Ressourcen mit Eigenschaften, die überwiegend Kombinationen aus Eigenschaften existierender Komponenten sind. Zu Beginn der Konstruktion stehen aufgrund der technischen Ausrichtung quantitative Eigenschaften im Vordergrund. Mit wachsendem Abstraktionsniveau (d. h. sinkendem $i$ bzgl. $m_i$) treten die quantitativen Eigenschaften in den Hintergrund und qualitative Merkmale gewinnen an Bedeutung. Abweichungen von den Zieleigenschaften der $q \in Q$ werden erst spät deutlich, wenn sich die Konstruktion an dieses Niveau annähert und müssen dann entweder durch Anpassungen behoben werden oder – der häufigere Fall – die Eigenschaften von $Q$ werden revidiert.

Bottom-up Konstruktion führt deshalb überwiegend zu sehr effizienten Verfahren, die auf Ebene der Programmierung jedoch häufig nicht den Anforderungen entsprechen und dort mit hohem Aufwand an die Bedürfnisse angepaßt werden müssen. Da die Angebote meist präziser als die Anforderungen spezifiziert sind und zusätzlich meßbare Effizienz-Vorteile erzielt werden können, werden Managementsysteme in der Praxis dennoch überwiegend bottom-up entwickelt. Dies spiegelt sich für gewöhnlich in allgemein gehaltenen Primitiven wider, die nicht die Bedürfnisse erfüllen. UNIX-Systeme bieten etwa schwergewichtige Prozesse an, die inadäquat sind, um feingranulare Parallelität im Sinne von 2.3.1.3 zu spezifizieren. Um dem Wunsch nach mehr Flexibilität Rechnung zu tragen, wurden in Systemen wie Mach 3.0 [ABG+86] leichtgewichtige Threads eingeführt. Doch auch hierbei handelt es sich um ein bottom-up entwickeltes Konzept, an das ein relativ großer Kellerspeicher, sowie ein vordefinierter *port name space* gebunden ist. Um tatsächlich leichtgewichtige Aktivitäten ungeachtet der Realisierungseigenschaften auf einem hohen Abstraktionsniveau formulieren zu können, ist auch diese Abstraktion ungeeignet. Ein weiteres wichtiges Beispiel sind Datei-Systeme. Der ursprüngliche Wunsch nach Persistenz erarbeiteter Daten muß anwendungsseitig wiederholt mittels aufwendiger Konvertierungen sowie fehlerträchtiger Speicher- und Leseoperationen realisiert werden.

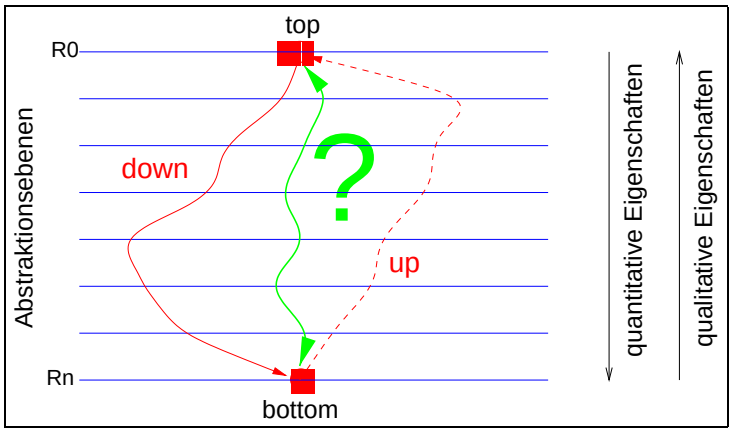


Abbildung 2.10: Top-down und bottom-up gerichtete Konstruktion

**Anforderungsorientiert: Top-Down Realisierung**
Der entgegengerichtete Ansatz, der die qualitativen Mängel der bottom-up Methode wirksam überwindet, ist die Konstruktion an den Anforderungen zu orientieren. $\Omega$ startet mit der Festlegung von $(R_1, m_1)$ und treibt die Konstruktion von $m$ schrittweise in Richtung $Z = R_n$ und $m_n$ voran. Die Eigenschaften der Zielressourcen $R_{i+1}$ und der Transformationsschritte $m_i$ werden durch Konkretisierung, d. h. Hinzunahme von Realisierungseigenschaften und Zerlegung der Eigenschaften von $R_i$, angepaßt an deren Anforderungen festgelegt. Invers zur bottom-up Methode stehen bei Beginn der Konstruktion qualitative Eigenschaften im Vordergrund, die mit sinkendem Abstraktionsniveau von quantitativen Aspekten ersetzt werden. Auch in diesem Fall entstehen Abweichungen vom Konstruktionsziel, welches in diesem Fall die Fähigkeiten der Hardware sind. Im Gegensatz zu bottom-up existiert in aller Regel jedoch nicht die Option, diese zu revidieren. Stattdessen müssen auf niedrigen Abstraktionsebenen gegebenenfalls umfangreiche Anpassungen vorgenommen werden um die gewünschte Abbildung auf die tatsächlichen Angebote zu erreichen, was entsprechende Effizienzverluste zur Konsequenz hat. Der enge Spielraum der Forderung, daß die Angebote $Z$ determiniert sind, erschwert zunächst das Finden einer akzeptablen Balance zwischen Qualität und Effizienz. Auf der anderen Seite fördert die Loslösung von existierenden Konzepten den kreativen Entwurf neuer Lösungswege, aus der sich langfristig Vorteile gegenüber der eher ad hoc motivierten bottom-up Konstruktion ergeben.

Das Problem der Konstruktion eines qualitativ und quantitativ leistungsfähigen Managements kann somit als Suchproblem verstanden werden (Abb. 2.10). Ausgehend von dem Abstraktionsniveau der Quelle sind Transformationsschritte für den Übergang auf das Abstraktionsniveau des Zieles zu finden. Die Länge des Pfades auf den einzelnen Abstraktionsebenen, sowie seine Gesamtlänge geben Aufschluß über die qualitativen und quantitativen Eigenschaften der konstruierten Transformation. Bei dem Versuch, einen Pfad von der Quelle zum Ziel zu finden, entstehen unabhängig von der Konstruktionsrichtung *bottom-up* oder *top-down*, Abweichungen vom Ziel, die dazu führen, daß entweder die Anforderungen oder die real existierenden Möglichkeiten zunächst verfehlt werden. Diese Aberration wird in der Regel erst gegen Ende des Pfades sichtbar und muß dann mit hohem Aufwand korrigiert werden. Für den Nutzer des Systems werden diese Korrekturen durch Angebote, die nicht den Anforderungen entsprechen (bottom-up) oder Ineffizienzen (top-down) spürbar. Die größere Zahl

zu überwindender Abstraktionsebenen in verteilten Umgebungen forciert diese Problematik.

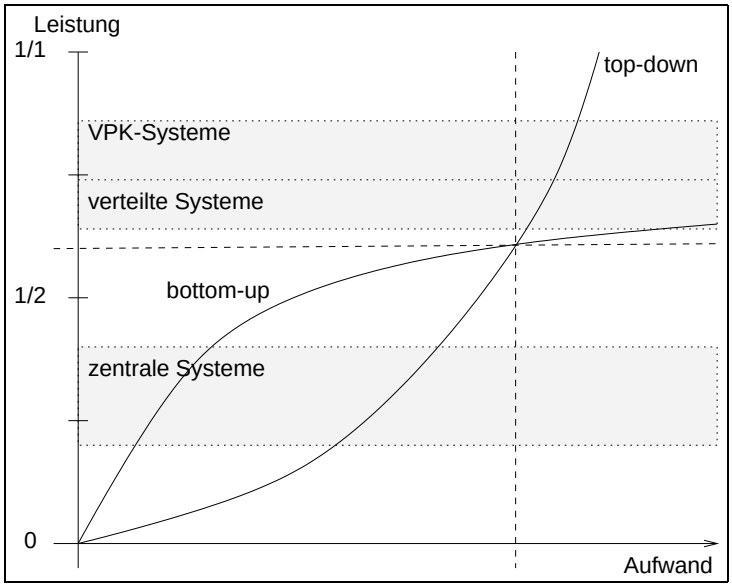


Abbildung 2.11: Aufwand und Leistung bei top-down und bottom-up

Abbildung 2.11 skizziert das Verhältnis zwischen Aufwand und Leistungspotential der beiden Ansätze. Bottom-up kann bereits mit geringem Aufwand zu akzeptabler Leistung führen, wobei mit Leistung sowohl qualitative als auch quantitative Eigenschaften gemeint sind. Top-down Konstruktion verursacht zunächst wesentlich größeren Aufwand, da Teilentwürfe nicht evaluiert werden können, das Transformationsziel nicht angepaßt werden kann, et cetera. Aufgrund der größeren Flexibilität und der Förderung kreativer Entwicklungsschritte ist anzunehmen, daß die top-down Konstruktion langfristig größeres Leistungspotential besitzt. Gemäß der Diskussion der Beweggründe für verteiltes Rechnen in 2.2 ist davon auszugehen, daß die Anforderungen, die an verteilte Systeme gestellt werden, deutlich höher sind als bei zentralen Systemen. Um diese gestiegenen Anforderungen beherrschen zu können, wird ein Verfahren benötigt, das die zielgerichtete Konstruktion eines zumindest suboptimalen Pfades ermöglicht. Dabei muß es sich um eine Kombination aus überwiegend top-down orientierter Vorgehensweise handeln, bei der zusätzlich Teilpfade bottom-up entwickelt werden, um Abweichungen möglichst früh erkennen und die maximale Aberration sowohl von den qualitativen als auch den qualitativen Eigenschaften minimieren zu können.

#### 2.3.2.2 Design Patterns

Ein interessanter Ansatz, der sich mit dem Finden adäquater Entscheidungen beschäftigt, wird im Bereich des Software Engineering mit *Design Patterns*[13] [App97, BMR+96, EGV94] verfolgt. Das Ziel lautet, Information über Entwurfsprobleme, ihrer Lösung und den resultierenden Konsequenzen systematisch zu sammeln, um in gleichen oder ähnlichen Situationen die bereits erarbeiteten Lösungen wiederverwenden zu können. Kostspielige Redundanz bei der Entwicklung komplexer Systeme kann auf diese Weise bei gleichzeitiger Steigerung der

[13] Entwurfsmuster

Leistung von Softwaresystemen vermieden werden. Entwurfsmuster existieren in vielen Varianten und werden in der Literatur dementsprechend auf unterschiedliche Weise definiert. Eine mögliche Erklärung lautet:

**Bemerkung 2.19 (Entwurfsmuster)**
*Ein Entwurfsmuster ist eine benannte Abstraktion einer erfolgreichen konkreten Lösung zu einem Problem, das in verschiedenen nicht-spezifischen Zusammenhängen wiederkehrt.*

Zur Beschreibung von Entwurfsmustern existieren verschiedene Formate, z. B. *alexandrinisch*, *GoF* und *kanonisch*. Abgesehen von den Details der verschiedenen Formate gehören zu einer Entwurfsmusterbeschreibung stets einige *essentielle* Komponenten:

| | |
|---|---|
| **Name** | Ein möglichst bedeutungsweisender Name. |
| **Problem** | Erklärung des Problems, für welches das Entwurfsmuster gedacht ist. |
| **Kontext** | Voraussetzungen, die im Zusammenhang mit der Wiederkehr des Problems und der Lösungen stehen. Der Kontext informiert somit über die Anwendbarkeit des Musters. |
| **Motivation** | Kräfte, Einschränkungen und ihre Wechselwirkungen mit den beabsichtigten Zielen. Häufig wird ein konkretes Szenario zur Motivation des Musters herangezogen. |
| **Lösung** | Konkrete Hinweise, wie das Muster realisiert werden kann. |
| **Beispiele** | Mindestens eine erfolgreiche Applikation des Musters. |
| **Verweise** | Hinweise auf andere Muster, die ähnlich sind oder zusammen mit diesem Muster eingesetzt werden. |

Entwurfsmuster, die einer gemeinsamen Domäne, z. B. CORBA-Systeme [MM97], zugeordnet werden können, werden zu *Musterkatalogen* und *Mustersprachen* zusammengefaßt. Darüber hinaus existieren Sammlungen von Mustern, die in einem komplexen Softwaresystem zusammenwirken und deshalb *Mustersysteme* genannt werden.

In [Cro97] unternimmt Crowley den Versuch, von der üblichen phänomenologischen und faktischen Betrachtung [NS98, SPG91, Tan92, Nut92, Wec89] der Architekturen, Verfahren und Mechanismen in Betriebssystemen abzurücken und statt dessen Konstruktionsprinzipien zu erarbeiten und als Design Patterns zu beschreiben. Damit wird ein wichtiger Schritt unternommen, um die Konstruktionsaufgabe zu systematisieren. Allerdings zeigen sich dabei auch die Schwächen von Design Patterns in Bezug auf ihren Nutzen für das Finden adäquater Entwurfsentscheidungen bei der Konstruktion eines neuen Systems.

Das Sammeln vorexerzierter Lösungen führt unweigerlich zu einer stark empirisch geprägten Vorgehensweise mit den Nachteilen, die bereits in Kapitel 1 diskutiert wurden. Veränderte Anforderungen, wie die notwendige Flexibilisierung von Mangementverfahren in verteilten Systemen, werden entweder nicht erkannt oder finden keine ausreichende Berücksichtigung.

Das zentrale Problem der Design Patterns ist allerdings die Abwesenheit von Kriterien für die systematische Erarbeitung und Klassifikation der Muster. Die Eigenschaften von Design Patterns sind in aller Regel in Prosa oder mit Bildern beschrieben. Zur Illustration soll das Beispiel in Tafel 2.3 dienen, das verkürzt aus [Cro97] entnommen wurde. Die Erarbeitung des Grundprinzips der Verzögerung von Entscheidungen ist äußerst nützlich. Die Wahl der Beispiele ist allerdings ebenso willkürlich wie die der Verweise. Durch den Mangel an fundierten

| | |
|---|---|
| **Name** | Verzögertes Binden |
| **Überblick** | Verzögertes Binden ist eine Möglichkeit Flexibilität durch das Zurückstellen einer Entscheidung zu gewinnen. |
| **Motivation** | Virtueller Speicher ist ein Beispiel für verzögertes Binden. Erst wenn auf eine Seite zugegriffen wird, wird eine Kachel an die virtuelle Seite gebunden. |
| **Beispiele** | Virtueller Speicher, Routing, Keller-Allokation, Kodierung von Tastaturereignissen |
| **Konsequenzen** | Präzisere Information zum Zeitpunkt der Entscheidung, frühes Binden kann durch die Zusammenfassung einzelner Bindungen effizienter sein |
| **Verweise** | Statisches/Dynamische Teilen, Raum/Zeit Ausgleich |

Tabelle 2.3: Design Pattern „verzögertes Binden“

Strukturierungskriterien entartet die Zielsetzung, wiederverwendbare Designs zu erarbeiten, zu einer flach strukturierten phänomenologischen Sammlung von Fakten. Da somit auch die Verweise zwischen den Mustern keiner einheitlichen hierarchischen Gliederung unterliegen, ist der Graph der Muster und Verweise nicht geeignet um zielgerichtet Top-Down oder Bottom-Up zu konstruieren. Darüber hinaus enthalten die Sammlungen redundante Muster, da keine Kriterien existieren, um Ähnlichkeit oder Gleichheit festzustellen. Ebenso ist das Auffinden geeigneter Muster schwierig, da keine Möglichkeit existiert die gewünschten Eigenschaften bei der Suche präzise zu spezifizieren. Zuletzt wären grundlegend Kriterien auch dafür notwendig, um bei Bedarf neue Muster generieren zu können.

Neben den Anwendungsdomänen existieren allerdings nur äußerst rudimentäre Klassifikationen von Entwurfsmustern, wie die Unterscheidungen zwischen Architektur-, Konzept- und Implementierungsmuster. Der Trend wiederverwendbare Entwurfsmuster zu erarbeiten ist zweifellos trotz dieser Schwächen ein interessanter Ansatz um den Entscheidungsprozeß bei der Konstruktion komplexer Systeme zu unterstützen. Um damit Vorteile bei der Konstruktion von Managementsystemen erzielen zu können, müssen jedoch dringend Kriterien gefunden werden, die erstens eine systematische Strukturierung der Entwurfsmuster ermöglichen und zweitens als Basis für die Formalisierung der Beschreibung von Mustereigenschaften dienen können.

#### 2.3.2.3 Betriebssystem-Strukturen

Weitere Anhaltspunkte für die Systematisierung der Konstruktion von Managementsystemen finden sich bei Bemühungen, den Raum der Komponenten von Betriebssystemen und ihre Abhängigkeiten zu strukturieren. Basierend auf einer vertikalen Separation von Aufgaben, wie Speicher-, Rechen- und Kommunikationsfähigkeit, und einer horizontalen Trennung von Abstraktionsniveaus wird versucht, Einheiten bzw. *Module* mit wohldefinierten inneren Eigenschaften und ebensolchen Bezügen zu anderen Modulen zu bilden. Für die Beurteilung der Vor- und Nachteile der verschiedenen Strukturierungsansätze, ist es hilfreich, das betrachtete System $\mathcal{S} = (K, A)$ als gerichteten Graph zu modellieren, der aus der Menge der Komponenten $K$ und den gerichteten Abhängigkeiten $A \subseteq K \times K$ besteht.

Die allgemeinste Architektur ist die des **Monolithen**. Die einzelnen Komponenten von $\mathcal{S}$ sind klein und die Abhängigkeiten $A$ unterliegen keinen Einschränkungen. Jedes $k \in K$

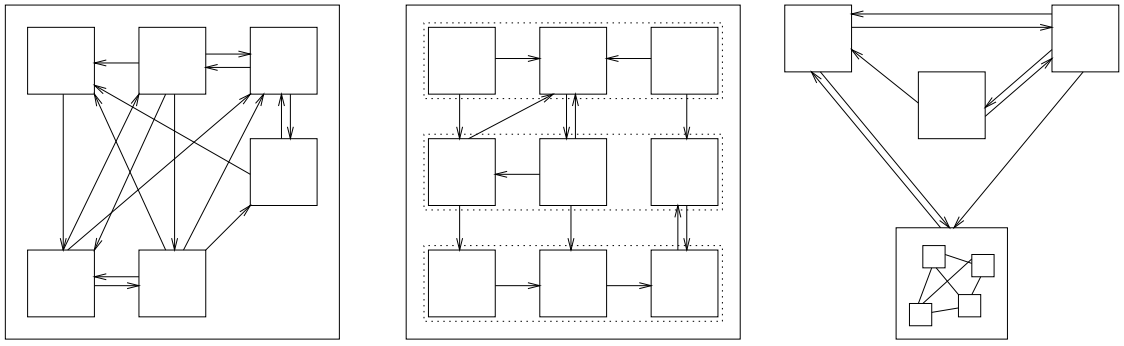


Abbildung 2.12: Strukturierungsansätze

kann Kanten zu beliebigen anderen Komponenten besitzen. Die Isolation oder Substitution einzelner Komponenten ist aufgrund der vielfältigen Abhängigkeiten schwierig und die Flexibilität des Designs ist gering. Gleichzeitig wird hohe Leistung erzielt, da $|K|$ minimiert und Bezüge zwischen den Komponenten angepaßt an die Erfordernisse ohne Indirektionen hergestellt werden können. Die Abhängigkeiten des monolithischen Modells sind jedoch schwer zu beherrschen, da keine Methoden existieren um mit den vielfältigen Bezügen systematisch oder automatisiert zu operieren. Dies hat negative Auswirkungen auf die Wiederverwendbarkeit, Wartung und Verifikation monolithischer Systeme und ihrer Komponenten.

Konsequenterweise wird versucht, Module mit größerer Granularität festzulegen, deren wechselseitige Abhängigkeiten eingeschränkt sind und bestimmten Gesetzmäßigkeiten genügen. Bei einem **Schichtenansatz** wird von $\mathcal{S}$ gefordert, daß der Graph nach Zusammenfassung von Knoten und Kanten mit genau zwei Farben gefärbt werden kann. Anders formuliert werden Ebenen $\{E_1, \ldots, E_n\}$ gebildet, so daß die einer Ebene $E_i$ zugeordneten Komponenten ausschließlich Abhängigkeiten zu Komponenten besitzen, die den Ebenen $E_i, E_{i-1}$ und $E_{i+1}$ zugeordnet sind. Der Vorteil der Schichtung sind handhabbare Einheiten mit Eigenschaften, die über die der atomaren Bestandteile hinausgehen und überschaubare Abhängigkeiten besitzen. Ebenen und Module können mit geringerem Aufwand ausgetauscht werden. Übergänge zwischen Ebenen und Komponenten sind abgestuft kontrollierbar. Die Voraussetzungen um Eigenschaften von $\mathcal{S}$ zu verifizieren sind deutlich besser als bei einem Monolithen. Der Preis dieses Modells ist a) die Notwendigkeit, zusätzliche Komponenten in $K$ aufnehmen zu müssen, die nicht funktional motiviert sind, sondern als Vermittler dienen. b) muß ein Teil der ursprünglich unmittelbaren Bezüge auf Grundlage der Vermittlerkomponenten durch transitive bzw. indirekte Abhängigkeiten ersetzt werden. Von $|K|$ und $|A|$ ist deshalb zu erwarten, daß sie bei gleicher Funktionalität von $\mathcal{S}$ größer sind als bei einer äquivalenten monolithischen Architektur. In erster Linie bedeutet das zusätzlichen Aufwand bei der Entwicklung und Effizienzverluste bei der Nutzung von $\mathcal{S}$. Erfahrungen mit dem Schichtenansatz wie mit dem Betriebssystem THE [Dij68] zeigen weiter, daß sich die intensiven funktionalen Abhängigkeiten in komplexen Systemen oft nur schwerlich und durch Produktion zahlreicher Vermittler und Indirektionen auf Schichten abbilden lassen.

Im Bereich der Betriebssystemkerne wurde in den vergangenen Jahren intensiv versucht, Trennlinien um Mengen von Komponenten mit substantieller Funktionalität und ihren Abhängigkeiten zu ziehen und auf diese Weise eine „minimale“ gemeinsame Basis zu isolieren.

Die aus $\mathcal{S}$ isolierte Komponentenmenge wird im Falle der Betriebssystemkerne **Mikro-** oder **Nanokern** genannt, um ihren geringen Umfang zu betonen [HK93, BCE+94, ABG+86]. Die besondere Motivation dieser Vorgehensweise im Bereich der Kerne basiert auf dem spezifischen Problem der Gewährleistung von Sicherheit und Zuverlässigkeit. Indem der privilegierte Teil des Kerns verkleinert wird, verringert sich der Aufwand, der erforderlich ist, um wichtige Eigenschaften des Systems $\mathcal{S}$ für das der Kern die Grundlage ist, zu überprüfen und ggf. nachzuweisen. Die Kosten sind ähnlich wie bei der Bildung von Schichten: zusätzliche Komponenten und Indirektionen. Ein Trend, der sich deshalb abzeichnet, ist die Verwendung erweiterbarer mikro-basierter Architekturen in der Entwicklung und Übergang zu engeren Bindungen für den produktiven Einsatz [PDZ97] nach erfolgtem Nachweis der gewünschten Eigenschaften. Generell bietet die Zerlegung von $\mathcal{S}$ in ein miniaturisiertes $\mathcal{S}_{min}$ und ein Residuum $\mathcal{S}_{res}$ nur geringe Fortschritte im Hinblick auf die Strukturierung der Abhängigkeiten $A$ von $\mathcal{S}$.

Die hier exemplarisch angeführten Strukturierungsprinzipien repräsentieren Managemententscheidungen auf einem sehr hohen Abstraktionsniveau. Mit einer abstrakten Festlegung dieser Art wird allerdings häufig bereits eine konkrete Implementierung assoziiert. Bei einem geschichteten Ansatz hat die Spezifikation der Übergänge zwischen Schichten mittels Signaturen von Prozeduren meist zur Folge, daß die entworfene Architektur im nächsten Entwicklungschritt auf Unterprogrammsammlungen und Aufrufe zwischen diesen abgebildet wird. Diese Assoziation ist naheliegend, jedoch inadäquat, da sie unausweichlich zu einem großen Überhang zur Laufzeit führt und Alternativen dazu existieren. Aus dem gleichen Grund ist es aus der Sicht des Gesamtmanagements nicht förderlich, einen Bruch zwischen Entwurfs- und Implementierungsentscheidungen vorzunehmen, der meist zur Folge hat, daß große Energie für den Entwurf aufgebracht werden und die Implementierung ad hoc durchgeführt wird. Stattdessen müssen die hier aufgeführten architekturellen Festlegungen als frühe Managemententscheidungen betrachtet werden, die systematisch und geschachtelt weiterzuführen sind, um den gesamten Pfad von den Anforderungen zu den Angeboten zielgerichtet integriert zu konstruieren.

### 2.3.3 Realisierung

Neben der methodischen Entwicklung des V-PK-Managements ist eine weitere wesentliche Forderung der Aufgabenstellung dieser Arbeit, daß die erarbeiteten Konzepte mit Schwerpunkt auf das Transformationsinstrumentarium in Form arbeitsfähiger Verfahren realisiert und anschließend evaluiert werden (Anforderungen 3 und 4 auf S. 8). Hierfür steht ebenfalls ein großes Spektrum an Alternativen zur Verfügung. Die Selektion einer konkreten Vorgehensweise und der eingesetzten Mittel müssen ihrerseits als Bestandteile des Geflechts aus Managemententscheidungen und Abhängigkeiten zwischen diesen betrachtet werden, um den oben genannten Bruch der durchgängigen Transformation von den Anforderungen auf die Angebote zu verhindern. Letztendlich wird die maximale Leistung (sowohl Qualität als auch Quantität) des V-PK-Managements von dem schwächsten Glied in der Kette der Entscheidungen bestimmt.

Der Übergang von dem Entwurf zur Implementierung ist aufgrund der vielfältigen wechselseitigen Abhängigkeiten ohnehin fließend und die übliche Grenzziehung erscheint von diesem Standpunkt aus eher willkürlich. Faktisch unterscheidet sich die Implementierung von dem Entwurf darin, daß die getroffenen Festlegungen verhältnismäßig konkrete Eigenschaften des Systems betreffen, die darüber hinaus möglicherweise bereits in einer Programmiersprache

formuliert werden können oder diesem Niveau zumindest nahe kommen.

Ungeachtet dieses Kontinuums werden in den folgenden Abschnitten unter dem Begriff „Realisierung“ Methoden herausgegriffen, die üblicherweise als Teil der „Implementierung“ von Managementsystemen angesehen werden. Ziel dieser Analyse ist, einen Raum von Alternativen aufzuspannen, der helfen soll, den Einfluß und das Potential verschiedener Implementierungsmöglichkeiten auf das Managementsystem besser zu berücksichtigen und von der besonders in den letzten Phasen der Entscheidungsfindung beliebten ad hoc oder empirischen Vorgehensweise abzurücken.

#### 2.3.3.1 Implementierungsstrategie

Implementierte Managementsysteme sind komplexe Software-Systeme mit $>$ 10 Millionen Zeilen Programmtext[14], die überwiegend von Hand kodiert werden müssen, da nach wie vor keine geeigneten generativen Verfahren existieren. Der Aufwand und die wirtschaftlichen Kosten, die sich hinter diesen Systemen und den darin enthaltenen hochoptimierten Verfahren verbergen, beeinflußt die Planung und Realisierung neuer Systeme maßgeblich. Zum einen zwingt sowohl der bereits investierte als auch der neu zu leistende Aufwand bewußt oder unbewußt zu Kompatibilität mit den existierenden Realisierungen. Zum anderen erscheint die vollständige Neu-Realisierung eines besseren Systems in absehbarer Zeit in Anbetracht des umfangreichen Wissens, das exklusiv in den existierenden Realisierungen verarbeitet ist, illusorisch. Ebenso ist es ein Faktum, daß die Mengen existierender - und neu zu implementierender Verfahren nicht disjunkt sind. Der Umgang mit existierenden Produkten bei der Implementierung neuer Konzepte hat allerdings einen starken Einfluß auf die Flexibilität und die Integrität des zu realisierenden Managements. Bei der Wahl einer geeigneten Ausgangsbasis sollte aus diesem Grund von Fall zu Fall sorgfältig abgewogen werden. Zur Einordnung des Umgangs mit übernommenen Realisierungen bieten sich die beiden folgenden Kriterien an:

**Umfang:** Der Umfang bestimmt den Anteil, den übernommene Verfahren in dem zu konstruierenden neuen Management einnehmen werden. Für eine erste Charakterisierung genügt das qualitative Spektrum: *unwesentlich*, *signifikant* und *überwiegend*.

**Art:** Bei der Übernahme existierender Verfahren sind zwei grundlegend verschiedene Arten der Wiederverwendung zu unterscheiden. Bei der *unmodifizierten* Integration werden an den übernommenen Komponenten keine wesentlichen Veränderungen vorgenommen. Im Gegensatz dazu werden bei einer *modifizierten* Integration die Eigenschaften der Verfahren an die veränderten Anforderungen adaptiert.

Anhand dieser Kriterien lassen sich folgende Vorgehensweisen charakterisieren, die unterschiedliche Auswirkungen auf den Aufwand, die Flexibilität und die Integrität haben:

**additiv:** Die Implementierung besteht überwiegend aus unmodifiziert übernommenen Verfahren. Diese Vorgehensweise ist in der Praxis am weitesten verbreitet, da sie weitreichende Kompatibilität gewährleistet und vermeintlich geringen Aufwand verursacht. Die Flexibilität des Systems wird von den übernommenen Konzepten diktiert. Bei der Integration der neuen Fähigkeiten müssen an Schnittstellen Transformatoren eingesetzt werden, was sich in einschneidenden Kompromissen und Reibungsverlusten niederschlägt.

[14] bezieht sich auf ein komplettes, zentrales UNIX-Betriebssystem; Kern + Übersetzer + Binder + etc.

Nicht selten führt die mangelhafte Integration der additiven Vorgehensweise zu Inkonsistenz, wie es am Beispiel nachträglicher Thread-Erweiterungen und existierender Bibliotheken (z. B. Speicherverwaltung und Ein-/Ausgabe der Standard-C-Bibliothek `libc`) deutlich zu beobachten ist. Der Aufwand ist aus diesem Grund oft nur „vermeintlich“ gering, da sich notwendige Anpassungen nachträglich als äußerst aufwendig und ggf. nicht machbar herausstellen können.

**from scratch:** Bei der vollständigen Neu-Implementierung (engl.: *from scratch*) ist der Umfang der übernommenen Realisierungen unwesentlich. Die Frage nach der Art der Übernahme erübrigt sich. Der Vorteil dieser Vorgehensweise ist die Möglichkeit, die entworfenen Verfahren mit der dafür erforderlichen Flexibilität und der gewünschten engen Integration realisieren zu können, ohne zusätzliche Randbedingungen beachten zu müssen. Auf der anderen Seite ist der zu erbringende Aufwand enorm. Der Glaube durch vollständige Neuentwicklung Besseres zu entwickeln ist zudem fragwürdig. In der Realität wird auf diese Weise ein Großteil der Energie für die redundante Realisierung bereits existierender Detaillösungen und triviale Datenstrukturen, wie Listen und Hashing, aufgebracht. Zusätzlich besteht große Gefahr, bereits gemachte Fehler zu wiederholen. Der geringe Vorbereitungsaufwand (Einarbeitung, etc.) dieser Vorgehensweise erklärt allerdings, daß sie im Bereich der Betriebssysteme neben der additiven Vorgehensweise am häufigsten angewandt wird. Besonders in Projekten wie Oberon und Plurix [Sch99, WG92], die sich unter dem Stichwort *lean production* bemühen, schlanke Systeme zu konstruieren, wird diese Vorgehensweise bevorzugt. Allerdings bleiben auch dort die Fragen offen, wie Redundanz, das Wiederholen bereits begangener Fehler und das streng monotone Wachstum trotz sukzessiver Hinzunahme zusätzlicher Funktionalität [Fra94b] vermieden werden soll.

**adaptiv:** Eine Alternative zu additiven Erweiterungen oder vollständiger Neu-Implementierung bietet die Modifikation existierender Produkte. Ein auf diese Weise realisiertes System besteht zu einem signifikanten Anteil aus übernommenen Komponenten, die jedoch an die veränderte Umgebung angepaßt wurden. Die Einschränkung der Flexibilität sowie der Integrationsmöglichkeiten ist umgekehrt proportional zu dem Aufwand, der für die Modifikation investiert wird. Die langjährige Dominanz proprietärer und damit nicht modifizierbarer Systeme sowie das Gewöhnung an Neu-Implementierungen trugen bislang dazu bei, daß diese Strategie trotz ihrer Vorteile in Bezug auf Aufwand, Flexibilität und Integrität bislang selten eingesetzt wurde. Der Trend zu Software, die im Quellcode frei verfügbar ist, führt aktuell zu einer deutlichen Veränderung dieser Situation. Das qualitative und quantitative Potential dieser Vorgehensweise kann deutlich am Beispiel das Betriebssystems Linux [Han99] und dessen rapider Entwicklung studiert werden.

**Generative Ansätze**

Eine wirksame Lösung des Aufwandsproblems ohne Einschränkung von Flexibilität und Integrität ist langfristig nur mit generativen Verfahren möglich, die auf ähnliche Weise wie Übersetzergeneratoren im Übersetzerbau die automatisierte Realisierung komplexer Managementsysteme ermöglichen müssen. Ansätze dieser Art existieren im Bereich der Betriebssysteme kaum, da methodische Grundlagen bislang fehlen. Versuche, generativ zu konstruieren, beschränken sich deshalb derzeit auf *komponentenbasierte* Realisierung und eher statische *Spezialisierung* allgemeiner Verfahren.

Im Projekt FLUX OSKit [FBB+97] werden wiederverwendbare Komponenten für die Konstruktion zentraler und verteilter Betriebssysteme auf Basis der x86 Architektur entwickelt, wobei ein Großteil der Komponenten aus der Dekomposition existierender und frei verfügbarer Systeme gewonnen wird. Das OSKit erleichtert die Neukonstruktion von Betriebssystemen sowie der Erweiterung um zusätzliche Fähigkeiten erheblich. Da die kombinierbaren Bausteine allerdings verhältnismäßig grobgranular sind, z. B. vollständiger Scheduler, ist die Flexibilität der Konstruktion stark eingeschränkt. Neu entwickelte Konzepte müssen deshalb nach wie vor überwiegend additiv oder adaptiv realisiert werden, wofür keine systematische oder automatisierte Unterstützung vorgesehen ist. Ferner beschränkt sich die Unterstützung auf den Betriebssystemkern. Abhängigkeiten mit anderen Managementinstanzen finden keine Berücksichtigung.

Der Ansatz der Spezialisierung wurde unter anderem in dem Projekt Choices [CJMR89, CI93] verfolgt. Choices ist ein Rahmen für die objektorientierte Entwicklung von Betriebssystemen in der Sprache C++. Abstrakte Klassen spezifizieren das allgemeine Verhalten bzw. die Signaturen prinzipieller Betriebssystemabstraktionen. Konkrete Klassen verfeinern die geerbten Eigenschaften einer abstrakten Klasse und dienen der Spezifikation konkreter Versionen, Politiken oder Mechanismen. Diese Methode wird unter anderem genutzt, um mit geringem Aufwand eine Ausnahmebehandlung, Dateisysteme und Gerätetreiber [Kou91] zu implementieren. Die Eigenschaften der abstrakten Klassen sind bereits relativ umfangreich. Der verbleibende Spielraum ist zusätzlich durch die Festlegung der Sprache C++ eingeschränkt, so daß die Flexibilität für die Realisierung neuer Komponenten und ihren Abhängigkeiten eingeschränkt ist. Zudem wird die erforderliche additive Erweiterung der Rahmenstruktur, abgesehen von der grundsätzlichen Objektorientierung, abermals nicht systematisch unterstützt. Ähnlich wie FLUX beschränkt sich auch Choices auf den Betriebssystemkern.

Ein geeignetes generatives Verfahren muß in der Lage sein, Entscheidungen der Managementkonstruktion mit unterschiedlicher Detaillierung systematisch zu erfassen und bei deren inkrementellen Konkretisierung bis zu einem ausführbaren Programm – dem Managementprogramm – zu assistieren. Ein Verfahren dieser Art ist bislang allerdings unbekannt.

#### 2.3.3.2 Instrumentierung

Verhältnismäßig wenig Augenmerk wird in der Praxis den Mitteln gewidmet, die zur Realisierung der entworfenen Verwaltungskonzepte und -verfahren eingesetzt werden. Je nach Kontext der Arbeiten und dem Aufwand werden naheliegende Instrumente eingesetzt, wie z. B. Übersetzer im Bereich des Übersetzerbaus und Kerne im Bereich der Betriebssysteme. Allerdings hat auch diese Wahl Folgen für die Fähigkeiten und Eigenschaften des Managements und sollte deshalb gründlich überdacht werden. Bei den existierenden Ausprägungen von Managementinstrumenten ist in aller erster Linie zu beachten, daß es sich um Artefakte handelt, deren Eigenschaften historisch gewachsen sind. Ihr mittlerweile beträchtlicher Umfang forciert sowohl ihre unmodifizierte Übernahme als auch die zunehmende Separation ihrer Fähigkeiten. Um gesamtheitlich integriertes Management durchführen zu können, ist es demgegenüber wichtig, daß die spezifischen Fähigkeiten verschiedener Instrumente wieder zusammengeführt und aufeinander abgestimmt eingesetzt werden.

**Definition 2.20 (Management-Instrument)**
*Ein Management-Instrument ist ein Programm, das bei seiner Ausführung Managemententscheidungen trifft oder Managementmaßnahmen durchführt.*

Im Folgenden werden die Eigenschaften klassischer Instrumentierungen analysiert, um später bei der Realisierung des V-PK-Managements eine adäquate Zuordnung von Managementaufgaben an Managementinstrumente durchführen und ggf. neue Lösungen planen zu können. Den Schwerpunkt der Betrachtung bilden dabei Interpretierer, Übersetzer und Binder auf die sich auch die weiter unten erklärten Realisierungsarbeiten in dieser Arbeit konzentrieren. Die besonderen Eigenschaften von Lader, Laufzeitsystemen und Kern werden beschränkt auf ihre Einordnung in eine gesamtheitliche Managemeninstrumentierung erklärt.

**Interpretierer**
Ungeachtet ihrer engen Verwandtschaft werden Interpretierer üblicherweise scharf von übersetzerbasierten Instrumentierungen getrennt. Beiden Ansätzen kommt die Aufgabe zu, die Bedeutung eines abstrakten Modells, repräsentiert durch ein Wort in einer Quellsprache $q \in Q$, zu erarbeiten [PH94], d. h. $q$ zu interpretieren. In Anlehnung an die Definition der Interpretation applikativer Programme aus [Bro98] sei der Begriff Interpretation hierzu wie folgt definiert:

**Definition 2.21 (Interpretation)**
*Eine Interpretation $i$ ordnet Worten $q \in Q$ bei einer gegebenen Belegung $\beta$ der freien Identifikatoren aus $q$ eine Bedeutung $d \in D$ in dem Raum möglicher Bedeutungen zu.*

$$\begin{aligned} \beta &: ID_q \quad \rightarrow D \\ i &: Q \times \beta \rightarrow D \end{aligned}$$

*Interpreter* oder *Interpretierer* werden in der Regel im Sinne von Definition 2.22 verstanden [Bro98, Dud93], wenngleich dies nicht zwingend aus dem Begriff „Interpretierer“ hervorgeht.

**Definition 2.22 (Interpretierer)**
*Ein Interpretierer $I$ für eine Sprache $Q$ führt Programme der Sprache $Q$ unmittelbar aus.*

Geprägt von den imperativen Sprachen wird die Bedeutung und Wirkung eines Programms in dieser Sichtweise durch die „Ausführung“ der einzelnen Anweisungen in $q \in Q$ erarbeitet. Mit „unmittelbar“ ist dabei gemeint, daß im wesentlichen keine Vorverarbeitung des zu interpretierenden Programmes erfolgt. Selbst verhältnismäßig einfache Programme und Programmteile werden deshalb bei erneuter Nutzung vollständig neu, d. h. wiederholt interpretiert. Dies gilt insbesondere auch für Unterprogramme und Schleifen. Aus der Sicht des Managements bedeutet das, daß bei jeder Ausführung von $q$ und Teilen davon sämtliche Realisierungsentscheidungen neu getroffen werden, da keine Zwischenergebnisse als Nebenprodukte erzeugt, aufbewahrt und bei Bedarf wieder genutzt werden. Dieses repetitive Verhalten ist im Gegensatz zu dem Herausstellen der Interpretation das eigentliche Charakteristikum dieser Art der Instrumentierung. Hinzu kommt, daß das Programm des Interpretierers, wie in Abbildung 2.13 dargestellt, auf dieser Ebene der Betrachtung nicht weiter strukturiert, d. h. ein Monolith ist. Der Nachteil eines Interpretierers ist seine lange Rechenzeit, die sich aus der Repetition ergibt. Speicheradressen von Datenobjekte werden ebenso bei jedem Zugriff neu berechnet, wie invariante Ausdrücke in Schleifen, et cetera. Der Vorteil dieser Vorgehensweise ist die große Flexibilität, die aus der Abwesenheit eines „Gedächtnis“ resultiert. So können in Ausführung befindliche Programme ohne großen Aufwand dynamisch erweitert oder korrigiert werden und Zustände von Komponenten und Berechnungen während der Ausführung mit geringem


Abbildung 2.13: Intepretierer

Aufwand erfragt bzw. modifiziert werden. Die klare Abgrenzung von Interpretierern zu den im Folgenden erklärten Alternativen ist schwierig, da in der Praxis auch Interpretierer entgegen Definition 2.22 durch Vorverarbeitungsstufen und Verwaltung von Zwischenergebnissen optimiert werden.

**Übersetzer**
Klassischerweise werden Übersetzer als Instrument zur Generierung „ausführbarer Programme“ betrachtet [RP97a, Dud93, ASU86, Muc97]. Im Gegensatz zu Interpretierern führen sie Quellprogramme nicht aus, sondern verarbeiten diese zu effizienten Zielprogrammen. In [PH94] sind Übersetzer wie folgt definiert:

**Definition 2.23 (Übersetzer)**
*Seien $Q, \beta, D$ wie in Def. 2.22 festgelegt. Ein Übersetzer*[15] *implementiert eine Abbildung $u$, die aus einem Wort $q \in Q$ eine Funktion generiert, die einer Belegung $\beta$ der freien Identifikatoren aus $q$ die Bedeutung $d \in D$ zuordnet.*

$$u : Q \rightarrow (\beta \rightarrow D)$$

Nach Curry [RP97a] ist das Ergebnis von $u$ äquivalent mit dem Ergebnis von $i$. Der Unterschied zu einem Interpretierer ist die Mehrstufigkeit des Verfahrens zur Interpretation von $q$ mit dem Charakteristikum, daß der Übersetzer eine *einmalige* Verarbeitung des Quellprogramms gegenüber der *Repetition* von Interpretierern vornimmt. Basierend auf dieser Vorverarbeitung folgen in aller Regel die Instrumente *Binder* (B), *Lader* (L), *Laufzeitsystem* (LZ) und *Kern* (K) (s. Abb. 2.14). Das Zielprogramm $z$, das in der Darstellung aus dem Übersetzungs- und Bindevorgang resultiert, konstituiert den $q$-spezialisierten Interpretierer, der seinerseits von $\{L, LZ, K\}$ ausgeführt wird. Zusammen bilden $\{z, L, LZ, K\}$ einen Interpretierer $i_\beta$, der die Sprache aller möglichen Belegungen der freien Identifikatoren aus $q$ versteht, d. h. $i_\beta : \beta \rightarrow D$. Betrachtet man die Zusammensetzung von $i_\beta$, so bildet $\{L, LZ, K\}$ einen unmodifiziert übernommenen, allgemeinen Teil $i_{std}$ und $z$ den dediziert für $q$ generierten Teil $i^q_{gen}$ des Interpreters von $i_\beta = i^q_{gen} \circ i_{std}$.

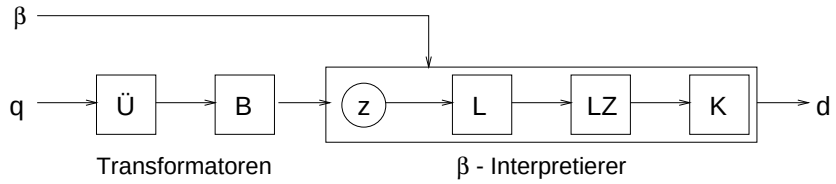


Abbildung 2.14: Transformatorische Instrumentierung

[15] engl.: compiler

Diese unübliche Sichtweise verdeutlicht sowohl den Zusammenhang als auch grundlegende Unterschiede zwischen den Instrumenten aus der Sicht des Managements. Übersetzer und Binder führen ihre Maßnahmen für ein $q \in Q$ einmalig durch. Lader, Laufzeitsystem und Kern führen äquivalent einem Interpretierer das Programm $z$ unmittelbar aus, wobei sie im hohen Maße repetitiv handeln. Aus dieser integrierenden Betrachtung und der Unterscheidung zwischen *Einmaligkeit* und *Repetition* ergibt sich das naheliegende Kriterium, Aufgaben nur dann an repetitiv handelnde Instrumente zu vergeben, wenn sie Veränderungen unterliegen. Alle anderen Aufgaben sollten durch eine einmalige Vorverarbeitung und Erstellung eines geeigneten Zwischenprodukts erfüllt werden. Dabei sei bemerkt, daß die bisher zur deutlichen Abgrenzung genannte „Einmaligkeit“ tatsächlich eher als „selten“ zu verstehen ist.

Diese Vorstellung der Zuordnung von Aufgaben ist allerdings stark idealisiert und ebenso wie die hier dargestellte Auffassung der Bedeutung von Übersetzern. Für gewöhnlich werden Aufgaben an den Übersetzer aus anderen Motiven zugeordnet, die ebenso wie für die anderen Instrumente, oft vor allem aus der Eigendynamik der separaten Fortentwicklung der Instrumente begründet sind. Der überwiegende Anteil an transformatorischen Maßnahmen von einer Quellrepräsentation $q \in Q$ in eine Zielrepräsentation $z \in Z$ motiviert den Begriff „Übersetzer“, dessen transformatorische Aufgaben klassisch wie folgt festgelegt sind:

- Analyse der Syntax und der statischen Semantik von $q$
- Synthese des Zielprogrammes $z$
- Generierung von Zusatzinformation für nachfolgende Managementinstrumente und weitere Entwicklungswerkzeuge zum Zweck der Fehlersuche, Leistungsanalyse, etc.

Für gewöhnlich ist das Abstraktionsniveau von $Q$ höher als das von $Z$. Die von einem Übersetzer implementierte Transformation $u : Q \to Z$ ist in diesem Fall eine Konkretisierung der Konzepte von $Q$ auf das realisierungsnähere Niveau $Z$. Meist ist die Sprache der Hardwareprozessoren eine Teilmenge $Z_p \subset Z$. Wörter $z \in Z$ werden dementsprechend auch *Maschinenprogramme* oder *Binärcodes* genannt.

**Behauptung 2.24 (Optimierung)**
*Eine Optimierung eines Verfahrens ist seine partielle Verbesserung.*

Zur Konstruktion einer geeigneten Abbildung von $Q$ nach $Z$ wird zunächst ein einfaches und semantisch korrektes Abbildungsverfahren entwickelt. Ausgehend von dieser Standardlösung wird anschließend sukzessive versucht, durch Erweiterung und Modifikation Verbesserungen der Abbildung zu erzielen, d. h. diese zu **optimieren**. Optimierung in Übersetzern subsumiert eine Vielzahl von Maßnahmen, die dazu dienen, innerhalb der fixierten Umgebung von Quellprogramm, Übersetzer und Zielsprache, das Laufzeitverhalten und den Speicherplatzbedarf der erzeugten Zielprogramme zu verbessern. Trotz des engen Spielraums sind Optimierungsschritte für die Effizienz essentiell. Bereits einfache Maßnahmen, wie das Entfernen gemeinsamer Teilausdrücke und Permutationen von Anweisungen, können zur Halbierung der Ausführungszeiten des Zielprogrammes führen. Abbildung 2.15 ist aus [Muc97] entnommen und stellt die Vielfalt der Optimierungsschritte dar, die in modernen Übersetzern für sequentielle, lokale Systeme eingesetzt werden. In parallelen und verteilten Systemen müssen einige dieser Verfahren angepaßt werden [Kno98] und es kommen zahlreich neue hinzu [KSV96]. Einige der Optimierungen gehen über das Transformieren des Quellprogramms hinaus. De facto erfolgt in Maßnahmen, wie der Faltung konstanter Ausdrücke (z. B. $5 + 3 \mapsto 8$), eine partielle

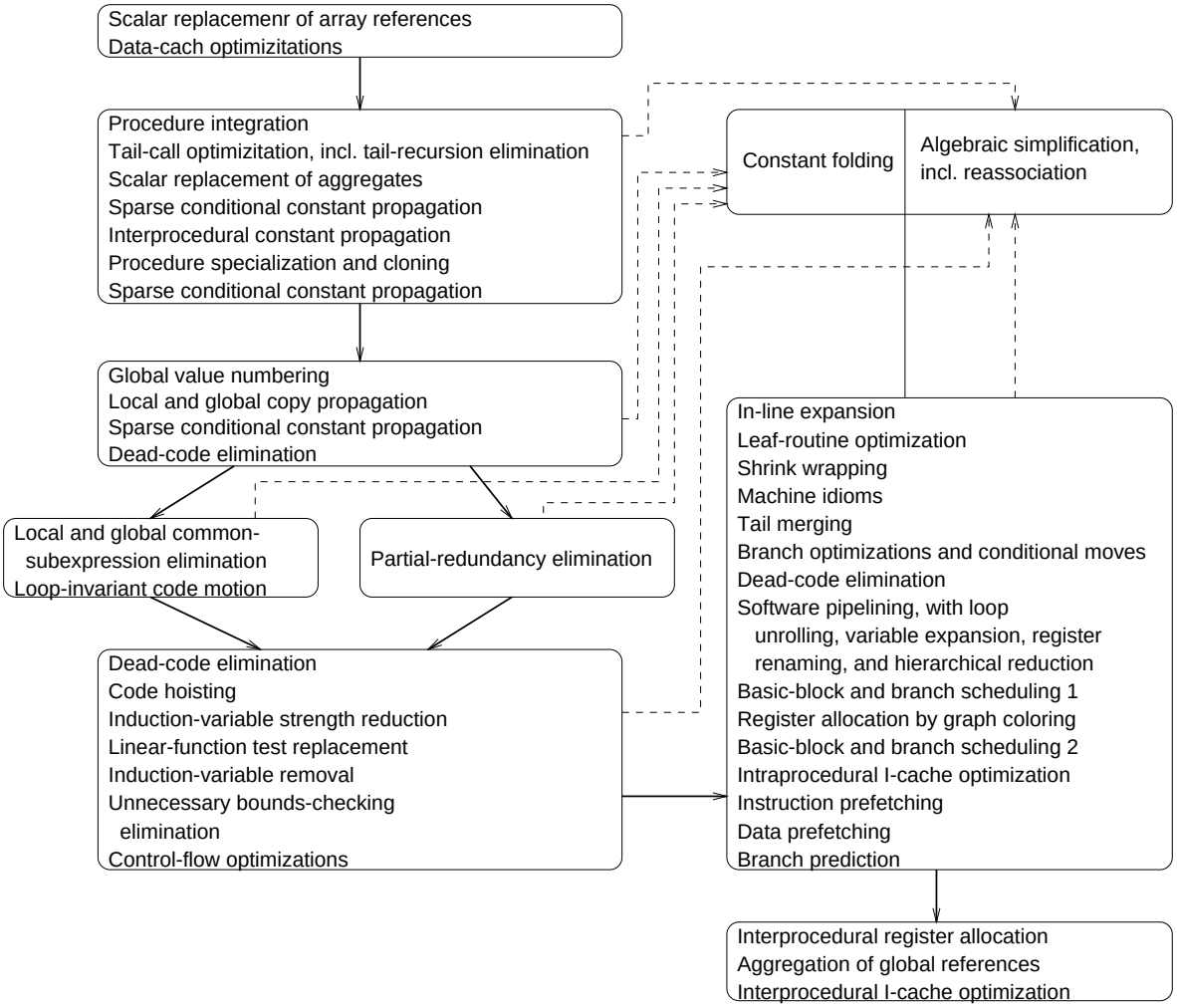


Abbildung 2.15: Übersetzer-Optimierung in sequentiellen Systemen

Ausführung, d. h. Interpretation des Quellprogramms. Der Übergang von Übersetzern zu Interpretierern ist somit auch aus dieser Perspektive fließend.
Ein großer Teil der optimierenden Maßnahmen stützt sich auf **Information**, die gegliedert in *Daten-* und *Kontrollflußanalysen* aus dem Programmtext gewonnen wird. Übersetzer nutzen dabei ihre Nähe zum Quellprogramm, die sie gegenüber den anderen klassischen Instrumenten, Binder, Laufzeitsystem, etc. auszeichnet. Information, die auf diese Weise entkoppelt von der Ausführung des Programms gewonnen wird, beschreibt das Entwicklungspotential der in $q$ formulierten Rechenvorschrift und wird deshalb auch *statisch* genannt. Typisch für existierende übersetzerbasierte Instrumentierungen ist, daß die gewonnene Information ausschließlich von dem Übersetzer genutzt und nach dessen Beendigung vernichtet wird. Der aus Sicht der Integration wünschenswerte Informationstransfer zu den nachfolgenden Instrumenten findet in der Regel nicht statt, da sich der Zuständigkeitsbereich des Übersetzers ausschließlich auf seine eigenen, optimierende Maßnahmen beschränkt. In parallelen und verteilten Systemen wird dieses Defizit wegen der größeren Sensibilität gegenüber inadäquaten Manage-

mententscheidungen wesentlich stärker spürbar. Information, die für das verteilte Management benötigt werden würde, wie z. B. Kooperationsabhängigkeiten, kann nachträglich, wenn überhaupt, nur mit zusätzlichem Monitoring-Aufwand ermittelt werden, falls der Übersetzer entsprechende Analysen nicht durchführt oder diese Information vernichtet. In verschiedenen Projekten wird deshalb versucht, den Übersetzer durch Erweiterung seines Zuständigkeitsbereichs enger in das Gesamtmanagement zu integrieren. Dabei wird zusätzliche Information aus dem Quellprogramm analysiert, an die folgenden Instrumente weitergereicht und Maßnahmen, wie die Plazierung von Datenobjekten auf Stellen, bereits durch den Übersetzer vorbereitet [DLC+99, CV95, ID98, BK93].

Die klassische Auffassung von Übersetzern geht ferner davon aus, daß der Übersetzer einmalig nach erfolgter Spezifikation von $q$ und lange vor dessen Ausführung seine Arbeit verrichtet. Eine Veränderung dieser Festlegung ermöglicht den flexibleren Einsatz des Übersetzers. Daraus ergibt sich neues Potential für Leistungssteigerung, dessen Nutzung in verschiedenen Ansätzen bereits angestrebt wird. Begriffe die in diesem Zusammenhang stehen sind:

**Inkrementelle Übersetzung:** Mit der inkrementellen Übersetzung wird versucht, die Vorteile gewöhnlicher Interpretierer und Übersetzer zu vereinen. Anstelle der Verarbeitung vollständiger Programme nach ihrer Spezifikation wird das Programms während seiner Spezifikation übersetzt. Kleine Veränderungen an $q$ führen im Idealfall zu kleinen Veränderungen an $z$. Durch diese Vorgehensweise entstehen neue Freiheitsgrade, die sowohl für den Programmierer als auch das Management interessant sind. Der Übergang von der Spezifikation zur Ausführung verkürzt sich, vollständige Programmfragmente können früh einmalig ausgewertet werden und das Zielprogramm kann schneller an veränderte Anforderungen angepaßt werden. Allerdings steht diese Vorstellung im Konflikt mit Eigenschaften der Grammatik sowie wichtigen Analysen und Optimierungsentscheidungen, deren Kontext meist nicht mit der gewünschten Granularität individuell übersetzbarer Programmfragmente übereinstimmt.

**Code-Generation On-the-Fly:** Bei diesem Verfahren wird das Quellprogramm gemäß dem Prinzip der *lazy evaluation* erst unmittelbar vor seiner Ausführung während des Ladens in den Hauptspeicher übersetzt [Fra94a]. In [FK97] wird am Beispiel von Oberon demonstriert, daß sich die Lade- und Startzeiten dadurch nicht verlängern müssen. Zu den Vorteilen gehören erhöhte Portabilität, signifikante Verringerung der Datenmenge bei Übermittlung und Speicherung von Programmen, Erweiterbarkeit bis zum Zeitpunkt der Ausführung sowie Ausweitung der Optimierung auf inter-modulare Beziehungen. Mit der Intensität der Analysen und Optimierungsschritte steigen allerdings auch die Ladezeiten, die bei jedem Programmstart wiederholt zu investieren sind. Im Vergleich zur obigen Diskussion von Interpretierer und Übersetzer wäre es auch hier hilfreich das Ergebnis vorhergehender Übersetzungs-/Ladevorgänge als Zwischenprodukt aufzubewahren, um repetitives Verhalten bei invarianten Anforderungen zu minimieren. Ferner hemmt die restriktive Haltung gegenüber der Veröffentlichung von Programmen in Quellrepräsentation bislang den Einsatz dieses Verfahrens.

**Run-Time Code Generation (RTCG):** Die Erzeugung des Maschinenprogramms findet während seiner Ausführung statt und besitzt Überlappungen mit der Ausführung des Programms. Dieser Ansatz ermöglicht es, Entscheidungen des Übersetzers so lange zu verzögern, bis die gewünschte Information vorliegt. Das Spektrum der Zielcodeerzeugung zur Laufzeit reicht von der verzögerten Generierung und Modifikation von Frag-

menten des Zielprogrammes bis zur vollständigen Neu-Übersetzung des Quellprogramms und Ersetzung des Zielprogramms während seiner Ausführung. Das Potential für Leistungssteigerungen ist dementsprechend groß. Im Umfeld objektorientierter Sprachen, z. B. SELF, wird dieser Ansatz genutzt, um Nutzungen dynamisch typisierter Objekte mit Hilfe von Laufzeitinformation (*dynamische* Information) zu optimieren. In [HA95] wird demonstriert, daß eine solche Integration statischer und dynamischer Analysen und Maßnahmen spürbare Verkürzungen der Programmlaufzeiten ermöglicht. In [Kis96] wird RTCG erfolgreich eingesetzt, um erweiterbare Programme inkrementell während ihrer Ausführung zu optimieren. Dabei wird die Verzögerung der Optimierung genutzt, um gezielt diejenigen Programmfragmente zu fokussieren, die gemäß ihrem dynamischen Ausführungsprofil besonders viel Rechenzeit in Anspruch nehmen. Eine weitere Einsatzmöglichkeit ist das Editieren in Ausführung befindlicher Programme zur Beseitigung von Fehlern [RBDL97].
Limitiert wird das Potential dieser Vorgehensweise allerdings von der Granularität der Übersetzungseinheiten, den Fähigkeiten des Binders und Konsistenzproblemen bei der Substitution von Realisierungen, die evtl. in Verwendung sind.

Diese Ansätze haben, mit Ausnahme der inkrementellen Übersetzung, allerdings gemein, daß die Integration von Fähigkeiten des Übersetzers mit anderen Managementinstrumenten nicht systematisch sondern phänomenologisch anhand konkreter Problemfälle betrieben wird.

**Binder**
Der Binder ist für diese Arbeit aus zwei Gründen von besonderem Interesse:

1. Am Beispiel des Zusammenwirkens von Übersetzer und Binder kann die Systematik der Produktion und flexiblen Komposition von Ressourcen exemplarisch studiert werden. Dabei wird insbesondere die Bedeutung der Granularität von Ressourcen und ihre Bindung über einen festgelegten Zeitraum sehr deutlich.

2. Die Fähigkeiten des relativ einfachen Instrumentes Binder definieren sowohl den Rahmen für die Integration des Übersetzers mit laufzeitbegleitenden Maßnahmen als auch die dynamische Erweiterbarkeit des in Ausführung befindlichen Systems.

Abgesehen von den beschriebenen Ansätzen zur verzögerten bzw. flexibilisierten Ausführung des Übersetzers, schließt sich an den Übersetzungsvorgang gewöhnlich die Tätigkeit des Instrumentes Binder an, der selbst ein Übersetzer mit einer Quellsprache – der Bindersprache – und einer Zielsprache – meist die Quellsprache des Laders – ist.

**Definition 2.25 (Binder)**
*Sei $U$ die Zielsprache des Übersetzers, $\mathfrak{P}(U)$ die Potenzmenge von $U$ und $Z$ die Zielsprache des Binders. Das Instrument Binder implementiert die Abbildung b:*

$$b : \mathfrak{P}(U) \rightarrow Z$$

*In $z = b(u)$ sind die freien Variablen $V_u$ mit Adressen von Bausteinen $u_i \in u$ belegt.*

Der Binder verarbeitet eine Menge von Bausteinen $u_i \in \mathfrak{P}(U)$, die i. d. R. Produkte des Übersetzers sind. Die freien Variablen $V_u$ der Bausteine sind Identifikatoren für Elemente aus $u$, die nicht von den Prozessoren der Zielmaschine interpretiert werden können und deshalb *offene Referenzen* oder *Relokationen* genannt werden. Der Binder ersetzt die offenen Referenzen

mit Adressen referenzierter Bausteine, wobei die eingesetzten Adressen Worte aus der Sprache der Prozessoren sind. Dieser Vorgang wird das *Auflösen von Referenzen* bzw. *Relokieren* genannt. Mathematisch entspricht es dem Binden freier Variablen. Das Resultat des Bindervorgangs ist ein Programm, dessen Abstraktionsniveau durch Transformation der Referenzen in die Sprache der Prozessoren für die Interpretation durch die Prozessoren vorbereitet ist. Nach wie vor gilt allerdings $Z_p \subseteq Z$; das Ergebnis des Binders ist nicht zwangsläufig ein maschinen-interpretierbares Programm, da es noch zusätzliche Information beinhalten kann. Die ursprüngliche Motivation für den Bindevorgang ist die flexible Wiederverwendung von Produkten des Übersetzers, unabhängig von der Quellrepräsentation, inklusive der Möglichkeit, komplexe Programme ähnlich der inkrementellen Übersetzung in separat übersetzbare Einheiten partitionieren zu können. Entscheidend für die erzielbare Flexibilität ist die minimale *Granularität* der Binder-Bausteine. In imperativen Programmiersprachen entspricht diese meist der Spezifikation von Unterprogrammen der Schachtelungsstufe 0 (nicht geschachtelt).

Die Rolle des Binders für die Gewinnung von *Managementinformation* ist, abgesehen von der Filterung von Zusatzinformation aus den Übersetzerprodukten, im allgemeinen eher gering. Typisch für die insgesamt schwache Integration der Managementinstrumente in herkömmlichen Systemen wird die Information über offene Referenzen und ihre Auflösung ebenso wie im Falle des Übersetzers nach Beendigung der Arbeit des Binders mit wenigen Ausnahmen [NH94] vernichtet. Entscheidungen des Binders können später nicht revidiert werden und bei erneuter Bindung eines Programms muß die gesamte Relokations-Information neu analysiert werden. Hierzu sei bemerkt, daß die Anzahl durchzuführender Relokationen bei gewöhnlichen Anwendungsprogrammen auf einer UNIX-Plattform etwa in der Größenordnung $10^5$ liegen [NH94].

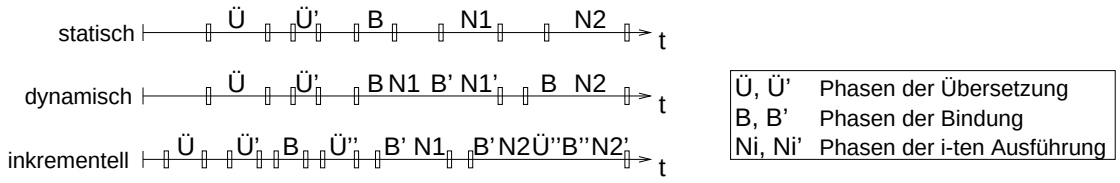


Abbildung 2.16: Statische, dynamische und inkrementelle Bindung

Das verhältnismäßig simple Grundprinzip des Binders wurde in einigen Arbeiten um Fähigkeiten weiterentwickelt, die Bezüge zur Programmentwicklung oder dem Laufzeitmanagement intensivieren und damit eine engere Integration in das Gesamtmanagement erzielen. Ein *statischer* Binder verrichtet seine Arbeit nach der Produktion aller Bausteine durch den Übersetzer einmalig vor der ersten Ausführung des Zielprogramms so, daß nach seiner Beendigung alle offenen Referenzen aufgelöst sind. Das Zielprogramm kann anschließend beliebig oft ohne zusätzliche Binder-Maßnahmen ausgeführt werden (s. Abb. 2.16 oben).

Relokiert der Binder zumindest einen Teil der offenen Referenzen erst unmittelbar vor jeder Ausführung des Programms oder währenddessen, so spricht man von *dynamischem* Binden, und das Ausgangsprodukt für den dynamischen Teil des Binders wird *dynamisch gebunden* genannt (Abb. 2.16 Mitte). Falls zusätzlich ein Teil der Relokationen erst unmittelbar bei Zugriff auf den Baustein zur Laufzeit aufgelöst wird, wird das Verfahren ferner als *lazy* (faul) bezeichnet. Faules Binden ermöglicht eine Einsparung der insgesamt durchzuführenden Relokationen. Dynamisches Binden war ursprünglich bereits Teil des Multics

Betriebssystems [Org72], wurde aus den Nachfolgesystemen jedoch wieder entfernt, da die damalige Instrumentierung mittels Kern-Einsprünge lange Rechenzeiten erforderte. Später wurde diese Idee in UNIX System V wieder aufgegriffen [Arn86] und erfolgreich außerhalb des Kerns realisiert. Variationen davon wurden anschließend auch auf anderen Plattformen, wie DLL[16] [JS89] in MS Windows, realisiert. Dynamisches Binden bietet eine Reihe von Vorteilen, die auf den flexibleren Umgang mit den Übersetzerprodukten zurückzuführen sind. Der Speicherbedarf dynamisch gebundener Programme ist, sofern sie sich nicht in Ausführung befinden, erheblich geringer als bei statischer Bindung, da das redundante Kopieren gemeinsamer Komponenten in die Zielprogramme entfällt. Ferner werden erst unmittelbar vor oder während der Ausführung des Programms die Eigenschaften der noch nicht gebundenen Bausteine festgelegt. Dies ermöglicht umfangreiche Anpassungen an die aktuelle Situation der ausführenden Zielmaschine.

Häufig verwechselt wird das dynamische Binden mit *shared libraries* (SL) bzw. *gemeinsam benutzbaren Bibliotheken*, obwohl es sich dabei um ein wohl zu unterscheidendes Konzept handelt. Eine SL ist ein Baustein, der von dem Übersetzer oder durch Vorverarbeitung des Binders erzeugt wurde und von dem Binder durch entsprechende Auflösung von Referenzen gleichzeitig an verschiedene Zielprogramme gebunden ist, d. h. sei $z_1 = b(u_1), z_2 = b(u_2)$, dann ist $sh = z_1 \cap z_2, sh \neq \emptyset$ die Menge der von $z1$ und $z2$ gemeinsam gebundenen SLs. Diese Technik erlaubt es, den Bedarf an Arbeitsspeicher während der Ausführung von Programmen durch das Eliminieren redundanter Kopie zu reduzieren. Falls die Ausführungen von Programmen in separaten Adreßräumen stattfinden, bedarf dieses Verfahren der Vorbereitung durch den Übersetzer, der verschiebbaren Zielcode produzieren muß (PIC[17]). Interessant ist dabei zu beobachten, daß dieser Vorteil durch eine Kombination von Maßnahmen in dem Übersetzer und dem Binder möglich wird. Die ursprüngliche angestrebte Orthogonalität der Maßnahmen dieser beiden Instrumente geht dabei verloren, was jedoch eine untergeordnete Rolle spielt.

Noch allgemeiner als die dynamische Bindung ist die *inkrementelle Bindung* (Abb. 2.16 unten), bei der die Ausführung des Binders mit der Ausführung des Übersetzers verschränkt ist. Je nachdem, ob sich das Zielprogramm bzw. Fragmente davon bereits in Ausführung befinden können, ist weiter zwischen *statisch inkrementell* und *dynamisch inkrementell* zu unterscheiden. Die Flexibilität der dynamisch inkrementellen Bindung ist Voraussetzung für die inkrementelle Erweiterbarkeit in Ausführung befindlicher Programme. Allerdings wird zu diesem Zweck keine SL-Technik benötigt.

Der Einsatz des Binders zur Unterstützung des Managements in parallelen und verteilten Umgebungen wurde bislang kaum untersucht.

**Lader, Laufzeitsystem und Kern**

Die verbleibenden Instrumente Lader, Laufzeitsystem und Kern spielen in dieser Arbeit, die sich in Bezug auf die Realisierung des V-PK-Managements auf transformatorische Maßnahmen konzentriert, eine untergeordnete Rolle. Im folgenden werden deshalb nur die Eigenschaften dieser Instrumente erklärt, die zu ihrer Abgrenzung benötigt werden.

An der Schnittstelle des Binders zu dem **Lader** manifestiert sich der Übergang von einmaligen, beziehungsweise, wie oben diskutiert, seltenen transformatorischen zu repetitiv interpretierenden Maßnahmen. Dies entspricht dem Übergang von der Entwicklung des Programms zu

[16] Dynamic Link Libraries
[17] Position Independent Code

seiner Interpretation, wobei die oben besprochenen verzögerten Übersetzungs- und Bindeverfahren diese Trennlinie selbstverständlich relativieren. Die Aufgabe des Laders ist meist primitiv und oft als Teil des Kerns oder Binders implementiert. Dem Lader wird ein Speicherbereich mitgeteilt, in dem ein Maschinenprogramm zur Ausführung durch den Prozessor abgelegt werden soll, woraufhin er das Programm in den Arbeitsspeicher transferiert und bei Bedarf enthaltene Identifikatoren in Adressen aus dem mitgeteilten Speicherbereich transformiert. Das Resultat ist im Standardfall ein vollständig prozessor-interpretierbares Programm. Der flexible Transfer des Binderproduktes zwischen Hintergrund- und Arbeitsspeicher ermöglicht es, Programme, deren Größe die Mächtigkeit des Adreßraumes der Prozessoren übersteigen, dennoch auszuführen.

Vor der Verfügbarkeit großer virtueller Adreßräume kam dieser Technik besondere Bedeutung zu. Das Konzept der *Overlays* [Flo89] ermöglichte die explizite Partitionierung von Programmen in Einheiten, die kleiner gleich dem Adreßraum der Prozessoren waren und zur Laufzeit von dem Lader je nach Bedarf dynamisch ausgetauscht wurden. Mit der Verfügbarkeit großer, 32 Bit weiter, virtueller Adreßräume hat sich die Granularität der Ladeeinheiten verändert. Es werden nicht mehr Fragmente von Produkten des Binders, sondern vollständige Binderprodukte oder vielfache davon geladen. Während der Ausführung findet kein weiteres Laden und Entladen statt. Nach wie vor ist dies notwendig, da der Adreßraum der Prozessoren nicht groß genug ist, um alle Binderprodukte gleichzeitig aufzunehmen.

Durch die Verfügbarkeit noch größerer Adreßräume mit $2^{64}-1$ Identifikatoren ändert sich das. Da in diesem Falle die Aufgabe der Adreßtransformation entfällt, degeneriert die Aufgabe des Laders zum Transfer zwischen Hintergrund- und Arbeitsspeicher, das ebenso von dem allgemeineren Seitenaustauschverfahren des Managementsystems erledigt werden kann. Die Rolle des Laders in verteilten Umgebungen hängt ähnlich von der Frage nach privaten oder gemeinsamen Adreßräumen und ihrer Größe ab. In Bezug auf verteilte Umgebungen gibt es bislang kaum Arbeiten, die sich mit Problemen des Transfers von Maschinenprogrammen zwischen Stellen beschäftigen [Com92]. Da verteiltes Rechnen ohnehin meist als Ausnahmefall betrachtet wird, wird das Maschinenprogramm in der Regel vollständig und redundant auf alle beteiligten Stellen kopiert. Soll verteiltes Rechnen jedoch als Standardfall betrieben werden, so ist der Mehrverbrauch an Netzbandbreiten und Arbeitsspeicher für kopierte aber nicht benötigte Programmfragmente ebenso wenig akzeptabel, wie die fehlende Gewährleistung der Konsistenz der Kopien. Ein verteilter Lader muß dann die Distribution adäquater Einheiten inklusive dem Einsatz leistungssteigernder und konsistenter Replikation von Programmfragmenten, auf die nahezu ausschließlich lesend zugegriffen wird, organisieren.

Den Erklärungen der Eigenschaften von Laufzeitsystemen wird der **Kern** vorgezogen, da seine Merkmale schärfer abgegrenzt werden können. In der Literatur werden Kerne immer stärker mit dem Begriff *Betriebssystem* gleichgesetzt. Diese Begriffsbildung ist äußerst kontraproduktiv. Das Instrument Kern erhält seine Bedeutung ebenso wie die anderen Instrumente ausschließlich durch seine Integration in das Gesamtmanagement, dessen Komponenten die mögliche Nutzung und den Betrieb eines Rechensystems gemeinsam bestimmen.

Eine mögliche Definition für Kerne basiert auf der Unterscheidung zwischen privilegiertem und nichtprivilegiertem Modus der Hardware. Der Kern ist das Programm eines Rechensystems, das exklusiv im *privilegierten Modus* ausgeführt wird und somit als einziges Programm uneingeschränkten Zugriff auf alle Ressourcen besitzt. Diese Betrachtung führt zu zwei Kategorien von Aufgaben, die einem Kern zuzuordnen sind. Zum einen ist die Unterscheidung der Privilegierung seitens der Hardware ein Ankerpunkt, um Separation nichtprivilegierter Be-

rechnungen durchzusetzen. Der Kern bietet hierzu separate Wirkungsbereiche an und gewährleistet deren Trennung durch Nutzung von Kontrollmechanismen der Hardware. Neben der horizontalen Kontrolle von Wechselwirkungen zwischen nichtprivilegierten Berechnung koordiniert und kontrolliert der Kern zusätzlich vertikale Zugriffe dieser Berechnungen auf die physischen Geräte. In den meisten Systemen erweist sich diese Sichtweise von Kernen als Kontrollinstanz für Wirkungsbereiche als tragfähig und nützlich. Zu beachten ist dabei allerdings, daß die Hardware meist mit einem Signal bzw. *trap*, nur eine sehr schmalbandige Schnittstelle für die Kooperation von nichtprivilegierten Berechnungen mit dem Kern vorsieht. Sowohl die Nutzung physischer Ressourcen als auch die kontrollierte Kooperation zwischen nichtprivilegierten Berechnungen wird dadurch sehr aufwendig. Dementsprechend unterliegt auch die Integration des Kerns in das Management Beschränkungen, die vor allem den Informationsaustausch stark limitieren. In den Kern werden deshalb oft zusätzliche Fähigkeiten integriert, die nicht unmittelbar zur Separation von Berechnungen und Kontrolle von Zugriffen auf die Hardware benötigt werden, sondern aus Effizienzgründen nur unbefriedigend außerhalb des Kerns realisiert werden können.

Eine andere mögliche Definition des Instruments Kern ergibt sich aus der Definition des $\beta$-Interpretierers $i_\beta$ auf Seite 60, in welcher der Kern als Komponente des unmodifiziert übernommenen Teilinterpretierers $i_{std}$ angegeben wurde. Das Charakteristikum des Kerns ist, daß er stets Teil von $i_{std}$ ist und damit inhärent zu jedem Interpretierer $i_\beta$ für Programme $q \in Q$ gehört und die Ausführungen der Restinterpretierer koordiniert. Im Gegensatz zu allen anderen Instrumenten existiert pro Stelle genau ein Kern. Neben den Prozessoren sind Kerne die einzige Instanz mit dieser Eigenschaft. Der Kern hebt das Abstraktionsniveau der Hardware auf das einer höherwertigeren virtuellen Maschine an und ermöglicht es, Redundanzen in den dediziert zu generierenden Teilinterpretierern $i^q_{gen}$ zu vermeiden. Aus der Eigenschaft, die gemeinsame Basis für alle Ausführungen zu sein, resultieren besonders hohe Listungsanforderungen. So erhöht jeder Aufwand, den der Kern erzeugt, unausweichlich den in 2.2.2 erläuterten konstanten stellenlokalen Mehraufwand $l$ für alle Berechnungen. Je nach Wichtigkeit der Effizienz werden aus diesem Grund, ähnlich wie oben, zusätzliche Fähigkeiten in den Kern aufgenommen, auch wenn sie nur von wenigen Interpretierern benötigt wird.

Die beiden angegebenen Definitionen sollten einander ergänzend betrachtet werden. Dies trifft insbesondere auf die Rolle des Kerns innerhalb des V-PK-Managements zu, für den eine sinnvolle Balance zwischen der Effizienz der Interpretation, Vermeidung von Redundanz und Kontrolle von Wirkungsbereichen gefunden werden muß.

Das **Laufzeitsystem** hatte historisch die Bedeutung, die Ausführung eines übersetzten Programms auf einer Hardwarekonfiguration zu unterstützen. Das bedeutet, dem Produkt des Übersetzers wurde während seiner Ausführung (der Laufzeit) auf einer Rechenanlage eine Menge vorbereiteter Komponenten beigestellt, die auf der einen Seite die Fähigkeiten der Hardware und auf der anderen Seite die der Sprache erweitern bzw. diese realisieren, falls dies nicht durch transformatorische Maßnahmen des Übersetzers geschieht. Mit der Weiterentwicklung von Betriebssystemen und insbesondere der Entwicklung von Kernen wurde die Einordnung bzw. Abgrenzung von Laufzeitsystemen zunehmend diffuser. Da Kerne eigentlich nach der ursprünglichen Auffassung dem Laufzeitsystem zuzurechnen sind, der Begriff Laufzeitsystem dann aber keine Differenzierung ergibt, hat sich seine Verwendung uneinheitlich entwickelt.

In der vorliegenden Arbeit bezeichnet der Begriff Laufzeitsystem die Menge vordefinierter Komponenten $L_z$, die für die Ausführung des Maschinenprogramms $z$ eingesetzt werden und nicht Teil eines der anderen Instrumente oder des Zielprogrammes $z$ selbst sind. $L_z$ be-

steht überwiegend aus Komponenten, die von dem Binder je nach Bedarf aneinander und an $z$ gebunden werden. Im Gegensatz zum Kern, der ebenso wie das Laufzeitsystem zu $i_{std}$ gehört, können sich Laufzeitsysteme unterschiedlicher Programme unterscheiden, da sie sehr flexibel je nach Bedarf durch dynamische Kombination benötigter Komponenten konstruiert und erweitert werden können. Komponenten des Laufzeitsystems können mit Hilfe des Binders sehr eng mit dem zu interpretierenden Maschinenprogramm verwoben werden, was intensive Informationsflüsse und flexible Abstimmung von Maßnahmen zwischen Programm und Laufzeitsystem während seiner Ausführung erlaubt. Darüber hinaus erfolgt allerdings in aller Regel weder eine Abstimmung mit Optimierungen des Übersetzers noch mit den ohnehin relativ starren Fähigkeiten des Kerns. Information, die das Laufzeitsystem während der Ausführung eines Programms über dieses sammelt, wird mit wenigen Ausnahmen analog zu Übersetzer und Binder nach Beendigung der Ausführung vernichtet.

Laufzeitsysteme sind ein beliebtes Mittel, um neue Konzepte zu implementieren, da sie mit geringem Aufwand und vermeintlich ohne vollständiges Verständnis der angrenzenden Instrumente Übersetzer und Kern zu bewerkstelligen sind. Diesem zugkräftigen Vorteil werden auch in verteilten Umgebungen viele andere Argumente bewußt oder unbewußt untergeordnet. In der Realität führt dies zur Anhäufung von Komponenten im Laufzeitsystem, die ungenügend mit Übersetzer, Kern und auch wechselseitig integriert sind und Anforderungen realisieren, die faktisch wesentlich besser transformatorisch anstelle repetitiv im Übersetzer oder wegen engem Bezug zur Hardware besser im Kern realisiert werden sollten. Besonders für das letztere gibt es zahlreiche Beispiele aus dem Bereich DSM. In dem Laufzeitsystem werden erneut aufwendige Signalbehandlungen durchgeführt und separat Seitentabellen verwaltet, obwohl dies bereits im großen Umfang und wesentlich leistungsfähiger durch den Kern erledigt wird.

Besonders zu beachten ist, daß die Maßnahmen des Laufzeitsystems einerseits repetitiven Charakter besitzen, andererseits das Instrument Laufzeitsystem es ermöglicht, Redundanzen in den dedizierten Teilinterpretierern $i_{gen}$ zu vermeiden, wobei gegenüber dem Kern eine wesentlich stärkere Bindung an $i_{gen}$ bei größerer Flexibilität erreicht werden kann.

## 2.4 Erweiterbarkeit von Systemen

Zahlreiche Veröffentlichungen und Tagungsbände [BH96, MDPP, SS96, SESS96, Con96, CL95] belegen, daß das Thema der Erweiterbarkeit umfangreich und das Spektrum der Möglichkeiten und Probleme vielfältig ist. In der vorliegenden Arbeit spielt der Ansatz eines erweiterbaren Gesamtsystems eine wichtige Rolle für die Informationsgewinnung des Managements. In Anbetracht des Schwerpunktes dieser Arbeit – Management-Konstruktion – genügt es hier jedoch, anstelle einer umfassenden Analyse von Eigenschaften erweiterbarer Systeme, ein Klassifikationsschema anzugeben, mit dem die weiter unten erklärten Konzepte der inkrementellen Erweiterbarkeit des INSEL-Gesamtsystems relativ zu anderen Ansätzen eingeordnet werden können.

**Definition 2.26 (Erweiterbar)**
*Ein System heißt erweiterbar, wenn neue Komponenten konsistent hinzugefügt werden können und bestehende Komponenten dafür zuvor nicht verändert werden müssen.*

Erweiterbarkeit von Systemen befaßt sich generell mit der Modifikation bestehender Systeme zum Zweck der Anpassung an veränderte Anforderungen. Hierzu werden bestehende Freiheitsgrade genutzt oder als Vorbereitung für künftige Erweiterungsschritte bei Bedarf neue

Freiheitsgrade in das bestehende System integriert. Die Wahl des Begriffs *Erweiterung* betont, daß es sich bei den Modifikationen überwiegend um die Hinzunahme neuer Komponenten und Fähigkeiten handelt. Das ebenso mögliche Entfernen nicht mehr benötigter Komponenten aus einem System sei dem Begriff der Erweiterung untergeordnet.

Im Gegensatz zu den vielfältigen Möglichkeiten der Klassifikation verteilter Architekturen und Managementsysteme wird bei der Diskussion der Erweiterbarkeit von Systemen bislang keine einheitliche Terminologie verwendet, wodurch die Abgrenzung verschiedener Ansätze schwierig ist. Zur Strukturierung der verschiedenen Möglichkeiten erweist es sich als zweckmäßig, folgende drei Dimensionen zu unterscheiden:

**Dimension 1 – Zeit:** Generell ist bei der Erweiterung eines bestehenden Systems zu unterscheiden, wann dieses relativ zu seiner Nutzung bzw. Ausführung geschieht. Rudimentär sind *statische*, *dynamische* und *inkrementelle* Erweiterungen voneinander zu trennen. Statische Erweiterungen eines Systems $\mathcal{S}$ sind wechselseitig ausgeschlossen mit der Ausführung von $\mathcal{S}$, d. h. finden vor oder nach $\mathcal{S}$-Ausführungen statt. Dynamische werden während seiner Ausführung vorgenommen. Inkrementelle Erweiterungen erfolgen sukzessive in separaten Schritten. Da inkrementelles Vorgehen entkoppelt von der Nutzung des Systems wenig neue Möglichkeiten gegenüber statischer Erweiterung bietet, ist mit inkrementeller -, falls nicht ausdrücklich anders vereinbart, stets *dynamisch inkrementelles* Vorgehen gemeint – das System wird während seiner Nutzung wiederholt erweitert.

**Dimension 2 – Abstraktionsniveau:** Die zweite Dimension betrifft die Frage nach dem Abstraktionsniveau auf dem die erweiternden Maßnahmen stattfinden. Erweiterungen können unter anderem die Sprache in der das System spezifiziert ist, seine abstrakte Spezifikation, den Übersetzer oder die Hardwarekonfiguration betreffen.

**Dimension 3 – Raum:** Komponenten innerhalb einer Abstraktionsebene des Systems sind Instanzen von Komponentenklassen mit gemeinsamen Eigenschaften. Da meist Erweiterbarkeit auf Instanzen bestimmter Klassen beschränkt ist, wie z. B. Subtyping von Record-Typen in Ada9X, lassen sich verschiedene Ansätze deutlich anhand der Klassen erweiterbarer Komponenten und ihrer Eigenschaften unterscheiden. Hinzu kommt die Frage nach der minimalen Granularität der erweiterbaren Einheiten, bei der es sich von einzelnen Instanzen bis hin zu Mengen abhängiger Instanzen unterschiedlicher Klassen handeln kann.

Um Systeme an neue Anforderungen in sich ändernden Umgebungen anpassen zu können, ist ein Maximum an Flexibilität der Erweiterbarkeit wünschenswert. Inkrementelle Erweiterung auf unterschiedlichen Abstraktionsebenen und nahezu beliebiger Komponenten ist die Voraussetzung für eine Evolution langlebiger Systeme, die es ermöglichen könnte, Redundanzen und wiederholte Fehler zu vermeiden und somit die Qualität von Systemen bei gleichzeitig geringerem Aufwand zu steigern. In verteilten Systemen kommt dieser Langlebigkeit besondere Bedeutung zu, da in diesem Fall komplexe Systeme mit weitreichenden Abhängigkeiten entstehen, deren Intensität es nicht erlaubt, Erweiterung statisch – durch Anhalten und Neustart des Systems – vorzunehmen oder auf einige wenige grobgranulare Komponenten einer bestimmten Abstraktionsebene zu beschränken. Allerdings wirft das flexible Erweitern eines Systems selbst neue Fragen auf. Hierzu gehören insbesondere Korrektheit der Erweiterungen, Rechtssicherheit sowie Redundanz, die in dieser Arbeit allerdings nicht vertieft werden.

# Kapitel 3

# MoDiS

Das Akronym MoDiS steht für *Model oriented Distributed Systems* (engl.: Modell orientierte Verteilte Systeme) und betont, daß abstrakte Modelle und Konstruktionskonzepte in diesem Projekt das Fundament für die Entwicklung verteilter Systeme sind. MoDiS ist ein top-down orientierter -, sprachbasierter - und gesamtheitlicher Ansatz zur Konstruktion qualitativ und quantitativ hochwertiger verteilter Systeme. Die Kombination der drei Säulen top-down, Sprache und Gesamtsystem ist, so wie sie in MoDiS betrieben wird, einzigartig.

Die Vorteile die sich daraus ergeben, sind ein sehr weiter Spielraum für kreative Entwicklungen und die Möglichkeit, ein hohes Maß an Homogenität und Integration zu erzielen. MoDiS bietet somit günstige Rahmenbedingungen, um hochwertige und leistungsfähige verteilte Systeme zu konzipieren. Die große Flexibilität und das gesamtheitliche Vorgehen bedeuten unweigerlich großen Aufwand, der zu leisten ist, um eine von Grund auf neu konzipierte Architektur in die Realität umzusetzen. Mit den derzeitigen primitiven Methoden und Werkzeugen der Systemprogrammierung kann dieser Aufwand in seinem vollen Umfang mit einer stark beschränkten Anzahl an Personenjahren in der Tat nicht bewältigt werden. Möglich ist dahingegen, Teilaspekte der zunächst frei entwickelten Konzepte zu realisieren, deren Vorzüge zu demonstrieren und sie in bestehende Systeme zu integrieren. Diese Integrationsschritte verlangen zwar Kompromisse, die von den Randbedingungen der Integrationsumgebung diktiert werden, sind aber dennoch wichtig, um sukzessive von sequentiellen und zentralen zu parallelen und verteilten Systemen migrieren zu können.

Die vorliegende Arbeit schließt sich dieser Vorgehensweise an, um losgelöst von den veränderbaren Eigenschaften gewohnter Systeme, neue Managementkonzepte möglichst frei entwickeln zu können. Da MoDiS als Ausgangsbasis für alle weiteren konstruktiven Maßnahmen gewählt wird, sind in den folgenden Abschnitten grundlegende Prinzipien der Vorgehensweise, Sichtweisen und bereits erarbeitete Konzepte beschrieben. Die Darstellung beschränkt sich dabei auf eine Zusammenfassung der wichtigsten Eigenschaften in der Reihenfolge Sprachbasierung, Gesamtsystem-Ansatz und top-down Orientierung. Ausführliche Darstellungen zu diesen Themen finden sich in [SEL+96, RW96, Win96b, Rad95].

## 3.1 Grundbegriffe

Ein *System* ist eine aus Komponenten zusammengesetzte Einheit, für deren Abgrenzung *innen* und *außen* definiert ist. Systeme können physisch existierende oder gedachte Einheiten sein und sind dementsprechend *konkret* oder *abstrakt*. Was außen zu einem System ist, wird

seine *Umwelt* genannt. Die Umwelt gehört nicht zu dem System. Können Wechselwirkungen zwischen dem System und seiner Umwelt existieren, so handelt es sich um ein *offenes*, andernfalls um ein *geschlossenes* System. In einem *dynamischen* System kann sich sowohl die Menge der Komponenten als auch ihre Eigenschaften über die Zeit verändern. Falls das nicht der Fall ist, heißt das System *statisch*.

Die *Komponenten* des Systems befinden sich innen zu diesem. Sie sind selbst Einheiten, für die *innen* und *außen* definiert ist und können aus weiteren Komponenten zusammengesetzt sein. Nicht zusammengesetzte Komponenten heißen *Atome*. Zusammengehörige Mengen von Komponenten eines Systems werden *Komplexe* genannt. *Aktive Komponenten* haben Speicher- und Verarbeitungsfähigkeiten, die sie zur Ausführung von Berechnungen befähigen. *Passive Komponenten* besitzen lediglich Speicherfähigkeit und sind Hilfsmittel der aktiven Komponenten. Komponenten bieten nach außen und nach innen *Operationen* für ihre Nutzung an. Äußere Operationen der Komponenten können aus ihrer Umwelt *aufgerufen* werden, während innere Operationen nur von ihr selbst und inneren Komponenten genutzt werden können. Eine aktive Komponente kann mit anderen aktiven Komponenten *kooperieren*, um gemeinsame Ziele zu erreichen. Hierzu können aktive Komponenten entweder wechselseitig ihre äußeren Operationen oder die gemeinsam genutzter, passiver Komponenten aufrufen. Existieren in einem System kooperierende aktive Komponenten, so heißt es *verteilt*. Je nachdem, ob es sich bei den kooperierenden Komponenten um physische oder gedachte Einheiten handelt, ist das System *real* (bzw. konkret) oder *abstrakt verteilt*.

Die *äußeren Eigenschaften* eines Systems sind die Grundlage für die Nutzung des Systems durch seine Umwelt. Die *inneren Eigenschaften* sind für die Konstruktion und das Verständnis eines Systems von Interesse. *Benutzern* in der Umwelt des Systems nutzen die Rechen- und Speicherfähigkeiten des Systems zur Lösung von Problemen, indem sie dem System *Aufträge* erteilen. Aufträge sind Berechnungvorschriften, die auf der einen Seite *Anwendungen* der Fähigkeiten des Systems und auf der anderen Seite *Problemlösungen* aus Sicht der Benutzer beschreiben.

Ein *Rechensystem* ist ein offenes, dynamisches System mit Fähigkeiten zur Speicherung und Verarbeitung von Information. Es besteht aus aktiven und passiven Komponenten. Jede Veränderung der Eigenschaften eines Rechensystems ist eine Wirkung der Ausführung einer Berechnung durch eine aktive Komponente.

Ein *INSEL-System* ist ein Rechensystem, das mit den Konzepten der Sprache INSEL konstruiert wurde. Die Konzepte von INSEL werden im folgenden erklärt.

## 3.2 Sprachbasierte Konstruktion

Da ein System aus Komponenten zusammengesetzt ist und sich seine Eigenschaften aus den Eigenschaften seiner Komponenten und den Abhängigkeiten zwischen diesen ergeben, werden Konzepte für die Konstruktion und Komposition von Komponenten benötigt. In MoDiS werden zu diesem Zweck mit der Sprache INSEL (Integration and Separation Supporting Language) Konzepte entwickelt und benannt, die geeignet sind, Parallelität, Verteilung zur Separation von Teilaufgaben sowie Kooperation zum Zweck der Integration auf hohem Abstraktionsniveau auszudrücken. Die Entwicklung eines homogenen Konzeptevorrats trägt den Beobachtungen Rechnung, die in 2.3.1 geschildert wurden. Abstrakte Verteilung, Parallelität und Kooperation sind in INSEL grundlegende Bestandteile der Sprache und homogen in diese integriert. Die Sprachbasierung im Projekt MoDiS bezieht sich somit weniger auf syn-

taktische Eigenschaften, als auf die Konzepte, die zur Spezifikation verteilter, paralleler und kooperativer Systeme – V-PK-Systeme – angeboten werden.

Die Angebote von INSEL zur Konstruktion von INSEL-Systemen basieren auf drei Grundprinzipien:

**Objektbasiert:** Die Komponenten des Systems sind Objekte, deren innere und äußere Eigenschaften durch innere und äußere Operationen definiert sind. Die Mengen der inneren und äußeren Operationen sind in der Regel nicht identisch. Objekte können nur durch Ausführung ihrer äußeren Operationen genutzt werden, die einen kontrollierten Übergang von außen nach innen bewirken. Somit haben sowohl INSEL-Systeme als Ganzes als auch die Komponenten, aus denen sie bestehen, die Eigenschaften von Objekten. Da das Konzepterepertoire von INSEL keine Vererbung von Eigenschaften vorsieht, sind INSEL-Systeme gemäß [Weg87] objektbasierte Systeme.

**Schachtelung:** Für jede Komponente des Systems ist ihre Einordnung innen und außen zu anderen Komponenten für die Dauer ihrer Existenz festgelegt. INSEL-Systeme werden durch konsequente Anwendung des Schachtelungsprinzips konstruiert. Zu Beginn besteht ein INSEL-System aus genau einer, der Haupt- oder synonym Wurzelkomponente, aus der sich das System dynamisch durch Erzeugung und Auflösung von Komponenten entwickelt, die sämtlich innen zur Hauptkomponente sind.

**Klassenbildung:** Die Komponenten des Systems sind Instanzen von Komponentenklassen, die die gemeinsamen Eigenschaften der entsprechenden Instanzen festlegen. Zusätzlich zu den Klasseneigenschaften besitzt jede Instanz spezifische Eigenschaften. Dies setzt voraus, daß es klassendefinierende Komponenten gibt, mit deren Hilfe die gemeinsamen Eigenschaften festgelegt und Instanzen der Klasse erzeugt werden können. Die Zuordnung von Komponenten zu Klassen ist für die Dauer ihrer Existenz invariant.

### 3.2.1 Komponentenarten

INSEL bietet ein Repertoire verschiedener Arten von Komponenten an, die als Hüllen mit wohldefinierten Eigenschaften den Rahmen für die Spezifikation von Klassen und der sich anschließenden Generierung von Instanzen vorgeben.

Die im letzten Abschnitt genannten, klassendefinierenden Komponenten, werden wegen ihrer Fähigkeit Instanzen zu erzeugen, **Generatoren** genannt. Generierte Instanzen werden **Inkarnationen** genannt. Generatoren sind damit ähnlich zu Typen oder Klassen in anderen imperativen oder objektorientierten Sprachen, unterscheiden sich jedoch konzeptionell in einigen Punkten von diesen, was vor allem in Kapitel 6 deutlich wird. Sie sind homogen in das System eingeordnet. Ihre Schachtelung bildet keine separate Hierarchie, die von Schachtelung der Inkarnationen, wie es zum Beispiel in C++ [Str91] oder Java [Sun95a] der Fall ist, getrennt ist. Da Inkarnationen ausschließlich als Instanzen implizit oder explizit definierter Generatoren erzeugt werden und ihre Klassenzugehörigkeit während ihrer Existenz invariant ist, ist INSEL als streng typisierte Sprache einzuordnen.

Entsprechend der Objektbasierung sind für alle Komponenten des Systems innere und äußere Operationen festzulegen. Diese Operationen können vordefiniert und damit implizit sein oder explizit definiert werden. Im einfachsten Fall besitzt eine Komponente lediglich vordefinierte Operationen und die Mengen ihrer äußeren und inneren Operationen sind identisch. Komponenten dieser Art werden **DE-Komponenten** (einfache Komponenten, bestehend aus

einem Deklarationsteil) genannt. Interessanter sind Konzepte für Komponenten, deren innere und äußere Operationen nicht ausschließlich vordefinierte Operationen sind und zudem verschieden sein können. Komponenten dieser Art heißen **DA-Komponenten** (Deklarations- und Anweisungsteil).

Zur Benutzung von Komponenten müssen diese identifizierbar sein. Bezüglich der Identifikation wird zwischen **anonymen** und **benannten** Komponenten unterschieden. Mit der Ausnahme von Generatoren, die stets benannt sind, können sowohl DE- als auch DA-Inkarnationen **benannt** oder **anonym** sein. Benannte Inkarnationen – **N-Inkarnationen** – werden bei der Erarbeitung des Deklarationsteils einer anderen Inkarnation erarbeitet und besitzen für die Dauer ihrer Existenz einen Namen, der eindeutig und nur innerhalb der erzeugenden Inkarnation bekannt ist. Namen von N-Inkarnationen können ferner weder dupliziert noch weitergereicht werden. Anonyme Inkarnationen – **Z-Inkarnationen** – können flexibel in Deklarations- oder Anweisungsteilen mit der Generierungsanweisung `NEW` erzeugt werden. Z-Inkarnationen werden durch Zeiger identifiziert, deren Werte dupliziert, weitergereicht und gelöscht werden können. Die N-/Z-Eigenschaft ist ebenfalls eine Klasseneigenschaft und für die Dauer der Existenz einer Inkarnation invariant.

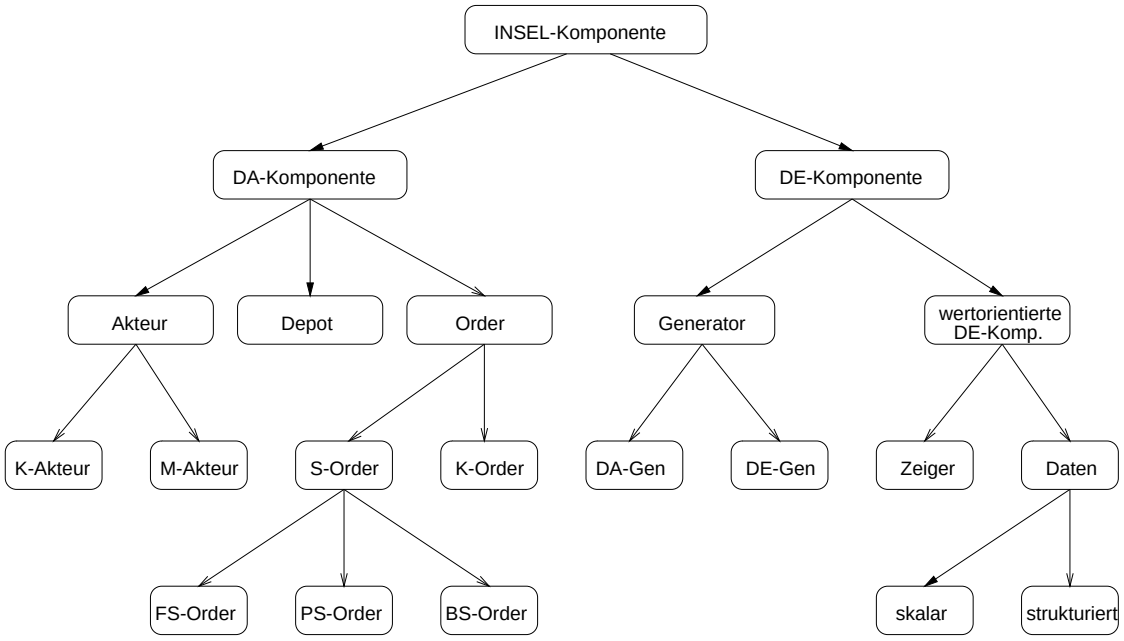


Abbildung 3.1: INSEL Komponentenarten

#### 3.2.1.1 DE-Komponenten

DE-Komponenten sind **wertorientierte Komponenten** oder Generatoren. Wertorientierte Komponenten besitzen Speicherfähigkeit und dienen der Speicherung von Zeiger und Daten, womit sie Variablen und Konstanten in imperativen Sprachen entsprechen. Wertorientiere Komponenten können skalar, zusammengesetzt oder Zeiger sein. Skalare Daten sind Inkarnationen vordefinierter Generatoren für `Character, Integer, Real` oder Inkarnationen eines explizit definierten Generators für einen skalaren Wertebereich. Der Wertebereich skalarer

Daten ist mit dem zugehörigen Generator festgelegt. Für skalare Daten sind stets die Operationen *lese* und *schreibe* für das Lesen und Schreiben von Werten sowie weitere für Vergleiche der Werte vordefiniert. Zusammengesetzte wertorientierte Komponenten sind entweder Felder (`Array`) oder Strukturen (`Record`), deren Elemente DA- oder DE-Komponenten jedoch keine Generatoren sein können. Für zusammengesetzte Komponenten sind Konstruktoren und Selektoren vordefiniert. Mit Zeiger können Z-Komponenten identifiziert werden, die bzgl. des Generators erzeugt wurden, mit dem der Generator des Zeigers qualifiziert ist (Zeiger sind typisiert). Die vordefinierten Operationen auf Zeiger sind *referenziere*, mit der die Identifikation einer Z-Komponente gespeichert wird und *dereferenziere* zum Selektieren der aktuell identifizierten Komponente.

Für jeden **Generator** ist die äußere Operation *erzeuge* implizit definiert, deren Ausführung die Generierung einer Instanz der Klasse bewirkt, die durch den Generator definiert ist. Bis auf Generatoren werden alle Komponenten des Systems mit den *erzeuge* Operationen von Generatoren generiert. Generatoren werden durch Erarbeitung ihrer Definition erzeugt. Die Deklaration von Variablen oder der Aufruf von Unterprogrammen entspricht in INSEL somit der Ausführung der Operation *erzeuge* auf dem entsprechenden Generator. Da alle Inkarnationen des Systems Instanzen von Generatoren sind, lassen sich ferner Feld-, Struktur, Zeiger-, Akteurgeneratoren, etc. unterscheiden. Ein Daten-Generator definiert eine Klasse skalarer oder strukturierter Daten mit einem explizit definierten Wertebereich. Ein DA-Generator ist ein Akteur-, Order- oder Depot-Generator. Die Eigenschaften von DA-Komponenten werden im nächsten Abschnitt erklärt. Zeiger-Generatoren definieren Klassen von Zeigern. Die Komponentenklasse, deren Elemente mit Zeigern der Zeiger-Klasse identifiziert werden können, ist die Qualifikation des Zeiger-Generators. Sie muß mit der Definition des Zeiger-Generators explizit festgelegt werden.

#### 3.2.1.2 DA-Komponenten

Auf der linken Seite der Abbildung 3.1 sind die verschiedenen möglichen Arten von DA-Komponenten dargestellt. DA-Komponenten sind die wesentlichen Bestandteile eines INSEL-Systems, da sie die Funktionalität und Berechnungen des Systems festlegen bzw. vorantreiben und dabei DE-Komponenten erzeugen, nutzen und auflösen. $X_t$ sei die Menge der DA-Komponenten des Systems $\mathcal{S}_t$ zum Zeitpunkt $t$. Alle DA-Komponenten besitzen unabhängig von ihrer Art gemeinsame Eigenschaften, die in Definition 3.1 aufgeführt sind.

**Definition 3.1 (Gemeinsame Eigenschaften von DA-Inkarnationen)**
*Jede DA-Inkarnation $x \in X_t$ besitzt folgende Eigenschaften:*

- *Sie ist aus Komponenten zusammengesetzt, die DA- oder DE-Komponenten inklusive Generatoren sein können.*

- *$x$ besitzt eine ausgezeichnete innere Operation, die mit $op(x)$ als „kanonische Operation“ von $x$ bezeichnet wird und explizit definiert ist. Der Körper von $op(x)$ ist der* Anweisungsteil *$L_a(x)$ von $x$.*

- *$x$ besitzt einen Deklarationsteil. Der Deklarationsteil bestimmt die linear geordnete Menge $L_0(x)$, deren Elemente deklarierte N-Komponenten sind. Elemente aus $L_0(x)$ können von $x$ nach außen zur Nutzung durch die Umwelt exportiert werden.*

- *für $x$ ist genau eine äußere Operation implizit festgelegt, die abhängig von der Art der DA-Komponente „initialisiere“, „führe_aus“ oder „starte“ heißt.*
- *Zu jedem Zeitpunkt ihrer Existenz befindet sich $x$ in einem der Zustände $V$ (vorbereitet), $A$ (in Ausführung), $R$ (rechnend), $W$ (wartend) oder $T$ (terminiert). Der Zustand von $x$ zum Zeitpunkt $t$ sei mit $\mu_t(x)$ bezeichnet. Nach Erzeugung von $x$ gilt zunächst $\mu_t(x) = V$, in dem die implizit definierte äußere Operation von $x$ aufgerufen werden kann.*

Es ist wichtig zu beachten, daß es die DA-Inkarnationen sind, die einen Anweisungsteil und einen Deklarationsteil besitzen und es sich dabei nicht um Generatoren oder gar um separat betrachtete Programmtexte handelt. Deklarations- und Anweisungslisten sind auf diese Weise in die Dynamik des Systems miteinbezogen.

$x$ setzt sich insgesamt aus der Menge der Komponenten $\tilde{L}(x) = L(x) \cup L_z(x)$ zusammen. $L(x) = L_a(x) \cup L_0(x)$ ist die Menge der $x$ statisch zugeordneten Komponenten. Nach der Erarbeitung des Deklarationsteils von $x$ ist $L_0(x)$ für die Dauer der Existenz von $x$ festgelegt. Im Falle benannter Komponenten ist die Zuordnung zu einer Menge $\tilde{L}(x), x \in X_t$ zu jedem Zeitpunkt $t$ eindeutig. $L_z(x)$ enthält anonyme Komponenten, die bei der Erarbeitung des Deklarationsteils von $x$, dynamisch im Laufe der Ausführung von $op(x)$ oder von anderen Komponenten des Systems erzeugt wurden und von $x$ identifiziert werden können. Die exakten Gesetzmäßigkeiten der Zuordnung von Komponenten ergeben sich aus der konzeptionell festgelegten Lebenszeit von Komponenten, die unten beschrieben wird.

Zur Spezifikation des Anweisungsteils bietet INSEL in etwa das Repertoire vordefinierter Operationen an, das auch in anderen imperativen Sprachen für gewöhnlich zur Verfügung steht. Neben den vordefinierten arithmetischen Operationen von DE-Komponenten gehören hierzu u. a. Fallunterscheidungen, Ein-/Ausgabeanweisungen, Schleifen zur Wiederholung von Abschnitten und Rücksprunganweisungen. Sprunganweisungen, die nicht der Schachtelungsstruktur von Komponenten und blockstrukturierten Anweisungen gehorchen, werden von INSEL nicht angeboten.

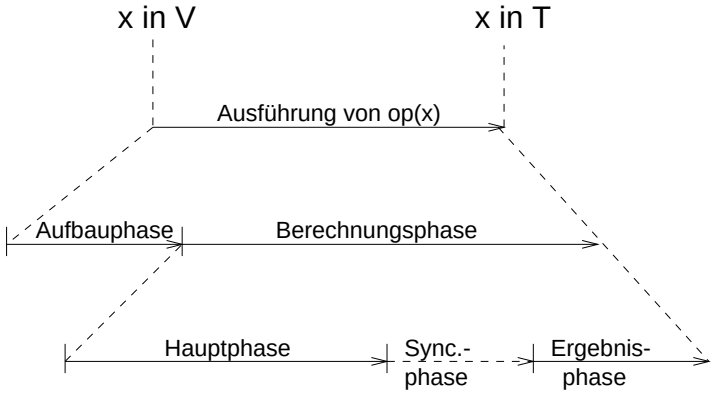


Abbildung 3.2: Ausführungsphasen der kanonischen Operation

DA-Komponenten werden dynamisch erzeugt und aufgelöst. Das Intervall $\Lambda(x) = [t_1, t_2]$ wird die **Lebenszeit** von $x$ genannt. Die Erzeugung einer DA-Komponente $x$ durch Aufruf von *erzeuge* auf dem $x$-Generator bewirkt die Generierung der $x$-Inkarnation im Zustand $V$, an die sich unmittelbar die Ausführung der auf $x$ definierten äußeren Operation anschließt. Der Aufruf der äußeren Operation bewirkt, abgesehen von Details:

- Zuweisung der Werte der aktuellen Eingabe- und Ein-/Ausgabeparameter an $x$
- den Wechsel der Operationsausführung von außen nach innen zu $x$, inklusive Übergang von $\mu(x) = V$ zu $\mu(x) = A$ bzw. $\mu(x) = R$
- Ausführung von $op(x)$
- Übergang zurück von innen nach außen

Die ersten beiden Schritte, Übergabe der Argumente und Wechsel nach $x$-innen, ist die **Startsynchronisation** von $x$ mit seinem Erzeuger. Die Ausführung von $x$ und die des Erzeugers sind während der Startsynchronisation unabhängig von der Komponentenart synchronisiert. Die anschließende Ausführung von $op(x)$ gliedert sich wie in Darstellung 3.2 abgebildet in verschiedene Phasen. Zunächst sind die **Aufbau-** und die **Berechnungsphase** zu unterscheiden. In der Aufbauphase wird der Deklarationsteil von $x$, d. h. die Menge $L_0(x)$ durch Generierung und Einordnung der Elemente erzeugt, wobei in der Regel weitere geschachtelte Erzeugungen stattfinden. Nach Beendigung dieser Kaskade beginnt die Berechnungsphase, die sich in die Abschnitte **Haupt-**, **Synchronisations-** und **Ergebnisphase** gliedert. In der Hauptphase werden die ebenfalls linear geordneten[1] Anweisungen des Anweisungsteils mit Ausnahme der Rücksprunganweisung `RETURN` gemäß ihrer Ordnung nacheinander ausgeführt. Die `RETURN`-Anweisung, die nur bei FS-Order (s. u.) möglich ist, wird in der abschließenden **Ergebnisphase** ausgeführt. In der **Synchronisationsphase** wird zuvor auf die Terminierung aller parallelen Berechnungen von Akteuren (s. u.) gewartet, die $\pi$-innen zu $x$ (siehe Strukturrelationen in 3.2.2) eingeordnet sind. Sie ist erst dann beendet, wenn jeder $\pi$-innere Akteur terminiert ist und ggf. seine Ergebnisse übergeben hat. In der Ergebnisphase werden schließlich Ausgabe- und Ein-/Ausgabeparameter-Werte zugewiesen und die `RETURN`-Anweisung ausgeführt. Anschließend befindet sich $x$ im Zustand $T$. Die Spanne von Beginn der Ergebnisphase bis zur Auflösung von $x$ wird **Abschlußsynchronisation** von $x$ genannt.

Neben den gemeinsamen Eigenschaften besitzt jede DA-Komponentenart spezifische Eigenschaften, die im folgenden beschrieben werden. Dabei sind zunächst die passiven Komponentenarten, Order und Depot, die ausschließlich Speicherfähigkeit besitzen, von den aktiven Akteuren zu unterscheiden, die zusätzlich über Rechenfähigkeiten verfügen.

**Order**

Order sind Inkarnationen, mit genau einer äußeren Operation, die implizit definiert ist und den Namen *führe_aus* trägt. Order dienen der Zusammenfassung und Strukturierung der Ausführung von Operationen, die im Anweisungsteil der kanonischen Operation $op(x)$ definiert sind und entsprechen Unterprogrammen in bekannten prozeduralen Sprachen. Das Generieren und Ausführen einer Order wird Aufruf genannt. Dementsprechend heißt die erzeugende Komponente *Aufrufer* (kurz „Rufer“) und die erzeugte Order *Aufgerufener*. Order sind stets anonyme Komponenten. Da zusätzlich keine Generatoren für Zeiger auf Order existieren, können Order nach ihrer Erzeugung weder identifiziert noch von außen genutzt werden. Mit Ausführung der äußeren Operation wird eine Order $x$ in den Zustand $\mu(x) = A$ überführt. Nach Beendigung von $op(x)$ gilt $\mu(x) = T$. Es existieren zwei Unterarten von Order, **S-Order** und **K-Order**. Letztere werden später zusammen mit dem K-Akteur-Konzept beschrieben. Die Taxonomie der S-Order (sequentiell) gliedert sich weiter

[1] Reihenfolge der Aufschreibung

in **PS-Order**, **FS-Order** und **BS-Order**, deren Eigenschaften Prozeduren, Funktionen und Blöcken entsprechen. Zu beachtende Besonderheiten sind, daß FS-Order einen Ergebniswert, jedoch keine Ausgabeparameter oder Ein-/Ausgabeparameter besitzen (frei von Nebeneffekten) können und BS-Order-Generatoren sich im Gegensatz zu allen anderen Generatoren in dem Anweisungsteil einer DA-Inkarnation befinden, d. h. für einen BS-Order-Gen. $b$ gilt: $\exists x \in X_t : b \in L_a(x) \wedge \nexists y \in X_t : b \in L_0(y)$. Ungeachtet dessen verfügen BS-Order ebenso wie alle anderen DA-Komponenten über einen Anweisungs- und einen Deklarationsteil.

**Depot**
Ein Depot definiert mit seinen inneren Komponenten einen strukturierten Speicher sowie Operationen zum Zugriff auf diesen, ähnlich wie ein Objekt in objektorientierten Sprachen. Für Depots sind die Zustände V, A und T definiert. Ein Depot $x$ ist von außen **benutzbar**, wenn es sich nach Ausführung von $op(x)$ im Zustand T befindet. Während der Ausführung von $op(x)$ gilt $\mu(x) = A$. Die implizit definierte äußere Operation von Depots heißt *initialisiere*, bewirkt die Ausführung von $op(x)$ und besitzt somit die Bedeutung eines Konstruktors. Weitere äußere Operationen, die **Zugriffsoperationen**, können explizit innerhalb Depots definiert sein und nach außen exportiert werden. Sie sind allerdings erst dann aufrufbar, wenn das Depot benutzbar ist. Depots können N-Inkarnationen oder Z-Inkarnationen sein. Dadurch, daß Depots beliebige Komponenten beinhalten können und deren Operationen als Depot-Zugriffsoperationen nach außen exportieren können, sind sie ein sehr mächtiges Konzept. In Ergänzung zu den Fähigkeiten von Objekten in anderen Sprachen bedeutet z. B. der Export geschachtelter Generatoren in etwa das Anbieten eines Typs durch ein dynamisch erzeugtes und aufgelöstes Objekt.

**Akteur**
Das Akteur-Konzept ermöglicht explizite Parallelverarbeitung (bzw. Task-Parallelität gemäß 2.3.1.3). Akteure sind die aktiven Komponenten des Systems, die dessen Berechnungen vorantreiben. Jeder Akteur definiert einen separaten, abstrakten Kontrollfluß, in dem eine Berechnung durchgeführt wird. Dies ist die Rechenfähigkeit der Akteure, zu der sie zusätzlich noch Speicherfähigkeit besitzen. Akteure werden wie alle anderen Komponenten des INSEL-Systems dynamisch erzeugt und aufgelöst. Der Parallelitätsgrad kann somit sehr flexibel und dynamisch gestaltet werden. Zudem kann sowohl die Zeit- als auch Raum-Granularität von Akteuren durch die Anzahl der auszuführenden Operationen und der Mächtigkeit von $L_0(x)$ stark variieren. Ebenso wie alle anderen Komponenten werden auch Akteure nicht explizit sondern implizit entsprechend ihrer strukturellen Einordnung aufgelöst. Das Akteur-Konzept entspricht somit der Forderung aus 2.3.1 nach expliziter Parallelität auf hohem Abstraktionsniveau, losgelöst von Realisierungseigenschaften sowie homogener und vollständiger Integration in die Sprache.

Für Akteure sind die Zustände V, R, W und T definiert. Die implizit definierte Operation eines Akteurs $x$ heißt *starte* und bewirkt $\mu(x) = R$, womit $op(x)$ ausgeführt wird, in der weitere Akteure, Order, Depots, etc. erzeugt werden. Bei Erzeugung eines Depot oder einer Order durch $x$, führt $x$ deren kanonische Operation sequentiell eingeordnet in seine eigene aus. Wird dahingegen ein Akteur $y$ erzeugt, so führt $y$ $op(y)$ selbständig parallel zu $x$ aus.

Die Komponente $z \in X_t$, deren kanonische Operation $op(z)$ zum Zeitpunkt $t$ von dem Akteur $x$ ausgeführt wird, heißt **Ausführungskomponente** von $x$, bezeichnet mit $\varphi_t(x) = z$. Zu Beginn und Ende der Ausführung von $op(x)$ gilt $\varphi_t(x) = x$. Die Ausführung des Akteurs

wechselt kellerartig zwischen den passiven Komponenten, die im Laufe der Berechnung des Akteurs erzeugt und aufgelöst werden. Das Starten einer parallelen Berechnung motiviert die Bezeichnung des Verhältnisses zwischen Akteur-Erzeuger und erzeugtem Akteur als *Starter* und *Gestarteter*. Das Akteur-Konzept wird von INSEL in zwei Varianten angeboten, den mono-operationalen M-Akteuren und den K-Akteuren (Kommunikationsoperationen).

**M-Akteur** M-Akteure besitzen analog zu Order nur eine implizit definierte äußere Operation. Weitere explizit definierte äußere Operationen sind nicht möglich, da M-Akteure ebenso ausschließlich anonyme Inkarnationen und nach ihrem Start nicht mehr von außen benutzbar sind. Die Erzeugung von M-Akteuren erfolgt ausschließlich im Anweisungsteil von DA-Komponenten.

**K-Akteur** Im Gegensatz dazu können für K-Akteure weitere äußere Operationen, die Kommunikationsoperationen, explizit festgelegt werden. Außer Kommunikationsoperationen können K-Akteure keine Komponenten nach außen exportieren. K-Akteure können Z- oder N-Inkarnationen sein, wobei sie im letzteren Fall im Deklarationsteil anderer Komponenten erzeugt werden.

Kommunikationsoperationen dienen der synchronisierten Kooperation von Akteuren und sind die *erzeuge*-Operationen, der K-Order-Generatoren (Kommunikations-Order), die von dem K-Akteur exportiert werden. Wenn $x$ ein K-Akteur und $y$ ein anderer Akteur ist, dann kann $y$ mit $x$ über dessen Kommunikationsoperationen zum Zeitpunkt $t$ synchronisiert kommunizieren, wenn $\mu_t(x) = R$ oder $\mu_t(x) = W$. $y$ erteilt als **Auftraggeber** dem K-Akteur $x$, der den K-Order-Generator anbietet, den Auftrag zur Erzeugung der K-Order und versetzt sich in den Zustand $W$. Dieser Auftrag wird von $x$ als **Auftragnehmer** entgegengenommen und an definierten Punkten der Ausführung seiner kanonischen Operation, gekennzeichnet durch die Anweisung `ACCEPT`, angenommen. An die Annahme schließt sich unmittelbar die gemeinsame Ausführung der kanonischen Operation der K-Order $k$ durch Auftraggeber und -nehmer – $\varphi(x) = \varphi(y) = op(k)$ – in einem synchronisierendem Rendezvous an. $x$ und $y$ befinden sich nach Beendigung von $k$ beide im Zustand $R$ und führen ihre Berechnungen wieder parallel fort. Dieses **operationen-orientierte Rendezvous** ist bislang das einzige Synchronisationskonzept von INSEL. K-Akteure sind damit vergleichbar mit dem Task-Konzept der Sprache Ada [Ada83]. K-Order verhalten sich mit Ausnahme ihrer synchronisierenden Wirkung und ihrer kooperativen Erzeugung und Ausführung sonst wie PS-Order (Prozeduren).

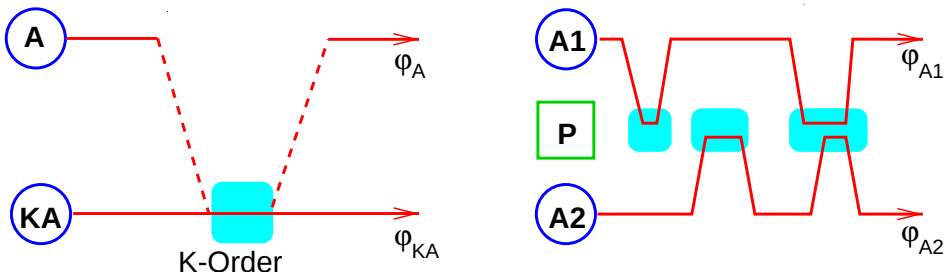


Abbildung 3.3: Kooperation zwischen Akteuren

**Kooperation** Das Konzept der Kooperation via K-Order ermöglicht zusätzlich zur Synchronisation den Austausch von Nachrichten mit Hilfe der K-Order-Parameter. Wenngleich dieses Konzept nicht mit dem Versenden von Nachrichten in „message passing“ Systemen verwechselt werden sollte, so ermöglicht es dennoch die Formulierung paralleler Problemlösungen in INSEL, deren Kooperation zwischen den parallel rechnenden aktiven Komponenten sich gut auf den Austausch von Nachrichten abbilden läßt. Neben diesem operationen-orientierten Rendezvous können INSEL-Akteure zusätzlich mittelbar und nicht-synchronisiert über gemeinsame nutzbare Depots oder wertorientierte DE-Komponenten kooperieren. Dies entspricht der Kooperation über gemeinsamen Speicher. INSEL ist daher gleichfalls für parallele Problemlösungen geeignet, deren Struktur auf dem Paradigma des gemeinsamen Speichers oder dem Austausch von Nachrichten beruht. Es sei noch einmal betont, daß in INSEL keine Kooperationsform an Stellen des verteilten Systems gebunden ist, sondern ausschließlich an die abstrakte Struktur des INSEL-Systems. Abbildung 3.3 stellt die beiden Kooperationsvarianten dar. Links kommuniziert ein Akteur $A$ mit einem K-Akteur $KA$ über eine K-Order. $\varphi(A)$ wechselt für die Dauer der mit $KA$ synchronisierten Ausführung der K-Order einmalig und kehrt anschließend zurück. Im rechten Beispiel kooperieren zwei Akteure $A_1$ und $A_2$, indem sie die äußeren Operationen des Depots $P$ aufrufen und dabei Werte in Komponenten $k \in L_0(P)$ schreiben und lesen. Die aktuellen Ausführungskomponenten der Akteure wechseln voneinander unabhängig zu der aufgerufenen Depot-Operation und zurück.

#### 3.2.1.3 Parametrisierung von DA-Komponenten

DA-Generatoren können mit formalen Parametern versehen werden, deren aktuelle Werte bei Erzeugung einer Inkarnation dieser zugewiesen werden. Für jeden formalen Parameter ist ein Name und ein *Übergabemodus* festgelegt. In der erzeugten Inkarnation sind die zugewiesenen Parameter N-Komponenten einer Klasse, die ihrerseits durch einen Generator definiert ist. Folgende drei Übergabemodi stehen zur Auswahl[2]:

- *call-by-value*; es handelt sich um einen Eingabeparameter, dessen Wert einmalig bei der Erzeugung der Inkarnation zugewiesen wird
- *call-by-result*; es handelt sich um einen Ausgabeparameter, dessen Wert einmalig in der Ergebnisphase bei Beendigung der kanonischen Operation an die vom Erzeuger bestimmte Komponente zugewiesen wird.
- *call-by-value-result*; es handelt sich um einen transienten Ein-/Ausgabeparameter, dessen Wert bei der Erzeugung der Inkarnation zugewiesen und in der Ergebnisphase ggf. modifiziert in die vom Erzeuger bestimmte Komponente zurückgeschrieben wird.

Diese Modi wurden gezielt im Hinblick auf die speziellen Anforderungen in verteilten und parallelen Umgebungen festgelegt. *Call-by-reference* würde die Schachtelungsstruktur des Systems zerstören, eine Vielzahl zusätzlicher und dauerhafter Kooperationsabhängigkeiten hinzufügen und eine weitere schwierig zu erkennende Fehlerquelle für parallele Problemlösungen bedeuten. Durch die gewählten Modi wird zum einen Struktur und zum anderen Black-Box-Verhalten gewährleistet. Die Möglichkeiten, Sprachen wie Java, die einzig *call-by-reference* anbieten, in verteilten Umgebungen erfolgreich einsetzen zu können, sind äußerst zweifelhaft.

[2]Die hier formulierten Eigenschaften der letzteren beiden Übergabemodi sind beabsichtigte Abwandlungen der Festlegungen in [SEL+96], die Erkenntnissen aus der Realisierung Rechnung tragen.

Die Zulässigkeit der verschiedenen Übergabemodi hängt von der Art der zu parametrisierenden Komponente ab und ist in Tafel 3.1 angeführt.

| **Komponenten-Art** | **Art des Parameters** | **Übergabemodi** |
|---|---|---|
| M-Akteur | Daten, Zeiger | alle |
| K-Akteur | Daten, DA-Zeiger | Eingabe |
| FS-Order | Daten, Zeiger | Eingabe |
| PS-Order | Daten, Zeiger | alle |
| BS-Order | — | keine |
| K-Order | Daten, Zeiger | alle |
| Depot | Daten, DA-Zeiger | Eingabe |

Tabelle 3.1: Parametrisierung von DA-Komponenten

### 3.2.2 Systemstrukturen

Mit den erklärten Konstruktionskonzepten von MoDiS steht ein Repertoire für die Spezifikation verteilter Problemlösungen auf hohem Abstraktionsniveau zur Verfügung. Durch die Klassenbildung, Schachtelung von Generatoren und Inkarnationen, Kapselung von Daten und Operationen in Objekten und die präzise Festlegung der Eigenschaften von Komponenten, z. B. Start- und Abschlußsynchronisation, bestehen zwischen den Komponenten eines INSEL-Systems enge Abhängigkeiten. Diese strukturellen Abhängigkeiten werden von dem Entwickler einer verteilten Problemlösung allein durch Nutzung der INSEL Sprachkonzepte etabliert, ohne daß hierfür zusätzliche Spezifikationen oder Hinweise an den Übersetzer oder das Laufzeitsystem erforderlich sind. Trotz der Einfachheit ihrer Spezifikation sind diese Strukturen eine wichtige Informationsquelle für das systemintegrierte Ressourcenmanagement, um automatisiert adäquate Entscheidung treffen zu können. Im folgenden werden die wichtigsten Strukturrelationen, soweit es für das Verständnis des Managements erforderlich ist, skizziert. Sämtliche unten angeführten Strukturrelationen sind irreflexiv, asymmetrisch und Teilmengen von $X_t \times X_t$.

#### 3.2.2.1 Definitionsstruktur und Ausführungsumgebung

Die Schachtelung der Generatoren induziert einen hierarchisch strukturierten Namensraum über der Menge der Komponenten. Für jede Komponente $x \in X_t$ eines INSEL-Systems ist durch Schachtelung mit lexikalischer Bindungsregel die Menge der sichtbaren und zugreifbaren Komponenten zum Zeitpunkt $t$ – die Ausführungsumgebung $U_t(x)$ – festgelegt. Die Grundlage für die Ermittlung von $U_t(x)$ ist die Definitionstruktur $\delta$.

**Definition 3.2 ($\delta$-Struktur)**
*Eine Komponente $a \in X_t$ ist definitorisch abhängig von $b \in X_t$ – $(a, b) \in \delta_t \Leftrightarrow$*
*Der Generator von $a$ ist im Deklarationsteil von $b$ definiert – $G(a) \in L_0(b)$.*
*$\delta_t^*$ ist die transitive Hülle von $\delta_t$.*

Jede DA-Komponente $x \in X_t$ ist für ihre gesamte Lebenszeit in $\delta$ eingeordnet. $U_t(x)$ wird im wesentlichen durch das Aufsammeln von Komponenten entlang der Schachtelung der Generatoren, d. h. Verfolgen von $\delta_t^*$, berechnet. Sei $x$ ein Element aus $X_t$, dann wird mit $\delta_t$ zunächst

der Generator $G(x)$ von $x$ bestimmt, der sich in $L_0(y)$ befindet. Alle Komponenten, die in $y$ vor $G(x)$ deklariert sind, sind für $x$ sichtbar und werden zu $U_t(x)$ hinzugefügt. Die Berechnung fährt rekursiv mit $z$ fort, wobei $(y, z) \in \delta_t$, bis die Wurzelkomponente des Systems erreicht ist. Sichtbare DE-Komponenten und anonyme DA-Komponenten sind ebenfalls in $U_t(x)$ enthalten. Eine exakte Darstellung des etwas aufwendigeren Berechnungsschemas ist im Kontext der vorliegenden Arbeit nicht erforderlich.

#### 3.2.2.2 Ausführungsstruktur

In der Ausführungsstruktur sind die parallelen, sequentiellen und kooperativen Beiträge der Berechnungen des Systems verzeichnet. Die Ausführungsstruktur liefert somit u. a. wertvolle Information für die Lastverwaltung und das Scheduling.

Jeder Akteur besitzt, wie oben bereits erklärt wurde, einen eigenen abstrakten Ausführungsfaden, in dem er seine kanonische Operation nebenläufig zur Berechnung seines Erzeugers ausführt. Die Beziehung zwischen den Berechnungen der Akteure und ihren Erzeugern ist in der Parallelitätsstruktur $\pi$ festgehalten.

**Definition 3.3 ($\pi$-Struktur)**
*Ein Akteur $a \in X_t$ ist parallel abhängig von Komponente $b \in X_t$ – $(a, b) \in \pi_t \Leftrightarrow$*
*$a$ ist ein M-Akteur oder ein benannter K-Akteur, der von $b$ erzeugt wurde.*
*$\pi_t^*$ ist die transitive Hülle von $\pi_t$.*

Die Ausführung der kanonischen Operation einer neu erzeugten passiven Komponente wird sequentiell in die Ausführung der erzeugenden Komponente eingebettet. Der Zusammenhang zwischen den sequentiellen Anteilen der Berechnung eines Akteurs $x \in X_t$, der durch das dynamische Wechseln der Ausführungskomponente $\varphi(x)$ entsteht, ist durch die $\sigma$-Struktur beschrieben.

**Definition 3.4 ($\sigma$-Struktur)**
*$a \in X_t$ ist sequentiell abhängig von $b \in X_t$ – $(a, b) \in \sigma_t \Leftrightarrow$ Die Ausführung von op$(a)$ erfolgt sequentiell eingeordnet in op$(b)$;*
*d h. $\varphi(x)_{t_1} = b, \varphi(x)_{t_2} = a, \varphi(x)_{t_3} = b \quad \wedge \quad t_1 < t_2 < t_3$*
*$\sigma_t^*$ ist die transitive Hülle von $\sigma_t$.*

Den dritten Beitrag zur Ausführungsstruktur liefert die Kommunikationsstruktur. Mit ihr werden Kommunikationsabhängigkeiten zwischen Komponenten beschrieben, die sich vorübergehend synchronisieren, um die kanonische Operation einer K-Order in einem Rendezvous gemeinsam auszuführen.

**Definition 3.5 ($\kappa$-Struktur)**
*Sei $k \in X_t$ eine K-Order; $b \in X_t$ – $(k, b) \in \kappa_t \Leftrightarrow \exists a \in X_t$ : $a$ ist ein Akteur $\wedge \varphi(a) = b \wedge a$ ist Auftraggeber für $k$.*

Die Ausführung von $k$ erfolgt sequentiell eingebettet in die Ausführung des K-Akteurs $c$, der den Generator für $k$ als lokale Komponente beinhaltet, d. h. $\exists$K-Akteur $c \in X_t$ : $(k, c) \in \delta_t \wedge (k, c) \in \sigma_t$. Die Kommunikationsstruktur $\kappa$ beschreibt somit die Synchronisation der Ausführungsfäden des Systems, die zusätzlich zur konzeptionell festgelegten Start- und Abschlußsynchronisation während der Ausführung von Akteuren stattfinden.

In der Ausführungsstruktur $\alpha$ werden die Beiträge der Parallelitätsstruktur, der Sequentialitätsstruktur und der Kommunikationsstruktur zusammengefaßt. Mit $\alpha$ sind sämtliche Ausführungen des Systems inklusive ihrer Abhängigkeiten vollständig erfaßt.

**Definition 3.6 ($\alpha$-Struktur)**
*Sei* $a, b \in X_t$*;*
*Die Ausführung von a ist abhängig von b –* $(a,b) \in \alpha_t \Leftrightarrow (a,b) \in \sigma_t \vee (a,b) \in \pi_t \vee (a,b) \in \kappa_t$*.*
$\alpha_t^*$ *ist die transitive Hülle von* $\alpha_t$*.*

#### 3.2.2.3 Lebenszeitstruktur

Besondere Bedeutung für das V-PK-Management kommt neben der Definitions- und der Ausführungsstruktur den Lebenszeitabhängigkeiten zwischen den Komponenten des Systems zu. Die definitorische Lebenszeit von Komponenten unterscheidet sich grundlegend für Z-Komponenten, deren Lebenszeit nach der Zeigergeneratorstruktur $\gamma$ festgelegt ist, und N-Komponenten, deren Einordnung gemäß der Lokalitätsstruktur $\lambda$ erfolgt.

Für die Komponenten, die im Deklarationsteil einer Komponente $x$ deklariert sind, kann angenommen werden, daß sie in erster Linie von $x$ und von in $x$ geschachtelten Komponenten, d. h. $y \in X_t : (y,x) \in \delta_t^*$, genutzt werden. In der $\lambda$-Struktur enthaltene Information über Lokalität wird von dem Management u. a. für die Plazierung von Komponenten genutzt.

**Definition 3.7 ($\lambda$-Struktur)**
$a \in X_t$ *ist lokal zu* $b \in X_t - (a,b) \in \lambda_t \Leftrightarrow$ *a ist eine lokale N-Komponente von b –* $a \in L_0(b)$*.*
$\lambda_t^*$ *ist die transitive Hülle von* $\lambda_t$*.*

Zeiger auf Z-Komponenten können weitergereicht, dupliziert und aufgelöst werden. Um das Problem verwaister Referenzen zu vermeiden, sollten auch anonyme Komponenten erst dann automatisch aufgelöst werden, wenn es keine Nutzer dieser Komponente mehr gibt. Für das Aufsammeln nicht mehr zugreifbarer Komponenten, *garbage collection* [Wil94, Cri98], ist kein in verteilten Umgebungen effizient realisierbares Verfahren bekannt [ML95, PS95]. Daneben erschweren die vielfältigen Abhängigkeiten, die durch N-Komponenten und Zeiger möglich sind, die verteilte Verwaltung des Speichers erheblich. Um diese negativen Eigenschaften zu reduzieren, definiert INSEL auch für Z-Komponenten eine konzeptionelle Lebenszeit, die sich an den Zeigergeneratoren orientiert. Definition 3.8 setzt voraus, daß weder Typumwandlungen für anonyme Komponenten noch Zeigerarithmetik erlaubt sind. In INSEL sind Operationen dieser Art nicht vorgesehen, um eine stärkere Strukturierung des Systems zu ermöglichen. In der Praxis erweist sich diese Einschränkung nur in seltenen Fällen als problematisch; dies zeigt auch die Sprache JAVA.

**Definition 3.8 ($\gamma$-Struktur)**
$a \in X_t$ *ist* $\gamma$*-abhängig von* $b \in X_t - (a,b) \in \gamma_t \Leftrightarrow$ *a ist eine Z-Komponente* $\wedge$ *in b befindet sich der Generator* $G(a)$*, der benötigt wird, um Zeiger auf a zu generieren.*

Basierend auf der $\gamma$ und der $\lambda$-Struktur sowie den konzeptionellen Festlegungen der Ausführungsstruktur kann nun die Lebenszeitstruktur $\epsilon$ angegeben werden, die als Grundlage für den Entwurf der Managementarchitektur dient.

**Definition 3.9 ($\epsilon$-Struktur)**
*Seien $a, b \in X_t$; a ist lebenszeitabhängig von b –*

$$(a,b) \in \epsilon_t \Leftrightarrow \begin{cases} (a,b) \in \sigma_t & \textit{falls a S-Order oder N-Depot im Zustand A ist} \\ (a,b) \in \pi_t & \textit{falls a M-Akteur ist} \\ (a,b) \in \kappa_t & \textit{falls a K-Order ist} \\ (a,b) \in \lambda_t & \textit{falls a N-Depot im Zustand T oder benannter K-Akteur} \\ (a,b) \in \gamma_t & \textit{falls a ein anonymes Depot oder ein anonymer K-Akteur ist} \end{cases}$$

*$\epsilon_t^*$ ist die transitive Hülle von $\epsilon_t$.*

Jede DA-Komponente wird bei ihrer Erzeugung in die Lebenszeitstruktur eingeordnet und bei ihrer Auflösung wieder ausgeordnet. Mit Ausnahme von Depots bleibt die Einordnung in $\epsilon$ während ihrer gesamten Lebenszeit invariant. Weiter folgt aus Definition 3.9, daß die Lebenszeit jeder DA-Komponente von genau einer anderen DA-Komponente des Systems abhängt. Mit der Lebenszeitstruktur ist auf diese Weise eine konzeptionelle Grundlage geschaffen, um zu gewährleisten, daß keine Komponente aufgelöst wird, solange davon abhängige Komponenten existieren. Die Terminierung und Auflösung von Komponenten ist basierend auf $\epsilon$ mit der *Auflösungsregel* (Definition 3.10) konzeptionell festgelegt.

**Definition 3.10 (Auflösungsregel)**
*$(a, b) \in \epsilon$ legt fest, daß b in den Zustand T übergegangen sein muß, bevor a aufgelöst werden kann. a muß terminiert und aufgelöst worden sein, bevor b aufgelöst wird.*

DE-Komponenten wurden bei der Definition der Systemstrukturen nicht explizit erwähnt, wenngleich besonders zusammengesetzte Felder und Records wichtige Beiträge zur Gesamtberechnung liefern. Für die weiteren Überlegungen genügt es, die Einordnung von wertorientierten DE-Komponenten als äquivalent zu der Einordnung von Depots zu betrachten, mit dem einzigen Unterschied, daß sie nicht in der Ausführungsstruktur verzeichnet werden, da sie keine kanonische Operation besitzen.

### 3.2.3 Programmiersprache INSEL

Die Programmiersprache INSEL ist das syntaktische Vehikel zur präzisen Beschreibung von INSEL Systemen. Mit ihr sind die in 3.2 erläuterten Konzepte operationalisiert. Die Wahl einer neuen Sprache anstelle der additiven Erweiterung einer existierenden Sprache ermöglicht es, konzeptionelle Homogenität auf Ebene der Programmierung durchzusetzen. Die neu entworfene Programmiersprache INSEL ist eine imperative Sprache mit einer an Ada88 [Ada83] angelehnten Syntax. Eine exakte Darstellung der Syntax ist in [RW96] erfolgt. Da die Sprachsyntax als Teil des Forschungsprojekts MoDiS Veränderungen unterliegt, ist im Anhang A zusätzlich ein aktueller Schnappschuß der konkreten und abstrakten Syntax von INSEL angeführt.

Trotz des selbstverständlich engen Bezugs zwischen den INSEL Konzepten und der Programmiersprache sollten diese beiden Aspekte nicht verwechselt werden. Das beschriebene Konzepterepertoire wurde entwickelt, um qualitativ hochwertige V-PK-Systeme auf hohem Abstraktionsniveau, orientiert an anschaulichen Modellen, konstruieren zu können. Die Maximen der Syntax sind anders gelagert und betreffen Prägnanz, Einprägsamkeit, Unterstützung der Fehler-Detektion, etc. und sind für die Entwicklung des V-PK-Managements von untergeordneter Bedeutung.

Wichtig ist allerdings die Einordnung von Programmtexten in das INSEL-System. Bei den oben beschriebenen Konzepten: Generatoren, Inkarnationen, benannt oder anonym, handelt es sich um Komponenten, die dynamisch erarbeitet, erzeugt und aufgelöst werden. Die naheliegende Assoziation, daß es sich bei Programmtexten um Generatoren handelt, ist falsch, da im Gegensatz zu dem Generator Text nicht mit der selben Dynamik erarbeitet und aufgelöst wird. Stattdessen ist jede DA-Inkarnation $x \in X_t$ des dynamischen INSEL-Systems $\mathcal{S}_t$ zu jedem Zeitpunkt $t$ durch einen INSEL-Programmtext mit ihrem Anweisungs- und Deklarationsteil beschrieben. Programmtexte $\mathcal{P}$, wie in Abbildung 3.4, beinhalten deshalb weder Inkarnationen noch Generatoren, sondern beschreiben lediglich mögliche Komponenten und ihr Entwicklungspotential. Werden die Begriffe Inkarnation, Generator usw. dennoch verwendet, so handelt es sich um eine Vorabinterpretation des Verhaltens einer Ausführung von $\mathcal{P}$ über die Zeit $t$.

```
MACTOR system IS                                    – Wurzel bzw. Haupt-M-Akteur (1)
 DEPOT SPEC D_t ...                                 – Export Depot-Zugriffsoperationen (2)
   FUNCTION get ...

 DEPOT D_t (x: IN INTEGER) IS                       – Deklaration einer Depot-Klasse (3)
   v : ARRAY[1..x] OF INTEGER;                      – wertorientierte N-Komponente (4)
   FUNCTION get ...                                 – Def. FS-Order-Gen.; Zugr.-Op des Depot (5)
     RETURN x;
   ...
 BEGIN ... END D_t;                                 – Anweisungsteil von D_t (6)

 TYPE DP_t IS ACCESS D_t;                           – DA-Zeiger-Klasse, D_t qualifiziert (7)

 CACTOR T_t IS                                      – K-Akteur-Klasse T_t (8)
   c     : INTEGER;                                 – benannte DE-Komponenten im Dekl.-Teil (9)
   e     : D_t;
   dp    : DP_t;
   ENTRY coop IS                                    – K-Order-Klasse Vereinbarung (10)
   BEGIN                                            – K-Order-Klasse coop Anweisungsteil (11)
     ...  count := count + 1; ...
   END coop;
 BEGIN                                              – Anweisungsteil der K-Akteur-Klasse T_t (12)
   dp := NEW D_t(42);                               – Erzeugung eines anonymen Depots (13)
   ... ACCEPT coop; ...                             – Annahme eines K-Order Auftrags (14)
 END

 d: D_t(8);                                         – benannte DE-Komponenten Wurzelakteur (15)
 t: T_t;                                            – benannter K-Akteur der Klasse T_t (16)
 BEGIN                                              – Anweisungsteil Wurzel-M-Akteur (17)
   OUTPUT(d.get);                                   – Ausgabe-Anw. + Depot-Zugriff (18)
   t.coop; ...                                      – Auftrag an den K-Akteur t (19)
 END;
```

Abbildung 3.4: INSEL Beispielprogramm

Abbildung 3.4 illustriert einige der INSEL-Konzepte und deren Spezifikation mittels der Programmiersprache INSEL. An der Position (1) beginnt die Beschreibung der M-Akteur-Klasse `System`. Dies ist eine ausgezeichnete Klasse von der genau eine Inkarnation erzeugt wird, bei der es sich dann um die Wurzelkomponente des, im Beispiel sehr einfachen, Systems handelt. Ab Position (2) wird die Schnittstelle von Depots der Klasse `D_t` deklariert, indem

die exportierten Zugriffsoperationen von Komponenten spezifiziert werden. Im Beispiel ist die *erzeuge*-Operation des FS-Order-Generators `get` als nach außen exportiert spezifiziert. Die Implementierung der Klasse `D_t` umfaßt einen Deklarations- (3) und einen Anweisungsteil (6). (7) spezifiziert einen DA-Zeiger-Klasse, die mit Komponenten der Klasse `D_t` qualifiziert ist. An der Markierung (8) beginnt die Beschreibung der K-Akteur-Klasse `T_t` (Schnittstelle und Implementierung), deren Inkarnationen einen Generator für die K-Order `coop` anbieten. Der Anweisungsteil des Wurzelakteurs beginnt an Position (17).

## 3.3 Gesamtsystem-Ansatz

In Rahmen der Beschreibung der Aufgabenstellung wurde in 1.1.1 auf Seite 4 mit dem Ein-Programm-Ansatz die zweite Säule des MoDiS-Ansatzes neben der Sprachbasierung bereits skizziert. Mit den beschriebenen INSEL-Konzepten wird genau ein abstrakt verteiltes System konstruiert, das Problemlösungskomplexe und damit Anwendungen und Betriebssystemfunktionalität in genau einem System integriert, das sich aus INSEL-Komponenten zusammensetzt, die in einem engen Zusammenhang zueinander stehen.

Eine wichtige Konsequenz dieses Ansatzes ist, daß eine Trennlinie zwischen Statik und Dynamik, wie sie in üblichen Systemen zwischen Sprache plus Programm und der davon losgelösten Ausführung von Programmen gezogen wird, in MoDiS nicht existiert. Stattdessen sind die Eigenschaften des Systems und der Rahmen für seine dynamische Weiterentwicklung mit Mitteln der Sprache INSEL klar definiert. Neue Anwendungen – im Sinne herkömmlicher Systeme – expandieren inkrementell und homogen das in Ausführung befindliche System, indem zunächst entsprechende Generatoren eingebracht und anschließend ihre *erzeuge*-Operationen aufgerufen werden. Sämtliche Abläufe des Systems sind Berechnungen von Akteuren genau eines INSEL-Programms und zwischen allen Komponenten des Systems existieren wohldefinierte Abhängigkeiten. Dies gilt für Komponenten innerhalb von Problemlösungen – Intra-Dependenzen – als auch für Komponenten verschiedener Problemlösungen – Inter-Dependenzen. Dazu sei bemerkt, daß beide Arten von Abhängigkeiten in jedem System existieren, letztere im allgemeinen aber weder wohldefiniert noch beherrschbar sind, weil keine Grundlage für ihre Erfassung existiert.

Die Vorteile dieses Ansatzes ähneln denen der Betrachtung isolierter Anwendungen auf dedizierten Anlagen in einigen Arbeiten im Bereich der Parallelverarbeitung, die dort dem Zweck dienen, unerwünschte Einflüsse und nicht kontrollierbare Störfaktoren von außen zu eliminieren. Während diese Vorteile dort aber auf in sich abgeschlossene, kurzlebige Anwendungswelten beschränkt bleiben, werden sie in MoDiS auf ein langlebiges System, das zahlreiche Problemlösungskomplexe beinhaltet und sich vielfältig weiterentwickeln kann, übertragen. Integration über den Raum bietet dabei neue Möglichkeiten für systemweites verteiltes Ressourcenmanagement. Ohne Information über globale Zusammenhänge muß ein verteiltes Managementsystem zwischen zwei möglichen Strategien wählen. Erstens könnte es versuchen, die Realisierung einzelner Anwendungen ohne Rücksicht auf ebenfalls in Ausführung befindliche andere Anwendungen zu optimieren, was explizit oder implizit geforderte Fairneß verletzten könnte oder zumindest Interessenskonflikte nach sich ziehen würde, die aufgrund der Abwesenheit einer gemeinsamen Basis nicht lösbar wären (s. S. 31). Zweitens könnte sich das Management darauf beschränken, bottom-up orientiert, die Last (Speicher und Prozessor) auf den Rechnern ungeachtet der Abhängigkeiten zwischen Berechnungen zu balancieren. Diese Vorgehensweise ignoriert spezifische Bedürfnisse von Anwendungen und führt in einer

verteilten Umgebung in vielen Fällen zu extrem schwacher Leistung. Keine dieser beiden Strategien erscheint erfolgversprechend und eine Kombination beider Möglichkeiten ist dringend erforderlich. Mit der Integration über den Raum scheint dieses Ziel erreichbar zu sein.

Die Voraussetzungen für die Realisierung dieses Gesamtsystem-Ansatzes sind Homogenität der Beschreibung und Konzepte für die inkrementelle Konstruktion des Systems, d. h. insbesondere eine Antwort auf die Frage, wie neue Generatoren in das System eingebracht bzw. existierende modifiziert werden können. Die erste Voraussetzung wird in MoDiS durch die INSEL-Sprachbasierung gewährleistet. Letztere wird weiter unten im Rahmen der vorliegenden Arbeit geschaffen.

## 3.4 Top-Down Konkretisierung

Die dritte tragende Säule des MoDiS-Ansatzes neben Sprachbasierung und Gesamtsystem-Ansatz ist die top-down orientierte Konstruktion des Managements, für den gerade der sprachbasierte Ansatz prädestiniert ist. Top-Down Konkretisierung bedeutet, daß die Eigenschaften des mit der Sprache INSEL spezifizierten *abstrakt verteilten* Systems, durch sukzessive Hinzunahme konkreterer Eigenschaften, schrittweise verfeinert werden, bis das System letztendlich technisch und damit *physisch verteilt* realisiert ist. Als erster Schritt im Zuge dieser vielschichtigen Transformation wurde im Projekt MoDiS, orientiert an den Anforderungen der INSEL-Konzepte, auf hohem Abstraktionsniveau ein reflektives, dezentrales Management-Modell für die verteilungsopake, skalierbare und adaptive Verwaltung der physisch verteilten Ressourcen von NOW-Konfigurationen entwickelt [Win96b, Rad95, PE97, EP99]. Wesentliches Merkmal dieser Architektur ist das Aufspalten der Gesamtaufgabe auf multiple autonom und kooperativ agierende Manager. Ausgehend von dieser Grobarchitektur wird das Ressourcenmanagement durch schrittweise Konkretisierung der abstrakten Manager konstruiert. Hierzu gehört die Präzisierung der Aufgaben der Manager und später die Festlegung von Realisierungsdetails, wie z. B. Realisierung von Manager als aktive oder passive Einheiten. Die bei Beginn der vorliegenden Arbeit in ihren Grundzügen ausgearbeitete MoDiS-Manager-Grobarchitektur dient als Ausgangsbasis für die Konstruktion des flexiblen und integrierten V-PK-Ressourcenmanagements und wird dementsprechend in den folgenden Absätzen erklärt.

Auf Grundlage der Lebenszeitabhängigkeit ($\epsilon$-Struktur) werden Komponenten des INSEL-Systems zu Akteursphären zusammengefaßt (s. Abb. 3.5).

**Definition 3.11 (Akteursphäre)**
*Sei $x \in X_t$ ein Akteur; Die Menge*

$$AS(x) = \{y \in X_t : ((y,x) \in \epsilon_t^* \wedge \neg(y \text{ ist Akteur oder K-Order})) \vee ((y,x) \in \sigma_t \wedge y \text{ ist K-Order})\}$$

*heißt Akteursphäre von $x$.*

Eine Akteursphäre umfaßt genau einen Akteur und seine lebenszeitabhängigen passiven Komponenten. Mit dem Konzept der Akteursphären wird die Betrachtung des aus DA-Komponenten bestehenden, strukturierten INSEL-Systems auf strukturell abhängige Akteursphären vergröbert (siehe Abbildung 3.5).

Akteursphären sind die wesentlichen Einheiten des Managements. Jeder Akteursphäre wird genau ein **abstrakter Manager** oder kurz **Manager** beigestellt. Auf diese Weise entsteht eine reflektive verteilte Managerarchitektur. Zum einen spiegeln sich die Berechnungen

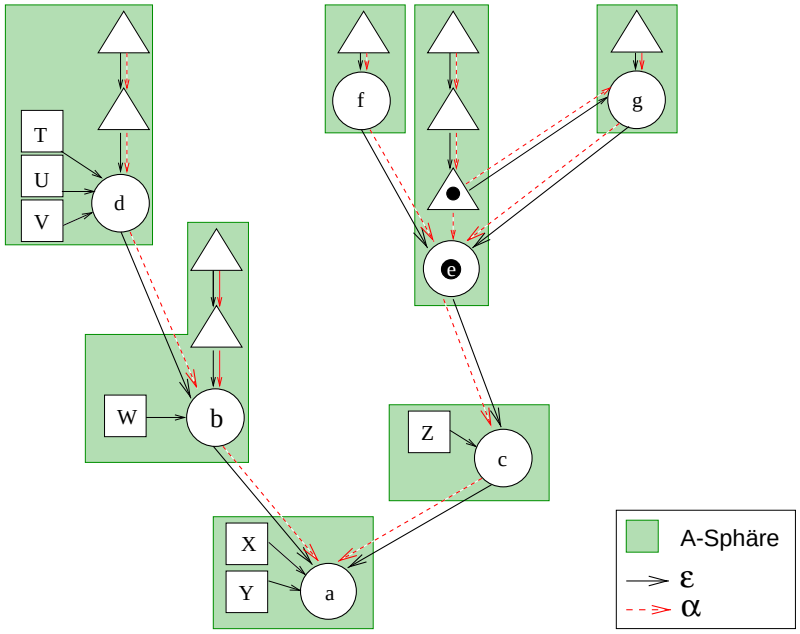


Abbildung 3.5: Partitionierung des Systems in Akteursphären

der Akteure in Zustandsänderungen der Manager wider und zum anderen existiert keine zentrale Verwaltungsinstanz. Die Zahl der Manager und Abhängigkeiten zwischen den Managern leitet sich unmittelbar aus der abstrakten Struktur des Systems ab. Zu diesem Zweck werden abstrakte Manager ebenso wie ihre assoziierten Akteure dynamisch erzeugt und aufgelöst. Jeder Manager hat die Aufgabe, spezifisch für seine assoziierte Akteursphäre, alle Ressourcen zu verwalten, die benötigt werden, um die Berechnung des Akteurs voranzutreiben.

**Definition 3.12 (Manager-Assoziation)**
*Sei $A_t$ die Menge der Akteure des System $\mathcal{S}_t$ und $M_t$ die Menge der Manager von $\mathcal{S}_t$;*

$$man : A_t \rightarrow M_t; \quad man(a) = m_a; m_a \text{ ist der } a \text{ beigestellte Manager}$$

*Die Abbildung man, die zu jedem Akteur seinen assoziierten Manager liefert, ist bijektiv.*

Neben grundlegenden Aufgaben, wie die Allokation ausreichender Mengen an Speicher für die Komponenten der Akteursphäre, haben Manager bei Bedarf auch die Aufgabe, akteursphärenspezifische Zugriffsrestriktionen durchzusetzen, Konsistenz replizierter Komponenten zu gewährleisten oder für Lastausgleich zu sorgen. Die Interaktion zwischen der abstrakt in INSEL formulierten Problemlösung und den Managern ist aus Sicht der Problemlösung opak. Die verteilte Verwaltung von Ressourcen erfolgt somit systemintegriert und automatisiert durch die Manager des Systems. Um Konflikte zu lösen und globales Ressourcenmanagement zu ermöglichen, kooperieren Manager. So sollten gemäß diesem Modell beispielsweise Kellerüberläufe und -kollisionen, die in Folge konkurrenter Allokationen verschiedener Manager möglich sind, durch Kooperation zwischen den betreffenden Managern gelöst werden. Die Menge der Manager ist zu diesem Zweck selbst strukturiert. Die Lebenszeitabhängigkeit zwischen den Akteuren, die auf der $\pi$-Struktur fußt, wird zu diesem Zweck auf die Managerstruktur $\theta$ transferiert.

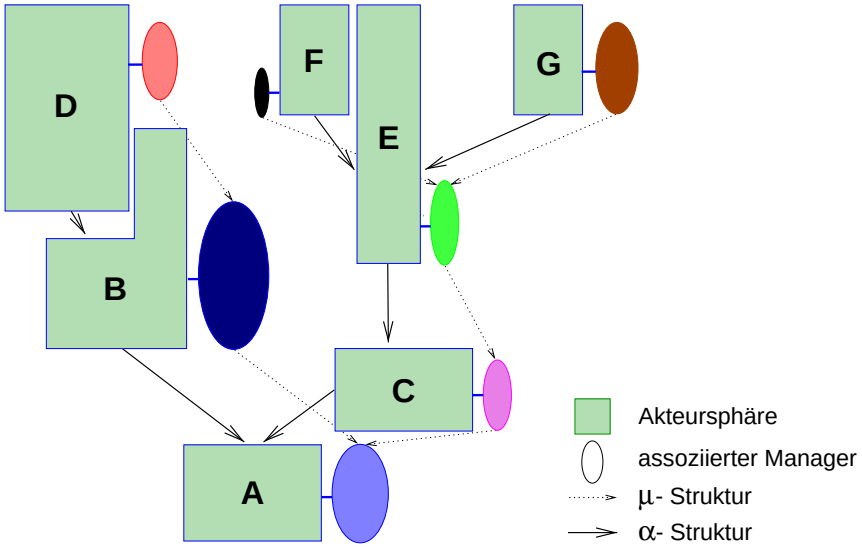


Abbildung 3.6: Akteuersphären und assoziierte Manager

**Definition 3.13 ($\theta$-Struktur)**
*Seien $M_t$ und $A_t$ wie in Def. 3.12 festgelegt und $m, n \in M_t$;*

$$(m,n) \in \theta_t \Leftrightarrow \exists a,b \in A_t : (a,b) \in \epsilon_t \wedge man^{-1}(m) = a \wedge man^{-1}(n) = b$$

*$b$ wird* Vater *von $a$, bzw. $a$* Sohn *von $b$ genannt.*
*$\theta_t^*$ ist die transitive Hülle von $\theta_t$*

Der von $\theta$ aufgespannte Baum bestimmt die primären Kooperationskanäle der Manager. Das heißt, um einen Konflikt zu lösen, kontaktieren Manager primär ihren Vater und ihre Söhne entsprechend der $\theta$-Struktur.

Offensichtlich benötigen Manager Wissen über die INSEL-Problemlösungen, um sich dynamisch an die Anforderungen der Akteursphären anpassen zu können. Im Projekt MoDiS wird beabsichtigt, diese Information überwiegend aus der Analyse der strukturellen Abhängigkeiten sowohl zum Zeitpunkt der Übersetzung als auch durch Monitoring während der Ausführung zu gewinnen. Die Extraktion und der Transport geeigneter Information zwischen den Instanzen, die für die Realisierung von Managern eingesetzt werden, ist für das gesamte Ressourcenmanagement essentiell und wird weiter unten als ein wesentlicher Aspekt der Integration des Managements ausgearbeitet.

Für das zu konstruierende V-PK-Ressourcenmanagement ist zu beachten, daß es sich bei den Managern um ein Konzept auf hohem Abstraktionsniveau handelt. Die intuitive Vorstellung von Managern als Objekte, die zur Laufzeit mit herkömmlichen Bindertechniken an Akteure gebunden werden, ist kontraproduktiv. Eine solch rigide Implementierung führt in erster Linie zu einem hohen konstanten lokalen Verwaltungsaufwand $l$, der wie in Abschnitt 2.2.2 dargestellt, auch durch effizientes verteiltes Management kaum kompensiert werden kann. Bereits die effiziente Realisierung sehr leichtgewichtiger Akteure für feingranulare, parallele Teilberechnungen würde auf diese Weise scheitern und das Problem der Festlegung von Akteuren mit adäquater Granularität auf die Ebene der Problemlösungen verlagern. Dies

würde bereits einen Verlust an Opazität bedeuten, d. h. zusätzliche Komplexität bei der Spezifikation von INSEL-Problemlösungen und somit Verlust von Qualität. In Vorgriff auf die spätere flexible Konkretisierung dieser Architektur stellt Abbildung 3.6 Manager bereits als Individuen mit unterschiedlicher Granularität und variierenden Eigenschaften dar. Ein einfaches Beispiel: falls eine Analyse von $\lambda_t$ erweitert um DE-Komponenten ergibt, daß für einen Akteur $a \in A_t$ keine Generatoren für Zeiger in $L_0(a)$, so muß der Manager $m_a = man(a)$ nicht für die Verwaltung von Halden vorbereitet werden. Die von $m$ geforderten Fähigkeiten für Speichermanagement können dann deutlich vereinfacht werden. Ähnliche Möglichkeiten können sich aus der Abschätzung der $\kappa$-Struktur basierend auf eine Analyse von $U(a)$ zum Zeitpunkt der Übersetzung ergeben. Falls $a$ keine K-Order-Aufträge in seiner Berechnung erteilt, so können die entsprechenden Kommunikationseinrichtungen in $m_a$ ausgelassen werden, was zu einer wesentlich leichtgewichtigeren und somit effizienteren Realisierung von $m_a$ führt.

Entscheidend für die Leistungsfähigkeit dieser Architektur ist somit der Grad der Angepaßtheit, der bei der Produktion und weiteren dynamischen Anpassung der Manager erzielt wird. Um dieser Anforderung entsprechen zu können, wird ein Repertoire alternativer Realisierungen von Managern benötigt, aus dem sich flexibel und dynamisch anpaßbare Manager konstruieren lassen. Die Konzeption und systematische Realisierung dieser Flexibilität ist neben dem Aspekt der Raum-/Zeitintegration ein Ziel dieser Arbeit.

# Kapitel 4

# Stand der Forschung

Als Abschluß der Problemanalyse und der Klärung der Ausgangsbasis für die Konstruktion eines leistungsfähigen, integrierten V-PK-Managements, werden in diesem Kapitel einige weitere ausgewählte Projekte aus den Bereichen verteiltes Ressourcenmanagement bzw. Erweiterbarkeit diskutiert. Die Breite des Themengebietes – integriertes Management erweiterbarer Systeme in verteilten Umgebungen – zwingt zur Auswahl, so daß nur einige wichtige Ansätze skizziert und mit den Begriffen aus Kapitel 2 schlaglichtartig kategorisiert werden können.

Das Ziel dieser Recherche ist dessen ungeachtet zum einen die Einordnung und Abgrenzung der in der vorliegenden Arbeit durchgeführten Entwicklungsschritte, und auf der anderen Seite das Aufsammeln von Erfahrungen als Vorbereitung der noch zu treffenden Entscheidungen. Hierzu gehört besonders die Evaluation vorangegangener prototypischer Realisierungen der MoDiS-Architektur.

## 4.1 Verteiltes Ressourcenmanagement

Bezüglich des Ressourcenmanagements in verteilten Systemen werden in diesem Abschnitt zunächst einige additive Erweiterungen für paralleles und verteiltes Rechnen betrachtet, da zweifellos der überwiegende Anteil aller Bemühungen verteiltes Rechnen zu unterstützen dieser Klasse angehört. Anschließend werden einige Ansätze diskutiert, die über das Hinzufügen zu bestehenden Systemen hinausgehen und entweder neu konstruieren, adaptieren oder sich einer gesamtheitlich integrierenden Vorgehensweise annähern.

### 4.1.1 Additive Erweiterungen

Bei additiven Erweiterungen für verteiltes Rechnen handelt es sich überwiegend um Bibliotheken, die mit den Produkten des Übersetzers bei Bedarf gebunden werden. Bereits existierende Ausführungsumgebungen zentraler Systeme werden damit für einige spezielle Programme im Hinblick auf verteilte Ausführung erweitert. Die Beliebtheit dieser Vorgehensweise beruht auf Kompatibilität mit existierenden Systemen, geringen bzw. kontrollierbaren Aufwand und der Tatsache, daß ein großer Teil der gewünschten Verbesserungen kurzfristig tatsächlich so erreicht werden kann, auch wenn dazu Kompromisse, wie z. B. Verlagerung von Managementproblemen auf die Nutzer der Erweiterung, eingegangen werden müssen. Die Integration der notwendigen neuen Ausdrucksmittel in die Sprache findet in der Regel lediglich durch die Hinzunahme der Namen vordefinierter Komponenten statt (siehe 2.3.1.1). Ebenso proble-

matisch ist in aller Regel die Komposition der neuen Fähigkeiten mit den Maßnahmen der gänzlich unmodifiziert übernommenen Instrumente Übersetzer, Binder und Kern. Additives Vorgehen leidet dementsprechend stets darunter, daß die Abhängigkeiten der Erweiterung zu vorhandenen Konzepten entweder nicht ausreichend bekannt sind oder bewußt ignoriert werden.

#### 4.1.1.1 Plattformen für verteiltes Rechnen: DCE und CORBA

Wichtige Beispiele für umfangreiche additive Erweiterungen sind Plattformen für verteiltes Rechnen wie DCE [Ope92, Ope96] (Distributed Computing Environment) der OSF[1] (Open Software Foundation) oder Produkte basierend auf der CORBA (Common Object Request Broker Architecture) Spezifikation [OMG95, OMG97] der OMG (Object Management Group). DCE wurde entworfen, um verteiltes Rechnen unabhängig von der Sprache, dem Übersetzer und dem zugrundeliegenden Betriebssystemkern mit einem prozeduralen Programmiermodell zu unterstützen. Angeboten werden RPC (Remote Procedure Call) zur Erleichterung der Kommunikation zwischen Clients und Server, Authentifikation, verteilte Benutzerkonten, global eindeutige Namensräume mit verteilten Verzeichnisdiensten, verteilte Dateisysteme und Threads. CORBA betont den Aspekt der räumlichen Verteilung weniger stark und konzentriert sich noch stärker auf die Überwindung der Heterogenität verschiedener Sprachen und Betriebssysteme speziell in objektorientierten verteilten Umgebungen, um ein hohes Maß an Interoperabilität zu erzielen. Das Kernstück einer CORBA-Implementierung ist ein Object Request Broker (ORB), der verteilte Methodenaufrufe realisiert und notwendige Konvertierungen durchführt.

**Qualitative Bewertung**

Die Konstruktion eher grobgranularer Client/Server-Systeme profitiert von diesen Umgebungen, da die Programmierung verteilter Anwendungen von den angereicherten Fähigkeiten der Ausführungsumgebung signifikant entlastet wird und sich wesentlich stärker auf abstrakte Eigenschaften der Problemlösung als auf Details der physischen Verteilung, wie Socket-Adressen, TCP/IP, etc. konzentrieren kann [SV95]. Spätes Binden und die Trennung zwischen Spezifikation und Implementierung im Falle von CORBA ermöglicht zudem in engen Grenzen dynamische Erweiterung und Adaption des laufenden Systems. Zweifellos können mit DCE und CORBA einige der qualitativen Ziele aus 2.2.1, Ubiquität, Mobilität, Telearbeit, etc., erreicht werden.

Allerdings treten auch im Beispiel CORBA und DCE Mängel der additiven Vorgehensweise deutlich zu Tage. Beide Umgebungen fügen heterogen zusätzliche Konzepte ein, wodurch die Programmierung erschwert wird und die Opazität verloren geht. CORBA führt unter anderem Objektreferenzen und ein aufwendiges Ausnahmebehandlungskonzept ein. DCE bietet Dienste zur Erzeugung von Threads an, die primitive passive Komponenten ohne Abstimmung mit der Sprache oder dem Übersetzer mit Rechenfähigkeit anreichern. Offensichtlich kann der Übersetzer der gewählten Sprache dies bei seiner Analyse und anschließender Synthese deshalb nicht berücksichtigen, was u. a. zu ineffizienten Parameterübergaben führt und inkompatible Optimierungsentscheidungen nach sich ziehen kann. Ferner muß ein Programmierer selbst in einfachsten Fällen explizit für Synchronisation sorgen. CORBA kennt demgegenüber gar kein Aktivitätsträgerkonzept, so daß parallele - und kooperative Algorithmen

[1] inzwischen in *OpenGroup* übergegangen

nur schwerlich auf das Programmiermodell abgebildet werden können. Die einzigen Hilfsmittel sind *concurrency services*, mit deren Hilfe Sperren realisiert werden können, die bei Bedarf Eigenschaften von Transaktionen besitzen können. Weitere Erschwernisse entstehen durch die Notwendigkeit mit Verzeichnisdiensten (*directory servers* in OSF/DCE) operieren zu müssen, um Dienste zu suchen bzw. zu registrieren sowie durch zusätzlich zu lernende und einzusetzende Schnittstellendefinitions-Sprachen (DCE-IDL, OMG-IDL). Die erreichte Opazität der Verteilung ist dennoch selbst in CORBA gering, da sich die Spezifikation entfernter Zugriffe deutlich von lokalen unterscheidet. Bei entfernten Zugriffen müssen zuerst explizit Client-Stubs gebunden werden. Insgesamt belasten diese Plattformen demnach immer noch stark den Programmierer verteilter Anwendungen, da dieser beträchtliche Zeit für das Erlernen der zusätzlichen Konzepte und ihrer Kompositionalität mit den bisherigen investieren muß. Heterogenität ist eine ungewollte Fehlerquelle und reduziert gleichfalls die Qualität und Effizienz von Systemen. Sie kann durch eine konzeptionell homogene Umgebung, wie sie INSEL zur Verfügung stellt, vermieden werden.

**Quantitative Bewertung**
Besonders im Falle von CORBA zeigen sich auch die quantitativen Nachteile mangelnder Integration. In [SV95] wird darauf hingewiesen, daß Methodenaufrufe zwischen Objekten, die sich ohnehin in dem gleichen Adreßraum eines UNIX-Prozesses befinden, durch das Eliminieren von Indirektionen innerhalb des ORB optimiert werden könnten. Transformationen der Darstellung und das aufwendige Verpacken übergebener Parameter könnten überdies eingespart werden, falls bekannt ist, daß die Hardware, auf der die interoperierenden Objekte realisiert sind, identische Eigenschaften besitzt. Voraussetzung für Verbesserungen der quantitativen Eigenschaften dieser Art wäre, daß dem ORB Information über die Begebenheiten von hohem bis niedrigem Abstraktionsniveau zur Verfügung steht. In Konsequenz sollte ein ORB nicht als additive Erweiterung einer bestehenden Plattform sondern durch ein integriertes Transformations- und Interpretations-Instrumentarium realisiert werden. Nur dann steht den Entscheidungsträgern die notwendige Information zur Verfügung. Diese Vorstellung widerspricht allerdings klar den ursprünglichen Zielsetzungen von CORBA.

#### 4.1.1.2 Multi-Threading und Verwaltung multipler Keller

Die im Zusammenhang mit DCE genannten Threads werden auch verstärkt von zentralen Systemen, basierend auf dem POSIX Standard 1003.1c [IEE95], zur Realisierung feingranularer Parallelität angeboten und eingesetzt. Wenngleich dieses Konzept, abgesehen von DCE Threads, nicht im Zusammenhang mit verteiltem sondern lediglich parallelem Rechnen steht, so ist es dennoch ein gutes Beispiel für die derzeit gängige Entwicklung von Betriebssystemkonzepten. Threads wurden bottom-up, orientiert an Prozessoren, Speicher und Prozessen entworfen und werden meist additiv realisiert. Eine Abstimmung mit den vorhandenen Konzepten, z. B. blockierende Ein-/Ausgabe, erfolgt nicht, was typische Probleme, u. a. *two-level scheduling* [ABLL92], nach sich zieht.

**Wechselwirkungen mit der Speicherverwaltung**
Besonders deutlich wird die mangelnde Integration bei der Speicherverwaltung. Die Prägung des virtuellen Adreßraumes (VA) bei genau einem Ausführungsfaden sieht das entgegengesetzte Wachsen und Schrumpfen der linearen Bereiche Keller und Halde vor. Kollidieren Keller

und Halde, so ist der VA erschöpft und ein irreparabler Fehlerzustand erreicht. Da bei mindestens 32 Bit weiten VA die physische Speicherkapazität von Arbeits- und Hintergrundspeicher in aller Regel vor dem Eintritt dieser Situation erschöpft wäre, ist dieses Verhalten akzeptabel. Bei multi-threaded Umgebungen gilt diese Annahme nicht. Keller verschiedener Threads können miteinander oder mit der Halde kollidieren bevor der VA auch nur annähernd ausgeschöpft ist. Typische multi-threading Umgebungen bieten dennoch keinerlei Unterstützung bei der Speicherverwaltung. Es obliegt dem Programmierer, Größe und Position der Keller korrekt zu wählen. Da die erforderliche Kellergröße aufgrund von Rekursion und dynamischer Strukturen in der Regel a priori nicht bekannt ist, wird meist ein grob abgeschätzter Vorgabewert verwendet. Das Spektrum entsprechender Vorgabewerte reicht von 16 KByte in Mach 3.0 [ABG+86] bis 1 MByte unter SUN Solaris 2.5.1 [Sun95b] und verdeutlicht die Willkürlichkeit dieser Entscheidung. Keine dieser Abschätzungen garantiert die Suffizienz der Kellergröße. Ein möglicher Überlauf in einen fremden Keller kann zur Zerstörung von Daten fremder Keller und Halden führen. Da ferner die Forderung nach Linearität der Kellerbereiche im Zuge der Einführung von Threads nicht revidiert wurde, können kollidierte Keller auch nicht an anderen Orten im VA fortgesetzt werden, wodurch eine ausschöpfende Nutzung des VA ausgeschlossen ist. Besonders negativ wirkt sich dieses Defizit auf die Möglichkeit aus, extrem große, 64 Bit weite Adreßräume moderner Architekturen zu nutzen, um gesamte Systeme innerhalb eines gemeinsamen Ein-Adreßraumes (SASOS - Single Address Space Operating Systems) zu realisieren [AS96, GHR98].

**Intuitive Lösungsansätze**

SUN Solaris versucht dieses Problem durch gesperrte Seiten am Ende allozierter Thread-Keller zu lindern. Bei einem Überlauf entsteht eine Unterbrechung, auf die der Kern reagiert. Diese Technik ist leicht zu realisieren, kompatibel und unabhängig von der restlichen Instrumentierung. Wenngleich Überläufe effizient detektiert werden, so ist auch auf diese Weise keine Behebung des Fehlers möglich. Überläufe bleiben unerkannt, bis auf ein Objekt auf einer gesperrten Seite zugegriffen wird, selbst wenn zuvor bereits Adressen von Objekten auf der gesperrte Seite verwendet, dupliziert und weitergereicht wurden. Zum Zeitpunkt der Detektion müßten zur Fehlerbehebung Speicher- und Registerinhalte global durchsucht und korrigiert werden, was unrealistisch ist. Der Entwurfsfehler liegt in diesem Fall an der Entscheidung, Fehlererkennung und -behebung anstelle Fehlerverhinderung zu betreiben, was auf eine empirische Konstruktionsmethode in Kombination mit trial & error zurückzuführen ist. Ebenso kontraproduktiv ist die Vorstellung, den Kellerspeicher so groß zu wählen, daß die Wahrscheinlichkeit für einen Überlauf gering ist. Diese Vorstellung erhöht die Kosten für die Erzeugung und Auflösung von Threads und widerspricht der ursprünglichen Zweckbestimmung leichtgewichtiger Aktivitätsträger. In Projekten wie Distributed Filaments [FLA94, EAL93] und [PE96, DW93] wird völlig entgegengesetzt versucht, den Nutzen von Threads durch Minimierung ihrer Erzeugungs- und Auflösungskosten zu verbessern, basierend auf der Erkenntnis, daß Threads zu schwergewichtig und damit zu inflexibel für die Spezifikation von Parallelität auf hohem Abstraktionsniveau sind. Programmierer sind nach wie vor gezwungen, Festlegungen über die Anzahl von Threads und damit wichtige Managemententscheidungen mit gravierenden Auswirkungen [CBZ95] zu treffen, die nicht zur abstrakten Problemlösung gehören, präzise Kenntnis über die Hardware erfordern, und als solche eigentlich automatisiert durch das Managementsystem erfolgen sollten (s. 2.3.1.3).

**Fortgeschrittene Lösungsansätze**
Weitergehende Lösungsansätze sind bisher selten, da sie umfangreiche Anpassungen der gesamten Speicherverwaltung vom Übersetzer bis zum Kern erfordern und großen Aufwand bedeuten. In [HL93] wird das Problem multipler Keller mit einem konzeptionellen *cactus stack* gelöst, der als pro Prozessor *meshed stack* realisiert ist. Als ein Beispiel für das Verfehlen der Angebote bei top-down Vorgehensweisen, wurde das Konzept allerdings mittels ineffizientem Aufsammeln (garbage collection) nicht mehr benötigter Aktivierungssegmente innerhalb des meshed stack implementiert. Concurrent Oberon [ARD97] bezieht den Übersetzer in die Lösung des Problems mit ein, der Code zur dynamischen Prüfung des Kellerzeigers auf ein festes Limit von 128$k$Byte Kellergröße in das Zielprogramm generiert. Kellerüberläufe von „*Active Objects*“ werden wirksam verhindert und der Verbrauch an physischen Speicher innerhalb des Limits ist adaptiv. Inkonsequenterweise wird die Integration des Übersetzers jedoch nicht genutzt, um die Forderungen nach Linearität der Keller zu revidieren, wodurch der VA auch in Concurrent Oberon nicht ausgeschöpft werden kann und bereits Keller $> 128k$ unlösbare Fehlerzustände darstellen.

**Schlußfolgerungen**
Es bleibt festzustellen, daß ein Konzept, wie das der Threads, vielfältige Abhängigkeiten zu anderen Konzepten innerhalb eines Systems besitzt, die nicht ignoriert werden dürfen. Qualitative und quantitative Mängel können nur durch gesamtheitlich integrierte Planung und Realisierung verhindert werden. Dies ist bislang aufgrund der primitiven Entwurfsverfahren und Realisierungswerkzeuge kompliziert und aufwendig. Bei der Realisierung der MoDiS-Architektur wird in der vorliegenden Arbeit ein Ein-Adreßraum-Ansatz verfolgt, der eine Lösung für das Problem der Verwaltung multipler Keller dringend erforderlich macht. In Kapitel 8 wird das integrierte Management-Instrumentarium genutzt, um dynamisch erweiterbare Keller effizient zu realisieren und damit eine adaptive und ausschöpfende Nutzung des VA zu ermöglichen.

#### 4.1.1.3 Nachrichtenorientiertes verteiltes Rechnen: PVM

Vermutlich ist PVM (Parallel Virtual Machine) [SGDM94, Sun93] die bislang am weitesten verbreitete und am häufigsten eingesetzte Erweiterung für verteiltes Rechnen. Bei PVM handelt es sich im wesentlichen um eine Bibliothek, die es erleichtert in einem heterogenen UNIX-Netz Prozesse zu starten, zu beenden und Nachrichten zwischen diesen auszutauschen. PVM-Implementierungen existieren für ca. 30 verschiedene Architekturen, begonnen bei Intel 80386 PCs über SUN, SGI, HP Arbeitsplatzrechner bis hin zu COMA Multiprozessoren wie der KSR-1. Ebenso umfangreich sind die Werkzeuge und Ergänzungen, die für PVM entwickelt und in zahlreichen Veröffentlichungen und speziellen PVM-Tagungen [BDLS96] vorgestellt werden. Der große Erfolg von PVM beruht auf dem Mangel an verfügbaren Alternativen, einer eindeutig quantitativen Zielsetzung für einen speziellen Einsatzbereich sowie Beschränkung auf rudimentäre Fähigkeiten, die effizient realisiert werden können. Verteilte Speicher- und Rechenkapazitäten können zur Durchführung aufwendiger Berechnungen im technisch-wissenschaftlichen Bereich mit PVM einfacher genutzt werden, als dies mit den Standardfähigkeiten von UNIX möglich wäre, ohne daß PVM selbst einen signifikanten lokalen oder verteilten Mehraufwand während der Ausführung von PVM-Programmen erzeugt. PVM wurde klar durch bottom-up Virtualisierung, orientiert an den physischen Eigenschaften von NOW-Konfigurationen sowie Prozessen und Sockets,

in UNIX-Betriebssystemen entwickelt. Das Programmiermodell kennt außer schwergewichtige Prozesse keine parallelen Aktivitätsträger. Wenngleich experimentelle DSM-Erweiterung für PVM existieren [Dum95, JKW95b], sieht PVM ausschließlich Nachrichtenaustausch als Kooperationskonzept zwischen Prozessen vor. Dementsprechend existiert auch kein gemeinsamer Adreßraum mit global eindeutigen Identifikatoren. PVM setzt auch keine Opazität durch. PVM-Prozesse können explizit auf Stellen plaziert werden und Details der Hardwarekonfiguration, wie die Anzahl der verfügbaren Stellen, werden von PVM-Programmen erfragt und zur Optimierung der Ausführungszeiten eingesetzt. Die Zuordnung von Managementaufgaben ist eindeutig. Entwickler von PVM-Programmen, Benutzer und PVM-Programme müssen nahezu alle wichtigen Entscheidungen, die im Zusammenhang mit der Verteilung stehen, selbständig lösen. Dazu gehört die Partitionierung der Daten und Berechnungen einer parallelen Problemlösung in geeignete grobgranulare Prozesse, die Vor- und Nachbereitung von Nachrichten inkl. Adreßtransformationen und vieles mehr.

**Bewertung**
Die Entwicklung von PVM Programmen ist kompliziert und erfordert große Anstrengungen, damit parallele Problemlösungen auf dieses Programmiermodell abgebildet werden können. Der Programmtext der in [FSGF97] mittels PVM realisierten Schaltungssimulation implementiert überwiegend Managementfunktionalität, wie die Transformation von Speicherinhalten in Nachrichten, Synchronisation mittels Nachrichten sowie Konvertierung von Identifikatoren, die nicht zum abstrakten Problemlösungsverfahren, sondern ausschließlich der konkreten Realisierung gehören. Effizienz wird mit diesen Anstrengungen durchaus erreicht. Für große Programmsysteme, an die zusätzlich qualitative Forderungen gestellt werden, eignet sich diese Vorgehensweise allerdings kaum, so daß PVM und vergleichbare Ansätze wie MPI [Ano93] wohl ausschließlich für Sonderfälle in dem genannten technisch-wissenschaftlichen Bereich geeignet sind. Erfahrungen mit *message passing* Ansätzen dieser Art demonstrieren eindeutig zum einen die größere Effizienz der Kooperation mittels expliziter Nachrichten gegenüber gemeinsamen Speicher [LDCZ97], zum anderen allerdings auch, daß einige Klassen von Problemen nur sehr schwer mittels Austausch von Nachrichten spezifiziert werden können und gemeinsamer Speicher dringend zusätzlich benötigt wird.

#### 4.1.1.4 Sonstiges

Neben den oben angeführten verbreiteten Erweiterungen für verteiltes Rechnen existiert mit dem Bereich DSM eine ganze Klasse von Ansätzen, die ebenfalls überwiegend durch partielle Erweiterung versuchen, verteiltes Rechnen zu ermöglichen. Auch dort wird mit partieller Erweiterung keine zufriedenstellende Balance zwischen qualitativer - und quantitativer Leistung erreicht und die Erfolge müssen bislang als dürftig bezeichnet werden [Car98]. Ansätze wie *Glunix* [GPRV98] und DIPC [Byn99] versuchen die Fähigkeiten von UNIX-Systemen durch Änderungen am Kern für opakes verteiltes Rechnen zu erweitern. DIPC modifiziert die UNIX IPC Dienste fork, message queue, memory mapping, Semaphore, etc. des Linux Kerns in einer Weise, die es erlaubt, sie auch über Rechnergrenzen hinweg einzusetzen. Die Zielsetzung bzw. der Gewinn, der dadurch erzielt wird, ist allerdings unklar. Die Leistungsfähigkeit gegenüber Systemen wie PVM ist gering und ihr Einsatz ähnlich schwierig. Aus der entgegengesetzten Richtung, Sprachen und Übersetzer, sind Ansätze, wie *Distributed C* [Ple93] bemüht, die Programmierung paralleler und verteilter Systeme zu erleichtern, stoßen allerdings an Grenzen, wo unmodifiziert übernommene Laufzeitsysteme und Kerne nicht die gewünsch-

ten Angebote zur Verfügung stellen. Schwierigkeiten bei der Prozeßmigration aufgrund enger Stellenbindungen (u. a. Dateideskriptoren), ist ein typisches Beispiel.

### 4.1.2 Verteilte Betriebssysteme

#### 4.1.2.1 Kern-orientierte Entwicklungen

Der Bereich der Betriebssysteme beschränkt sich bisher auf die Neukonstruktion oder Anpassung von Kernen, um eine bessere Grundlage für die Ausführung verteilter Programme anbieten zu können. In Projekten wie Amoeba [MvRT+90], Mach [ABG+86] und Sprite [OCDN88] wurden leistungsfähige Stellenkerne mit neuen und verbesserten Eigenschaften in physisch verteilten Umgebungen entwickelt. In Sprite wurde ein log-strukturiertes verteiltes Dateisystem entwickelt, das größere Zuverlässigkeit und durch exzessives Client- und Sever-Caching hohe Leistung bietet. In Mach wurde u. a. Speicher von Rechenfähigkeit getrennt, Kooperation zwischen Ausführungsfäden mittels *Ports* systematisiert und das System als Mikrokern strukturiert und flexibilisiert. Während Sprite Benutzer starr mit Arbeitsplatzrechnern assoziiert, sieht Amoeba eine Kombination von Terminals, Prozessor- und Speicherpools vor. Terminals sind die flexiblen Zugriffstore auf die Rechen- und Speicherressourcen der Pools. In Amoeba werden, ähnlich wie in Mach, alle Komponenten des Systems (Prozeß, Dateien, ...) als Objekte mittels *capabilities* identifiziert, die Ports für Zugriffe beinhalten. Ports sind logische Adressen, die nicht an Stellen gebunden sind. Benutzerprozesse sind ferner nicht in der Lage die physische Lokalisation ihrer Ausführung zu ermitteln [DKOT91]. Amoeba erreicht somit ein hohes Maß an Opazität und Flexibilität. Objekte unterschiedlicher Art können flexibel migriert werden, was eine wichtige Voraussetzung für das Erreichen der mit verteilten Systemen anvisierten quantitativen und qualitativen Ziele ist (Lastausgleich, Mobilität, Fehlertoleranz, ...). Konsequent wurden im Projekt Amoeba benötigte Eigenschaften zielgerichtet und losgelöst von Bestehendem neu konstruiert, siehe z. B. FLIP [KvRvST93].

**Bewertung**
Mit wenigen Ausnahmen wie der Sprache Orca [BKT92] (Amoeba) beschränken sich diese und ähnliche Arbeiten im Bereich verteilter Betriebssysteme nahezu ausschließlich auf den Kern, wobei übersehen wird, daß es sich bei dem Kern nur um ein spezielles Instrument des Ressourcenmanagements handelt. Resultat ist, daß sowohl auf Sprite, Mach und Amoeba als auch in ähnlichen Entwicklungen nach Entwicklung der Kernfunktionalität vor dem Aufwand der Neuentwicklung weiterer Komponenten des Systems kapituliert wird und stattdessen sequentielle und zentrale UNIX-Programme und Werkzeuge auf die neue Basis mit geringen Anpassungen portiert werden. Letztendlich entsteht dadurch die Gefahr, daß, wie am Beispiel Mach, das Interesse an diesen Systemen verloren geht, da die Effizienz der aufgesetzten Umgebung für den Benutzer gegenüber herkömmlichen Systemen eher schwächer ausfällt und gleichzeitig keine deutlichen Vorteile erkennbar sind. Vermieden werden kann dies nur, wenn die Konstruktion konsequent gesamtheitlich bis auf Ebene der Sprache weiter betrieben und auch geeignete neue Nutzungsmöglichkeiten zur Verfügung gestellt werden. Eine weitere Gefahr, die aus dem Aufsetzen von Vorhandenem auf das neu entwickelte entsteht ist, daß schon während der Entwicklung, siehe Sprite, Entscheidungen stark von dem Zwang zur Kompatibilität bestimmt sind und den Raum für den kreativen Entwurf notwendiger neuer Konzepte stark einschränken.

#### 4.1.2.2 Gesamtprojekt Comandos

Mit Comandos [Com92, CBH+94] wurde ab Ende der 80er Jahre wohl das bislang umfangreichste Projekt im Bereich der verteilten Betriebssysteme unternommen. Dem von der Europäischen Gemeinschaft im Rahmen von ESPRIT-1 geförderten Comandos-Konsortium gehörten 15 Institution aus sieben europäischen Ländern an, darunter Bull SA (Frankreich), Siemens und die SNI AG (Deutschland), Universität von Dublin (Irland), Chorus (Frankreich), das Frauenhofer Institut und die GMD (Deutschland). Comandos war in seiner Struktur gesamtheitlich ausgelegt und sah die Entwicklung von Programmiersprachen, Entwicklungswerkzeugen, verteiltem persistenten Speicher für Datenobjekte, Aktivitätsträger- und Synchronisationskonzepte [MBWD94] bis hin zu speziellen verteilten Verzeichnisdiensten, Sicherheits- und Administrationswerkzeugen vor. Klassen abstrakter Ressourcen wie z. B. *Cluster, Container, Extends* wurden systematisch entworfen und realisiert, wobei diese Festlegungen bottom-up betrieben wurden, wodurch die Flexibilität bezüglich Anwendungsanforderungen gering ist. Leider wurde auch die Rolle der Instrumentierung nicht neu überdacht, sondern empirisch, d. h. wie gehabt, vereinbart. Das Fundament der Comandos Plattform ist die Comandos Virtual Machine, die auf abstraktem Niveau festgelegt ist und den Rahmen für interoperable Comandos-Implementierungen definiert. Sie beinhaltet Festlegungen bezüglich des Objektmodells inklusive Kooperationsmöglichkeiten und globale Identifikatoren, dem Sicherheitsmodell, Ausführungsmodell sowie Festlegungen eines Transaktionskonzeptes. Von dieser Architektur wurden simultan und separat drei Implementierungen betrieben. Amadeus [C+92] ist eine UNIX-basierte additive Realisierung, COOL [LJP93] wurde basierend auf dem Chorus Mikrokern [The92] entwickelt und bei GUIDE [BBD+91] handelt es sich um eine Mach-basierte Realisierung.

**Bewertung**
Trotz dieser großen Anstrengungen und einiger interessanter Ansätze, wie die Anlehnung verteilten Ressourcenmanagements an mikroökonomische Theorien [CT93], blieb auch der Erfolg von Comandos, abgesehen von neuen Konzepten und Detaillösungen gering. Die ursprünglich anvisierte Praxistauglichkeit und der Einsatz in der Industrie konnte nicht erreicht werden. Die Gründe dafür sind vermutlich in der Größe des Projektes selbst zu suchen. Zum einen fehlt eine klare Zielsetzung des Projektes in Form quantitativer oder qualitativer Anforderungen, was eng mit der bottom-up Vorgehensweise zusammenhängt. Zum anderen haben vermutlich die stark unterschiedlichen Interessen der Mitglieder des Comandos-Konsortiums zu einer ineffizienten Aufspaltung in separate Teilgebiete geführt. So wurden mit C++, Eiffel und der Comandos-Sprache insgesamt drei Sprachen mit unterstützt, um dem Wunsch der Industrie nach Interoperabilität zu entsprechen, anstelle die Chance zu ergreifen, eine neue geeignete Sprache zu entwickeln. Der Effekt davon ist, daß alle drei Varianten aufgrund von Heterogenität unter Kompromissen und mangelnder Integration mit der Werkzeugumgebung und der Ausführungsumgebung leiden. Z. B. produziert der Übersetzer für die eigens geschaffene Sprache Comandos C-Code, der mit einem unmodifiziert übernommenen C-Übersetzer weiter übersetzt wird.

#### 4.1.2.3 Adaptionsfähige Betriebssysteme

Basierend auf der Beobachtung, daß oftmals eine größere Flexibilität und Anpaßbarkeit des Ressourcemanagements an dynamisch sich ändernde Anwendungsanforderungen wünschens-

wert wäre, untersuchen einige Forschungsprojekte mögliche Flexibilisierungsansätze. Eine weitergehende Flexibilisierung relativ zu den weiter oben bereits erwähnten, statisch rekonfigurierbaren, objektorientierten Systemen (z. B. Choices [CI93]) wird mit reflektiven und anpaßbaren Betriebssystemen, z.B. Muse/Apertos (u.a. [YMFT91, Yok92, LYI95, CAK+96]), erzielt. Die Grundidee dieser Ansätze besteht darin, der Anwendungsebene eine Menge von Realisierungsalternativen in Form von Meta-Objekten zur Verfügung zu stellen. Meta-Objekte realisieren unterschiedliche Ausprägungen von Betriebssystemdiensten wie beispielsweise unterschiedliche Schedulingverfahren. Anwendungsprogramme haben Zugriff auf die Schnittstelle der Meta-Objekte und können über Meta-Level-Protokolle Meta-Objekte gezielt auswählen und so eine angepaßte Realisierung erreichen. Damit wird das Verhalten der Anwendung dynamisch zur Laufzeit verändert und angepaßt. Im Gegensatz zu der Flexibilisierung in der vorliegenden Arbeit, werden dynamische Anpassungen nicht automatisiert durch die Ressourcenverwaltung, sondern explizit durch den Anwendungsprogrammierer initiiert, der ein Meta-Objekt nach von ihm selbst festzulegenden Kriterien auswählen muß.

### 4.1.3 Integriertes Management

Die Komplexität von Managementproblemen erfordert oftmals präzise Information und Maßnahmen, die in aufeinander abgestimmten Schritten vollzogen werden. Dies gilt besonders in parallelen und verteilten Umgebungen. Die konventionelle strikte Isolation von Realisierungsinstrumenten steht dem im Weg. In einigen Forschungsarbeiten wird deshalb versucht, die Leistung des Managements durch den konzertierten Einsatz der Instrumente zu erhöhen. Ansätze dieser Art ordnen sich unmittelbar in die integrierende Vorgehensweise der vorliegenden Arbeit ein.

#### 4.1.3.1 Übersetzer → Laufzeitsystem

Innerhalb des Managementsystems der Sprache Orca [BK93] werden Analyseergebnisse des Übersetzers über Abhängigkeiten bezüglich der Nutzung passiver Objekte von dem Laufzeitsystem während der Ausführung leistungssteigernd zur Steuerung der Objekt-Replikation genutzt. In dem ebenfalls sprachbasierten Ansatz Diamonds [NC96] mit der Sprache ODL [SL94] beteiligt sich der Übersetzer an der verteilten Realisierung der Rechenfähigkeit. Er erarbeitet nicht nur Information, sondern entscheidet bereits, welche abstrakt parallel spezifizierten Abläufe konkret parallel realisiert werden. Im Programm festgelegte, abstrakt parallele Methoden-Aufrufe werden nicht uniform auf Threads oder Prozesse abgebildet, sondern flexibel zu größeren Einheiten zusammengefaßt und anschließend mit entsprechenden Laufzeitmechanismen konkret parallel realisiert. Diamonds leidet dabei darunter, daß ODL kein Aktivitätsträgerkonzept anbietet sondern jedes Objekt potentiell aktiv ist. Die Präzision der analysierten Information ist dementsprechend gering.

Noch umfangreicher und systematischer wurde die Integration von Übersetzer und Laufzeitsystem in dem System Concert [CKP93] konzipiert. Zur Bestimmung der Granularität parallel rechnender Einheiten, werden Entscheidungen in einer trichterartigen Hierarchie getroffen. Im ersten Schritt analysiert der Übersetzer wie üblich und trifft Entscheidungen für Probleme, zu deren Lösung bereits ausreichend präzise Information vorliegt. Im zweiten Schritt spekuliert der Übersetzer über die unbekannten Eigenschaften und erzeugt Mengen alternativer Maschinenprogramme. Das Laufzeitsystem besitzt später die erforderliche Information, um eine geeignete Alternative zu wählen. Der dritte angedachte, aber nur schwach ausgearbeitete Schritt der Concert Architektur ist dynamische Übersetzung während der Ausführung.

**Bewertung**
Orca, Diamonds und Concert erleichtern und verkürzen die Entwicklung paralleler und kooperativer Abläufe durch die Automatisierung von Managementaufgaben erheblich und sind deshalb in der Lage, deren Qualität deutlich zu erhöhen. Das Managementsystem kann gleichzeitig flexibler auf Veränderungen der Ausführungsumgebung, z. B. Latenz und Bandbreite des Kommunikationsnetzes und Anzahl der Prozessoren, reagieren. Langfristig ergeben sich daraus auch quantitative Vorteile gegenüber anwendungsintegriertem Management. Diamonds, Orca und einige vergleichbare Ansätze aus dem Bereich DSM [MH94], beschränken sich allerdings auf einen unidirektionalen Informationsfluß von Compiler an das Laufzeitsystem, der überdies nur einmalig stattfindet.

#### 4.1.3.2 Übersetzer ← Laufzeitsystem

Besonders im Bereich des Übersetzerbaus werden in aktuellen Forschungsarbeiten Möglichkeiten untersucht, den Informationsfluß in der umgekehrten Richtung, von der Ausführung an den Übersetzer zu nutzen, um Entscheidungen innerhalb des Übersetzers zu verbessern [Pro98, BKK+98]. Der Prozeß der Zielcode-Optimierung iteriert über vollständige Übersetzungen und Ausführungen eines Programmes, wobei bei wiederholten Übersetzerläufen Information über das Laufzeitverhalten in die Optimierung miteinfließt. Die Erforschung des Potentials dieser Möglichkeit scheint bislang auf spezielle Optimierungsprobleme aus dem Bereich des Übersetzerbaus, wie dem Klonen von Funktionen [WP98], beschränkt zu bleiben. Da die Arbeiten davon motiviert sind, die Optimierungsfähigkeiten des Übersetzers zu verbessern, verwundert es nicht, daß auch in diesem Falle ausschließlich unidirektionaler Informationsfluß geplant ist, wenn auch in der weniger offensichtlichen Richtung von der Laufzeit an den Übersetzer.

#### 4.1.3.3 Integration des Binders

In den Ansätzen der beiden oberen Abschnitte wurde der Binder ausgeklammert. Da der Binder keine ausgezeichnete Bedeutung für die Informationsgewinnung besitzt, ist dies nachvollziehbar. Eine engere Integration der transformatorischen Fähigkeiten des Binders in das Laufzeitmanagement ermöglicht indes ebenfalls signifikante Leistungssteigerung.

**Kommerzielle Systeme**
Verzögertes und dynamisches Binden in SUN Solaris [Sun96] und Linux [WO90] ermöglicht es, Produkte des Übersetzers in ELF-Dateiformat erst spät bei konkreter Nutzung mit bereits vorhandenen Bausteinen zu binden und in Einzelfällen explizit nachträglich zu laden [Sun97]. Genutzt wird die Kombination des Binders mit der Laufzeitumgebung in erster Linie, um Startzeiten zu reduzieren, in dem die Auflösung von Relokationen erst bei Bedarf und nicht schon vor der Programmausführung stattfindet. Um dieses Lazy-Verhalten effizient zu gestalten, wird pro Prozeß eine *Procedure Linkage Table* (PLT) eingesetzt. Sei $w$ ein Benutzerprogramm und $k$ eine Komponente aus einer Bibliothek, die verzögert an $w$ gebunden wird. $w$ referenziere $k$ $n$-malig. Vor dem Start von $w$ werden alle $n$ $k$-Referenzen von dem Binder auf den $k$-Eintrag $plt(k)$ in der PLT von $w$ gesetzt. $plt(k)$ wird mit einem Vorgabewert, der nicht die Adresse von $k$ ist, initialisiert. Wird im Verlauf der $w$-Ausführung auf eine $k$-Referenz zugegriffen, so wird über den Vorgabewert in $plt(k)$ der Binder aktiviert. Dieser hat die Aufgabe, $k$ aufzufinden und $plt(k)$ so zu modifizieren, daß weitere Zugriffe ungehindert und effizient

passieren können. In $plt(k)$ kann dabei nicht die Adresse von $k$ gespeichert werden! Wäre dies der Fall, so müßten $k$-Aufrufe in $w$ vom $w$-Übersetzer bereits entsprechend für das Lesen des Wertes aus einer Tabelle vorbereitet sein und das verzögerte Binden wäre nicht beliebig mit der Übersetzung komponierbar. Tatsächlich generiert der Binder deshalb in $plt(k)$ ein kurzes Maschinenprogramm, bei dem es sich im wesentlichen um eine Sprunganweisung handelt. Aufrufe von $k$ aus $w$ springen somit an $plt(k)$ und von dort weiter an $k$.

Diese Technik ist sehr effizient. Zum einen entstehen trotz des zusätzlichen Sprungs kaum spürbare Leistungsverluste bei Aufrufen und die Relokationsauflösung ist erheblich vereinfacht. Möglich wurden diese Leistungssteigerungen durch die engere Integration des Binders in das Management, die hier nicht auf der Intensivierung des Informationsflusses, sondern auf der Modifikation der temporalen Ordnung von Maßnahmen sowie der flexibleren Zuordnung von Aufgaben – der Binder generiert Zielcode – basiert. Trotz der Integration bleibt die Kompositionalität von Übersetzer- und Binder-Maßnahmen gewährleistet.

**Forschungsarbeiten**

Im Projekt Alto [Deb99] wird ein Binder entwickelt und realisiert, der die Optimierung des Übersetzers fortsetzt. Der Ansatz ist von der Beobachtung motiviert, daß Übersetzer stets nur Programmfragmente verarbeiten, größere Zusammenhänge zu den Komponenten, die später gebunden werden, aber nicht kennen und deshalb auch keine globalen Optimierungen durchführen. Speziell allgemein gehaltene Bibliotheksroutinen können nicht an spezifische Nutzungskontexte angepaßt werden. Alto versucht diese Lücke durch Optimierung zum Zeitpunkt der Bindung zu schließen, indem über das gebundene Gesamtprogramm erneut Datenflußanalysen durchgeführt und anschließend global optimierter Zielcode durch den Binder produziert wird. Im geometrischen Mittel wird durch die Optimierung des Binders eine Beschleunigung um den Faktor 1.25 erzielt. In Anbetracht der Tatsache, daß der Binder keine Kenntnis über den Quelltext besitzt, überraschen die Erfolge. Der Hauptbeweggrund für die gewählte Methode ist offenkundig der Wunsch nach Kompatibilität mit verschiedenen Sprachen und Übersetzern. Alto ist tatsächlich in der Lage, Programme zu optimieren, die sich aus in unterschiedlichen Sprachen verfaßten Teilen zusammensetzen. Bei Verzicht auf diese Heterogenität könnte der Übersetzer in das Optimierungsverfahren miteinbezogen werden und der Erfolg wäre vermutlich noch deutlich größer. Der Optimierung würden präzisere Information über die Anforderungen (Quelltext) zur Verfügung stehen und Konflikte zwischen Entscheidungen von Binder und Übersetzer könnten wirksam vermieden werden.

Die weiter oben in diesem Abschnitt diskutierte Koppelung von Übersetzer und Laufzeitsystem kann durch den Einsatz des Binders weiter intensiviert werden. Der Schritt drei im System Concert (s. 4.1.3.1) sieht den Fluß von Information an den Übersetzer vor, der auf dieser Grundlage während der Programmausführung neuen, nochmals optimierten Zielcode erzeugt. Damit dieser optimierte Code anschließend eingesetzt werden kann, muß der Binder die Fähigkeit besitzen, Zielcodes während ihrer Ausführung auszutauschen. Unklar bleibt, wie dies effizient, transparent für die Problemlösung und dennoch konsistent geschehen kann. Auch in [Kis96] bleiben diese Fragen bei der dynamischen Laufzeitoptimierung mittels iterativer Übersetzung offen.

Zuletzt sei in diesem Kontext das Projekt OMOS [OMHL98] (Object/MetaObject Server) erwähnt, in dem ein kombinierter Progamm-Binder/Lader als Server-Prozeß entwickelt wird, mit dem ausführbare Maschinenprogramme flexibel und dynamisch verwaltet werden können. In [OM98] wird der OMOS-Server zur dynamischen Maximierung der Lokalität von Instruk-

tionsausführungen auf Maschinenebene verwendet. Code-Fragmente werden während ihrer Ausführung anhand kontinuierlich beobachteter Zugriffsprofile im Speicher verschoben. Die Kosten der Informationsgewinnung zur Laufzeit bleiben offen.

#### 4.1.3.4 Diskussion

Der Informationsaustausch bleibt bei den vorgestellten integrierenden Ansätzen meist unidirektional. Die Abstimmung von Entscheidungen zwischen Instrumenten findet nur in seltenen Fällen statt und anstelle eines vollständigen Regelkreises bleiben Informationstransfers und Maßnahmen in aller Regel einmalig. Von den möglichen Instrumenten werden je nach Kontext der Arbeiten stets nur einige wenige naheliegende betrachtet. Kerne scheinen, abgesehen von unmittelbaren Zwängen, z. B. Vereinbarung zwischen Kern und Übersetzer über die Verwendung von Maschinenregistern, von der Integration bisher völlig ausgenommen zu sein.

Obwohl weder konsequent noch systematisch betrieben, zeigen die verschiedenen Integrationsversuche dennoch bereits großes Potential für Leistungssteigerungen. Die Lösung der schwierigen Managementaufgaben in verteilten Systemen: Lastbalancierung, Objekt-Migration/Replikation, etc., können vermutlich nur mit einer solchen Kombination verschiedener Fähigkeiten bewältigt werden. Isolation diente bisher dem Zweck, die wechselseitigen Abhängigkeiten zu reduzieren und somit besser beherrschen zu können. Als Voraussetzung für die notwendige Integration müssen deshalb auch Methoden entwickelt werden, die es erlauben, die vielfältigen Abhängigkeiten zwischen Ressourcen auf unterschiedlichen Abstraktionsebenen mit weniger restriktiven Mitteln zu kontrollieren.

Flexibilisierende und automatisierende Schritte, wie im Falle der Objektplazierung von Orca und der Realisierung der Rechenfähigkeit in Diamonds, sind ohne Zweifel wichtige Meilensteine im Zuge des Wechsels von zentralen und sequentiellen zu V-PK-Systemen. Für das V-PK-Management werden ähnliche integrierende Verfahren erarbeitet, wobei der Prozeß der Informationsgewinnung und -anreicherung gegenüber den hier dargestellten Ansätzen weiter systematisiert und auf Bidirektionalität ausgedehnt wird. Ferner wird die Zuordnung von Managementaufgaben an Instrumente generell neu überdacht. Sie sollte nicht, wie z. B. in Alto, empirisch sondern top-down, orientiert an den Anforderungen, erfolgen.

## 4.2 Erweiterbarkeit

Der Gesamtsystem-Ansatz von MoDiS erfordert die Entwicklung neuer methodischer Grundlagen für die inkrementelle Konstruktion des Systems. Komponenten und Komponentenmengen sollen während der Ausführung des Systems in dieses eingebracht und ggf. auch wieder entfernt werden. Bei den oben diskutierten flexiblen Bindeverfahren handelt es sich um mechanistische Ansätze zur Realisierung bestimmter Formen der Erweiterbarkeit. In diesem Abschnitt werden konzeptionelle Ansätze im Bereich der Erweiterbarkeit untersucht, die selbstverständlich nicht unabhängig von den Mechanismen sind. Faktisch handelt es sich auch bei den oben vorgestellten, adaptionsfähigen Betriebssystemen (Meta-Ebenen-Ansätze) um erweiterbare Systeme. Während in 4.1 jedoch die Integration und Flexibilität des Managements diskutiert wurde, werden im folgenden Ansätze analysiert, die Erweiterungen von in Ausführung befindlichen Programmen zum Gegenstand haben. Betrachtet werden Ansätze, die gemäß dem Dimensionenmodell aus 2.4 in der Dimension Zeit dynamisch inkrementelle Erweiterungen und zudem Variabilität in der Dimension Raum ermöglichen, da dies in etwa den Anforderungen der inkrementellen Konstruktion des MoDiS-Gesamtsystems entspricht.

In der Dimension Abstraktionsniveau wird mit erweiterbaren Kernen zunächst das untere und anschließend mit sprachbasierten Ansätzen das obere Spektrum untersucht. Es sei bemerkt, daß die meisten der im folgenden diskutierten Ansätze nicht im Zusammenhang mit Verteilung stehen. Die Konstruktion eines erweiterbaren Gesamtsystems zur Verbesserung der Informationsgrundlage des verteilten Managements ist ein neuer Ansatz von MoDiS und der vorliegenden Arbeit.

### 4.2.1 Erweiterbare Kerne

Im Bereich der Betriebssysteme ist der Begriff der Erweiterbarkeit, abgesehen von den oben diskutierten Meta-Ebenen-Ansätzen, stark mit der Anpaßbarkeit von Betriebssystemkernen und im Zuge dessen mit der Entwicklung von Mikrokernen verknüpft.[CL95] gibt einen guten Überblick über die Evolution monolithischer Kerne zu erweiterbaren - und Mikrokernen, wobei darauf hingewiesen wird, daß die Assoziation von Erweiterbarkeit mit einer Kernstruktur inadäquat sei. Erweiterungen im Kontext der Betriebssystemkerne dienen dem Zweck der Leistungssteigerung, Hinzunahme neuer Funktionalität oder der Festlegung von Politiken auf Grundlage flexibler Mechanismen. Erweiterungen, wie sie in MoDiS zur Integration neuer Problemlösungen von Anwendern in das Gesamtsystem angestrebt werden, sind davon ausgenommen. Trotz der technisch-mechanistischen Ausrichtung lohnen diese Ansätze eine kurze Analyse, da sie Hinweise für mögliche Realisierungen der MoDiS-Erweiterbarkeit und der Flexibilisierung des V-PK-Managements liefern.

#### 4.2.1.1 Mikro-Kerne der ersten Generation

Ende der 80er Jahre wurden u. a. mit Mach [ABG+86] und Chorus [The92] Mikro-Kerne ($\mu$-Kerne) entwickelt, die als kleine Kerne grundlegende, primitive Fähigkeiten umfassen sollten, um sie als Basis für modulare, portable und flexible Erweiterungen einsetzen zu können. Ihre Fähigkeiten werden dynamisch durch Prozesse auf Benutzerebene auf das Spektrum erweitert, das von den Problemlösungen der Anwender benötigt wird. Mechanistisch findet die Integration der Erweiterung über eine nachrichtenorientierte Schnittstelle statt, über die der Kern mit der Erweiterung oder umgekehrt kommuniziert. Die Bindung der Erweiterung an das ursprüngliche System – den $\mu$-Kern – ist verhältnismäßig lose. Darin begründet liegt der Kritikpunkt der mangelnden Effizienz von $\mu$-Kernen. Die beobachteten Ineffizienzen wurden kontrovers, mit dem Ergebnis der Spaltung der $\mu$-Kern-Entwicklung in zwei grundlegend verschiedene Richtungen, diskutiert.

#### 4.2.1.2 Mikro-Kerne der Folge-Generationen

In der ersten Entwicklungslinie wurden noch kleinere Kerne entwickelt, die ebenso wie die $\mu$-Kerne der ersten Generation nicht im privilegierten Modus erweitert werden. Hierzu gehören L4 [Lie95], der Cache Kernel [CD94] und der Exokernel [EKO95]. Motiviert sind diese Schritte wie im Falle von L4 von der Argumentation, daß nicht der häufige Wechsel zwischen Benutzer- und Kern-Modus für die Ineffizienzen verantwortlich sei, sondern zum einen das Anbieten ungeeigneter Primitiven anstelle adäquater Abstraktionen und zum anderen die bislang ungenügend effiziente Implementierung von $\mu$-Kern-Konzepten (s. [Lie95]). Diese Argumentation wirft einige Fragen auf, die hier wegen dem schwachen Bezug zur Erweiterbarkeit nicht vertieft werden.

Die zweite Richtung verfolgt den entgegengesetzten Weg – Erweiterung des $\mu$-Kern Programms im privilegierten Modus; d. h. dynamische Skalierung von Kernen. Auf diese Weise wird eine wesentlich stärkere und effizientere Bindung zwischen Ursprünglichem (*core*) und Erweiterungen möglich. Dieser Ansatz wird unter anderem von SPIN [BSP+95], VINO [SESS94, SS96, SESS96] und Synthesis/Synthetix [PAB+95] verfolgt. Der Wegfall von Hardware-Kontrollmechanismen durch die Erweiterung des Kerns im privilegierten Modus, macht neue Methoden zur Gewährleistung der Sicherheit erforderlich. SPIN erlaubt Benutzerprozessen dynamisch selbstdefinierte Erweiterungen (*spindles*) in den Kern zu laden um diesen an spezielle Anforderungen anzupassen. Spindles müssen dafür in der streng typisierten Sprache Modula-3 geschrieben und mit einem geprüften Übersetzer übersetzt werden (*trusted computing base*). Ähnlich wie die Benutzerprozesse bei den $\mu$-Kernen der ersten Generation werden auch Spindles nicht wie gewöhnliche Komponenten an den Core gebunden, sondern als *Handler* für *Events* über einen *Dispatcher*-Mechanismus des Core integriert. Das bedeutet, daß es zwischen den Erweiterungen und dem Core inhärent qualitative und quantitative Unterschiede gibt, wenn auch nicht in dem oben beschriebenen Maße. Für eine langfristig ausgelegte Evolution des Systems, so wie sie in MoDiS angestrebt wird, ist auch diese Methode inadäquat. Das Ziel muß die homogene Integration neuer Komponenten in das bereits vorhandene System sein. Andernfalls bleiben Erweiterungsschritte auf dem Niveau von Sonderfällen. Der Binder bzw. der Bindungsprozeß muß in der Lage sein, das gesamte Spektrum von Bindungsintensitäten auch für Erweiterungsschritte durchzusetzen.

#### 4.2.1.3 Synthetix: Automatisierte Synthese angepaßter Kerndienste

Synthetix verfolgt einen Ansatz zur Leistungssteigerung durch inkrementelle Spezialisierung der Kerndienste. Die Besonderheit von Synthetix ist, daß nicht Benutzerprozesse oder ein Systemkonstrukteur von außen Veränderungen am Kern vornehmen, sondern in den Kern ein Übersetzer integriert ist, der von dem Kern selbständig zur inkrementellen Übersetzung des Kerns während seiner Ausführung auf Grundlage bereits bekannter Invarianten sowie Spekulation aktiviert wird. Spezialisierung dieser Art heißt *partielle Evaluation* (PE) [SZ88]. Durch PE wird die wiederholte Interpretation von Pfaden innerhalb des Kerns reduziert und ein signifikanter Leistungsgewinn erzielt. Die Spekulation über mögliche Eingaben erfordert das dynamische Prüfen der Gültigkeit der Spezialisierung, wofür ein *Guard*-Konzept eingesetzt wird. Um die Zeiten für die Produktion von Spezialisierungen zu minimieren, sind stets einige Schablonen und Alternativen vorbereitet. Allerdings ist auch in Synthetix die Erweiterung mittels einer losen Bindung von Komponenten durch separate Datenstrukturen und indirekte Sprünge realisiert. Zusätzlich müssen Aufrufe austauschbarer Dienste von asymmetrischen Sperroperationen begleitet werden [CAK+96]. Synthetix ist damit bezüglich flexiblem, automatisiertem V-PK-Management konzeptionell äußerst interessant. Im Hinblick auf Erweiterbarkeit ergeben sich keine neuen Ergebnisse.

### 4.2.2 Sprachbasierte Ansätze

Bei keinem der oben genannten Ansätze im Bereich der Kerne ist Erweiterbarkeit mit adäquaten sprachlichen Ausdrucksmittel verankert. In Projekten wie SPIN wird die Sprache zwar eingeschränkt, aber Erweiterungen sind dennoch nicht Teil der Sprache. Ein sprachbasierter Ansatz wie MoDiS ermöglicht es, Freiheitsgrade für Erweiterungen konzeptionell bereits auf Ebene der Spezifikation zu verankern. Der potentielle Gewinn daraus ist durchgängige Kennt-

nis möglicher System-Entwicklungen. Daraus ergibt sich die Möglichkeit, ad hoc Lösungen, wie die schwache Bindung eines Großteils aller Komponenten über Indirektionstabellen, zu vermeiden und entsprechende Festlegungen dediziert zu treffen. Zwei Ansätze, die Erweiterbarkeit bereits in der Sprache verankern, sind Oberon und Napier88, wobei die Möglichkeiten im letzteren Fall wesentlich weitreichender sind.

#### 4.2.2.1 Oberon

Das System Oberon [GW89, WG92] wurde als homogenes, sprachbasiertes System inklusive Kern und einer vollständigen Infrastruktur auf Grundlage der gleichnamigen streng typisierten und objektorientierten Sprache [Wir88] konzipiert. Das primäre Ziel dieses Projekts war, ein „gut“ zu beschreibendes und „verständliches“ System von Grund auf neu zu entwickeln und *from scratch* zu implementieren. Im wesentlichen ist es deshalb ein Ein-Benutzer Ein-Prozeß-System, das weder über private Adreßräume noch preemptives Scheduling verfügt. Dem Wunsch nach Vereinfachung wird pragmatisch entsprochen. Fragen, z. B. welche Komponenten einem minimalen System angehören, werden nicht gestellt. Minimalität wird indes oft auf die Größe von Programmtexten bezogen und ggf. durch Verzicht auf Funktionalität erreicht [Fra94a]. Zum aktuellen Zeitpunkt existieren verschiedene Realisierungen [Obe98], die zum Teil in der Forschung und Industrie eingesetzt werden und in akademischen Arbeiten weiterentwickelt werden [WM91, BCFT95].

Die vollständige Neuentwicklung bewirkte eine Reihe pragmatischer Lösungswege, darunter auch einen einfachen Ansatz für die dynamische Erweiterbarkeit des Systems [Mös94], der auf folgenden vier Konzepten der Sprache und des Managementsystems fußt: *Typ-Extension*, dynamisch ladbare *Module*, *Commands* und Ein-Adreßraum. Typ-Extension in Oberon entspricht in etwa einfacher Vererbung in anderen objektorientierten Sprachen. Subklassen werden von Basisklassen abgeleitet, erben dabei die Eigenschaften der Basisklasse und erweitern sie um eigene neue Eigenschaften. Module sind separat übersetzbare Einheiten, die – ähnlich den Depots in INSEL – Objekte und Operationen nach außen exportieren können. Von jedem Modul existiert in einem Oberon-System genau eine Modulinkarnation, die einen dynamischen Zustand besitzt. Modulinkarnationen werden bei Zugriffe auf das Modul erzeugt, dynamisch in den Speicher geladen und gebunden. Zur Nutzung nachträglich entwickelter und geladener Module existiert das *commands*-Konzept. Neben Modulen und Objekten existiert in Oberon kein zusätzliches Konzept für grobgranulare ausführbare Programme, die für gewöhnlich mittels einem Kommandozeileninterpreter zur Ausführung gebracht werden. In Oberon kann jede exportierte Operation eines Moduls aus beliebigen anderen Modulen aufgerufen werden. Weil ferner nur ein gemeinsamer Adreßraum existiert, kann ein Objekt auf alle anderen geladenen Objekte zugreifen. In [WM91] wird gezeigt, wie mit diesen vier Konzepten ein bereits existierender Editor während seiner Ausführung um eine Rechtschreibkorrektur erweitert werden kann. Die Rechtschreibkorrektur befindet sich in einem Modul, das nachträglich implementiert und dynamisch geladen wird. Mit dem *commands*-Konzept können aus dem Text innerhalb des Editors oder anderswo im System Operationen des Rechtschreibkorrektur-Moduls aufgerufen werden, die wegen des gemeinsamen Speichers Zugriff auf den Text besitzen.

Nachträglich hinzugefügte Module werden äquivalent wie vorgefertigte in das System integriert und unterscheiden sich von diesen nach ihrer Integration, im Gegensatz zu den Kernerweiterungen oben, nicht.

**Bewertung**
Die Erweiterbarkeit in Oberon ist ebenso wie das gesamte Projekt von äußerst pragmatischer Natur. Im wesentlichen wird nicht das Potential der Sprachbasierung genutzt, sondern Betriebssystem-Mechanismen angeboten. Die Freiheitsgrade für die Erweiterung von Programmen sind dementsprechend gering. Existierende Module und enthaltene Objekte können nachträglich nicht angepaßt werden, sondern Erweiterungsschritte beschränken sich auf die Aufnahme neuer Module in das System. In der Dimension Raum der Erweiterbarkeit besteht somit kaum Flexibilität, wie es z. B. das Verändern des Anweisungsteiles eines bereits existierenden Objekts erfordern würde und die Granularität von Erweiterungen ist grob. Ferner ist diese Form der unkontrollierten und flach strukturierten Erweiterung nur unter den vereinfachenden Rahmenbedingungen von Oberon tragfähig. In einem verteilten Mehr-Benutzer Mehr-Prozeß-System müssen Erweiterungen flexibler und kontrollierter durchführbar sein. Interessant ist dennoch die hohe Kompositionalität, Wiederverwendbarkeit und Entwicklungsfähigkeit des Systems und seiner Komponenten, die mit diesen einfachen Mitteln tatsächlich bereits erzielt werden.

#### 4.2.2.2 Napier88

Ein gänzlich anderer Weg wurde bei der Entwicklung der Sprache Napier88 [BDM+90, MBC+96] gewählt. Die Motivation für Napier88 ist, eine Sprache anzubieten, die geeignet ist, um Programme in einer persistenten Umgebung flexibel inkrementell zu konstruieren. Im Umfeld der Sprache Napier88 entstand konsequent auch das Betriebssystem Grasshopper [Dea94, RDH+96], dessen Konzepte auf die Gewährleistung von Persistenz als Ersatz für herkömmliche Dateisysteme ausgerichtet sind. Die automatisierte Durchsetzung von Persistenz hat eine Vielzahl von Vorteilen. U. a. entfällt die fehlerträchtige Notwendigkeit, aufwendige Transformationen zwischen den strukturierten Daten in Anwendungen und der flachen Organisation in Dateisystemen vorzunehmen.

Programme, die Typen, Prozeduren oder Datenobjekte beschreiben, sind in dieser Umgebung ebenfalls Daten und werden wie diese behandelt. Persistente Daten und somit auch Programme, existieren so lange, wie sie benötigt werden und sind in der Lage sich von innen oder außen über die Zeit weiter zu entwickeln. Die für Napier88 entworfenen Konzepte für die Evolution von Programmen, sind vielfältig und reichen von Typ-Algebren über Laufzeit- und Compile-time-Reflektion bis hin zu Hyper-Programming [Dea89, KCC+92]. Mechanistisch wurden ebenso umfangreiche Arbeiten durchgeführt. So wurde ein dynamisch aktivierbarer Übersetzer, inkrementeller Lader und Hyper-Programm-Browser realisiert. Programme entstehen mit diesen Konzepten und Werkzeugen nicht durch lineare Aufschreibung ihres beschreibenden Textes, sondern werden flexibel durch Komposition von Daten (Programme oder Datenobjekte) aus dem persistenten Speicher und neu hinzugefügten Fragmenten erstellt. Daten innerhalb des persistenten Speichers werden via *bindings* benannt. Ein binding assoziiert einen Namen mit einem Wert, einem Typ und einer Konstante. Im Falle von Programmen, bezeichnet der Wert das gewünschte Programm. Mit dem Sprachkonzept *environments* werden bindings innerhalb von Programmen flexibel und dynamisch initiiert, aufgelöst und kontrolliert.

Um diese Konzepte effizient zu realisieren, wurde im Kontext von Napier88 versucht, den Übersetzer eng in das Management erweiterbarer Programme miteinzubeziehen. Anstelle einer eigenen Codeerzeugung, die volle Kontrolle über die Registerzuteilung, etc. ermöglicht hätte, wurde aus Aufwandsgründen die Eingabesprache des GNU C-Übersetzers gcc als Ziel-

sprache für den Napier88-Übersetzer eingesetzt [Dea87]. Die GNU-Erweiterungen der Sprache C wurden genutzt, um Entscheidungen nahe an der Hardware, z. B. über die Verwendung von Registern durch den Napier88-Übersetzer, zu steuern. Weiter wurden neue, flexible Bindertechniken [Dea89] zur Unterstützung der inkrementellen System-Evolution entwickelt. Letztendlich unterstützt allerdings auch der Binder von Napier88 nur eine Bindungsvariante für alle Komponenten des Systems: lose Koppelung mittels Tabellen, deren Einträge bei Nutzung von Komponenten interpretiert wird.

**Bewertung**
Mit Napier88 wurden sowohl konzeptionell als auch bezüglich der Instrumentierung zahlreiche neue Ansätze kreativ entwickelt, die wertvolle Impulse zur Lösung der Erweiterungsproblematik in MoDiS liefern. Konträr zu den MoDiS-Konzepten geht Napier88 allerdings von einer flachen Struktur mit globaler Sichtbarkeit aus, was sowohl aus rechtlichen Gründen als auch bezüglich physischer Verteilung aus der Sicht von MoDiS fragwürdig ist. Ebenso verhält es sich mit der Festlegung der Persistenz. Napier88 geht von orthogonaler Persistenz aus: Die Erzeugung und Manipulation von Daten ist unabhängig von deren Lebenszeit. INSEL-Komponenten besitzen eine konzeptionell definierte Lebenszeit, die sehr wichtig ist, um Strukturierung durchsetzen und effizientes Management auf Grundlage wohldefinierter Abhängigkeitsintervalle durchführen zu können. Napier88 wurde im gewissen Sinne nach wie vor bottom-up orientiert entwickelt; zwar nicht an der Hardware orientiert, aber dennoch ausgehend von vorhandenen Übersetzer- und Bindertechniken. In der vorliegenden Arbeit wird Erweiterbarkeit noch stärker top-down orientiert anhand der INSEL-Konzepte ausgearbeitet, um zu einem homogenen Konzeptevorrat zu gelangen, der den Anforderungen auf Ebene der Sprache genügt. In Bezug auf die Realisierung wird in der vorliegenden Arbeit ein inkrementeller Binder für INSEL entwickelt, der sich nicht auf eine lose Bindung von Komponenten beschränkt, sondern ein Repertoire von Bindungsintensitäten anbietet, aus dem angepaßt an die gewünschte Flexibilität, ein geeignetes Verfahren gewählt werden kann.

## 4.3 Prototypische Realisierungen der MoDiS-Architektur

In Arbeiten, die der vorliegenden voran gingen, wurden bereits auf unterschiedlichen Lösungswegen INSEL-Experimentalsysteme realisiert und damit prototypische Implementierung der MoDiS-Architektur vorgenommen. Diese Arbeiten liefern wertvolle Erkenntnisse für eine adäquate Konkretisierung der abstrakten INSEL-Konzepte und der MoDiS-Management-Grobarchitektur. In diesem Abschnitt werden die Eigenschaften der unterschiedlichen Lösungsansätze evaluiert und daraus Anforderungen an das neu zu konstruierende V-PK-Management abgeleitet.

Die Entwicklung der Prototypen in der Vergangenheit fand unter anderen Voraussetzungen als den heutigen statt. Unter anderem hat erst in den letzten drei bis fünf Jahren ein rapider Wandel in der Betrachtung von Systemprogrammen eingesetzt. Vor allem im Zuge der Entwicklung von Systemen wie Linux [Han99] und FreeBSD entstand ein enormes Repertoire qualitativ hochwertiger Systemprogramme, die im Quellcode frei verfügbar sind, beliebig modifiziert werden können und somit eine ideale Ausgangsbasis für Experimente im Bereich der Forschung darstellen. Die Randbedingungen der prototypischen Realisierungsarbeiten waren demgegenüber ca. 1990 noch deutlich schwieriger, geprägt von proprietären Systemen und restriktiver Informationspolitik. Ferner spiegelt sich in den Unterschieden zwischen den Pro-

totypen auch die Reihenfolge ihrer Entwicklung wieder. Selbstverständlich profitierten spätere Arbeiten von den Ergebnissen vorangegangener Versuche. Bei der Interpretation der folgenden Diskussion der INSEL-Experimentalsysteme sollte der Leser diese historisch bedingten Eigenschaften berücksichtigen. Im Text werden die Eigenschaften der Experimentalsysteme anschließend ausschließlich mit den Zielen des V-PK-Managements aus heutiger Sicht verglichen.

Teilaspekte der MoDiS-Architektur wurden bislang mit den vier Prototypen DAViT[2] [Wei97], EVA[3] [Rad95], AdaM[4] [Win96b, Win96a] und Shadow [GP97, Gro98] realisiert. Die Reihenfolge der Nennung entspricht der zeitlichen Ordnung in der die Arbeiten durchgeführt wurden.

DAViT nimmt unter diesen Prototypen eine Sonderstellung ein. Als Werkzeuge zur interaktiven Interpretation und Visualisierung von INSEL-Systemen inklusive innerer Komponenten und der INSEL-Strukturrelationen auf Basis von X11 und HP-UX [Hew92], dient DAViT in erster Linie dem Erlernen und Erproben der INSEL Konstruktionskonzepte. Aspekte der Erweiterbarkeit sowie effizientes Management fehlen naturgemäß, wenngleich DAViT in der Lage ist, INSEL-Programme parallel und verteilt auf einer NOW-Konfiguration zu verarbeiten. Für die vorliegende Arbeit ist DAViT dennoch aus zwei Gründen interessant. Erstens ist DAViT die bislang einzige Implementierung des gesamten INSEL Sprachumfangs, aus der sich Hinweise für die Realisierung des V-PK-Managements ableiten lassen. Zweitens, handelt es sich gemäß 2.3.3.2 bei dem Instrument DAViT um einen Interpretierer in „Reinform“. Damit ist gemeint, daß DAViT tatsächlich nahezu jeden Berechnungsschritt wiederholt neu interpretiert und keine Zwischenergebnisse invarianter Berechnungsschritte, z. B. in Schleifen, während der Ausführung von Programmen produziert und wiederverwendet. Die quantitative Leistung ist dementsprechend gering und liegt um einige Zehnerpotenzen hinter optimiert übersetzten Programmen zurück. Auf der anderen Seite wird mit der Instrumentierung des V-PK-Managements in der vorliegenden Arbeit versucht, Interpretation durch leistungsfähige Transformation zu realisieren. DAViT ist in diesem Kontext ein Paradebeispiel, an dem das Wesen wiederholter Interpretation studiert werden kann.

Alle anderen MoDiS-Prototypen stützen sich auf zusätzliche transformatorische Maßnahmen ab und orientieren sich an der klassischen Aufteilung in Übersetzer, Binder, Lader, Laufzeitsystem und Kern. In den folgenden Abschnitten werden zunächst ihre gemeinsamen Eigenschaften erörtert, bevor auf Spezifika eingegangen und ein abschließendes Fazit gezogen wird.

### 4.3.1 Gemeinsame Eigenschaften

Obwohl im Rahmen von MoDiS z. B. in [EM97] auch qualitative Aspekte verteilten Rechnens (s. 2.2) untersucht werden, konzentrierten sich alle drei transformations-basierten INSEL-Experimentalsysteme auf die Nutzung verteilter Ressourcen zur Steigerung der quantitativen Leistung und dabei insbesondere der Verkürzung von Ausführungszeiten. Die Tatsache, daß akzeptable Performanz zum einen eine wichtige Voraussetzung ist, um verteilte Systeme aus qualitativen Beweggründen einsetzen zu können und zum anderen diese Forderung mit automatisierten systemintegrierten Management nach wie vor nicht erreicht wird, rechtfertigt diese Schwerpunktbildung.

[2] Dynamic Analyzation and Visualization Tool
[3] Experimentalsystem für Verteilte Anwendungen
[4] Adaptives Ressourcenmanagement unter Mach 3.0

Das Thema der Erweiterbarkeit wurde in keiner der drei Arbeiten ausgearbeitet; weder konzeptionell noch durch entsprechende flexibilisierende Maßnahmen des Managements, wie zum Beispiel durch einen inkrementellen Binder. Infolgedessen erlauben die Experimentalsysteme nicht die Konstruktion und Verwaltung eines INSEL-Gesamtsystems, sondern lediglich die physisch verteilte Ausführung verhältnismäßig kurzlebiger, kleiner INSEL-Programme zu Versuchszwecken. Nach der Beendigung seiner Berechnung wird das Exempel-INSEL-System vollständig aufgelöst. Eine längerfristige Entwicklung eines INSEL-Systems, die über Start, Ausführung und Terminierung genau eines INSEL-Programmes hinaus geht, findet nicht statt.

Das gesamtheitliche Vorgehen bei der Systemkonstruktion war daher insgesamt noch schwach ausgeprägt. Die Ressourcenverwaltung der Prototypen wurde selbst nicht in INSEL sondern der Sprache C++ oder C implementiert. Wechselwirkungen von den Sprachkonzepten mit der Idee des Gesamtsystems, wie z. B. eine INSEL-Sprachhierarchie von abstrakt nach konkret, wurden dementsprechend nicht untersucht. Weil die Ressourcenverwaltungen selbst nicht mit den INSEL-Konzepten konstruiert wurden, wurde die Ressourcenverwaltung auch selbst nicht an das INSEL-System integriert. Die Verwaltungssysteme der Prototypen profitieren somit nicht von den Vorteilen der INSEL-Konzepte, Probleme der Strukturierung des Gesamtsystems in System- und Anwendungskomplexe blieben offen und die Erfahrungen mit INSEL beschränken sich auf die Implementierung kurzer Beispielalgorithmen anstelle der Erkenntnisse, die mit einer Implementierung des INSEL-Managements in INSEL möglich gewesen wären. Insgesamt zieht sich somit ein Bruch zwischen INSEL und INSEL-Management, der die gewünschte Homogenität behindert und sich von der Sprache und der Anbindung von Problemlösungen an das Management über die gesamte Architektur der Experimentalsysteme erstreckt.

Alle Prototypen sind ferner von einer drei-geteilten Betrachtung von INSEL-Systemen geprägt, die zwischen dem INSEL-Programm, den Managern (keine INSEL-Komponenten) und der zugrundeliegenden Plattform (Hardware und übernommene Instrumente) unterscheidet. Diese Auffassung war noch stark mit einem additiven Ansatz bei der Realisierung der Experimentalsysteme verhaftet, der wechselseitig dadurch wiederum weiter gefördert wurde. Das Ausklammern der Erweiterbarkeit führte überdies dazu, daß das Potential der Integration des Übersetzers und vor allem auch des Binders in das Management wie auch in anderen Ansätzen im Bereich der Betriebssysteme nicht erkannt wurde. Erst die Zielsetzung Erweiterbarkeit zu ermöglichen hätte dazu gezwungen, transformatorische Mittel zu analysieren und ihre Rolle grundlegend zu überdenken. Die Realisierung der Manager beschränkte sich deshalb in allen drei Ansätzen im wesentlichen auf nicht weiter differenzierte Objekte von Laufzeitbibliotheken, die nach der Übersetzung eines INSEL-Programmes an dieses statisch gebunden werden[5]. Eine deutlich darüber hinausgehende Integration der Fähigkeiten verschiedener Instrumente wurde nicht geplant. Die Kreativität und Flexibilität war damit a priori in vielen Richtungen eingeschränkt.

### 4.3.2 Experimentalsystem 2: EVA

Für die Realisierung von EVA wurde wie schon zuvor bei DAViT das UNIX-Derivat HP-UX gewählt, mit dem zu diesem Zeitpunkt > 100 HP PA-RISC Arbeitsplatzrechner für Experimente zur Verfügung standen. Mangels Alternativen mußte in Kauf genommen werden, daß HP-UX nicht im Quellcode zur Verfügung stand und die Management-Instrumentierung

[5] Ausnahme Shadow: Forderungen an die Speicherverwaltung des Kerns und die Maßnahmen des Übersetzers

damit stark eingeschränkt war. Bezüglich der Ressourcenverwaltung konzentrierten sich die Arbeiten an EVA auf die Plazierung von - und Kooperation zwischen INSEL-Akteuren im verteilten System. Da zwischen den Angeboten von HP-UX und den Anforderungen leichtgewichtiger, beliebig granularer Akteure eine weite Lücke klaffte, wurde zunächst mit großen Anstrengungen der Distributed Thread Kernel – DTK – entwickelt und darauf aufbauend die abstrakten Manager realisiert.

#### 4.3.2.1 Der verteilte Thread-Kern DTK

Das Angebot von HP-UX Version 9 zur Realisierung von Rechenfähigkeit war auf Prozesse mit eigenen privaten Adreßräumen beschränkt, deren Kosten für Erzeugung, Auflösung sowie Kontextwechsel nur im Falle langlebiger, sprich grobgranularer, Berechnungen lohnen. Prozeßadreßraumgrenzen behindern weiter den flexiblen und effizienten Austausch von Daten und erlauben somit keine effiziente Realisierung der Kooperation von Akteuren mittels gemeinsamer DE-Komponenten oder Depots. Zusätzlich ist die Zahl gleichzeitig existierender Prozesse stark, auf möglicherweise maximal 64, beschränkt. Weil die Eigenschaften der bottom-up Abstraktion Prozeß als Realisierungsinstrument für Akteure somit völlig unzulänglich waren, wurde DTK auf HP-UX entwickelt. DTK ist eine C++ Bibliothek mit Basis-Klassen für leichtgewichtige DTK-*Threads*, die dynamisch in nahezu beliebiger Zahl ($> 1000$) explizit erzeugt und aufgelöst werden können. Der Benutzer kann beim Start des Systems zwischen den Strategien *Random, Optimal Local, Load* und $\{R|S\}$*Bidding* [RP97b] wählen, die von DTK zur initialen Plazierung der Threads auf den Stellen angeboten werden. Thread-Migration zwischen Stellen während der Ausführung wird nicht unterstützt. Erzeugte DTK-Threads besitzen global eindeutige DTK-Identifikatoren und können synchron nachrichtenorientiert kommunizieren. Alle auf einem Rechner erzeugten Threads werden in dem gemeinsamen Adreßraum des UNIX-DTK-Prozesses (DTK-Task) auf dieser Stelle ausgeführt. Über Stellengrenzen hinweg bietet DTK keinen gemeinsamen Speicher. Kooperation via passive Komponenten muß deshalb mit der synchronen DTK-Kommunikation realisiert werden. Die Angebote von DTK sind somit leichtgewichtige Threads, systemintegrierte Lastverteilung, globaler Namensraum der DTK-Identifikatoren und verteilungsopake synchrone Kommunikation.

#### 4.3.2.2 Konkretisierung der konzeptionellen Manager

Die Gesamtarchitektur von EVA ist in Abbildung 4.1 dargestellt. Das auszuführende INSEL-Programm wird in die Hochsprache C++ [Sch95] und anschließend mit einem unmodifiziert übernommenen C++ Übersetzer in Maschinensprache transformiert. Das Produkt der zweistufigen Übersetzung wird statisch mit der DTK-Bibliothek zu einem DTK-Programm gebunden. Das DTK-Programm wird auf jeder Stelle der Konfiguration kopiert und als DTK-Task mit separatem Adreßraum pro Stelle zur Ausführung gebracht.

Die Akteursphären-Manager, die im Abstrakten unterschiedliche Aufgaben besitzen, sind uniform als Instanzen einer von DTK-Thread abgeleiteten Klasse realisiert. Da jeder Akteur selbst auch durch einen Thread realisiert wird, ist jedes konzeptionelle Akteur/Manager-Paar durch zwei DTK-Threads innerhalb eines DTK-Tasks implementiert. Akteur- und Manager-Thread kooperieren via synchroner DTK-Kommunikation. Order mit der Ausnahme von K-Order sind auf C++ Unterprogramme abgebildet. K-Order werden mittels der dafür entwickelten synchronen DTK-Kommunikation realisiert. Der Akteur-Thread erteilt seinem Manager-Thread hierzu den Auftrag, der diesen an den Kooperationspartner in der gleichen

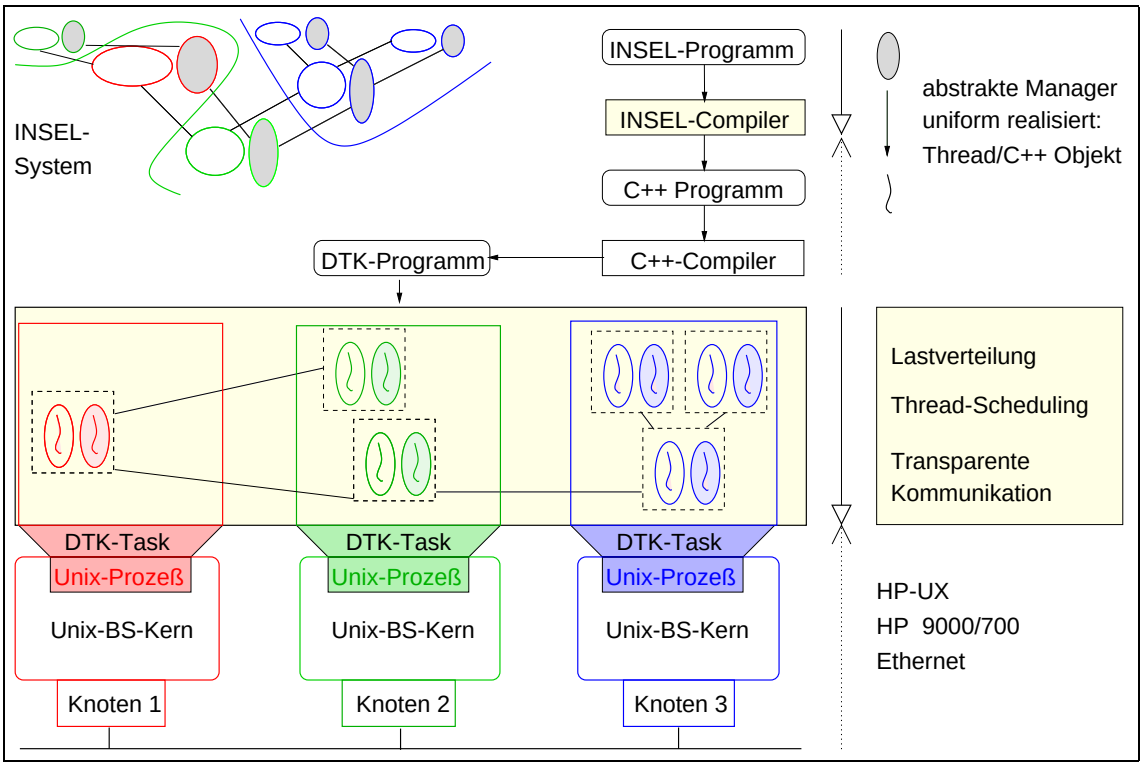


Abbildung 4.1: Struktur der INSEL-Realisierung auf Basis von HP-UX/DTK

Systematik weiterleitet.

Problematisch ist die Realisierung von wert-orientierten DE-Komponenten und Depots. Ausführungsumgebungen können durch $\delta$-Schachtelung und der verteilten Thread-Plazierung physisch verteilt realisiert sein. Um Zugriffe auf Datenobjekte[6] entfernter Akteursphären (AS) zu ermöglichen, wurden in EVA *Frames* und Pfadbeschreibungen eingeführt. Sämtliche Datenobjekte einer AS werden ungeachtet, ob es sich um N- oder Z-Komponenten handelt, nicht in einem Laufzeitkeller sondern einer separaten Frame-Liste realisiert, auf die sowohl der zugeordnete Akteur-Thread als auch sein Manager-Thread direkten Zugriff besitzen. Bei Zugriffen auf Daten einer fremden Akteursphäre beauftragt der Akteur-Thread seinen Manager-Thread mit der Durchführung der Operation. Der Manager-Thread interpretiert die Pfadbeschreibung und leitet den Auftrag an den Manager-Thread des Akteurs weiter, in dessen AS sich die gewünschte Komponente befindet. Der kontaktierte Manager-Thread führt letztendlich die Operation auf dem Datenobjekt aus. Durch die räumliche Verteilung von Akteuren kann die Interpretation eines einzigen Zugriffspfades das mehrfache Versenden von Nachrichten über das Netz erfordern. Dieses aufwendige Zugriffsverfahren wird selbst dann durchlaufen, wenn die AS des gewünschten Datenobjekts auf der selben Stelle wie der anfragende Akteur realisiert ist. Eigentlich könnte in diesem Fall der anfragende Akteur-Thread den gemeinsamen Speicher innerhalb des DTK-Tasks nutzen, um auf die gewünschte Komponente direkt zuzugreifen. Diese Flexibilität besitzt das Verfahren der Pfad-Interpretation nicht. Aufgrund der separaten Adreßräume, besitzen die Identifikatoren von Z-Komponenten ferner immer nur innerhalb einer DTK-Task ihre Gültigkeit, was weitere repetitive Transformationen erfordert.

[6] hier: wertorientierte DE-Komponenten und Depots

#### 4.3.2.3 Bewertung

Umfangreiche Experimente mit EVA demonstrierten die Effizienz der verteilten Erzeugung, Auflösung und Kommunikation von DTK-Threads. Zudem erwies sich das Zurückstellen von Erzeugungsaufträgen im Falle der Bidding-Lastverteilungsstrategien als sehr zweckmäßig. Allerdings zeigten sich auch gravierende Schwächen des Frame-Konzepts. Die überaus häufigen Zugriffe auf wert-orientierte Komponenten sind aufgrund der Frames auch im lokalen Fall innerhalb einer AS um Größenordnungen langsamer als in anderen Systemen. Zur Behebung dieses Defizites müssen Datenobjekte zwischen Stellen migriert und repliziert werden, um Nachrichtentransporte zu minimieren und das wiederholte aufwendige und vielschichtige Interpretieren der Pfadbeschreibung so weit als möglich zu minimieren. Voraussetzung dafür ist ein gemeinsamer Speicher, dessen Identifikatoren weitestgehend in Hardware anstelle von Software interpretiert werden können und die Realisierung des Managements auf einem niedrigeren Abstraktionsniveau als C++, um direkt auf Laufzeitkeller et cetera Einfluß nehmen zu können.

Betrachtet man die absoluten Beschleunigungswerte von EVA, so zeigt sich, daß die Leistungsfähigkeit dieses ersten Versuches noch nicht befriedigen kann. Selbst bei ausschließlich lokaler Ausführung von INSEL-Programmen und der vollständigen Eliminierung AS-übergreifender Zugriffe liegen die Ausführungszeiten immer noch um den Faktor sieben hinter der Ausführung eines vergleichbaren Programmes in einer anderen Hochsprache zurück. Bei einem konstanten lokalen Mehraufwand in dieser Größenordnung besteht kaum Aussicht, de facto Effizienzgewinne mit der verteilten Ausführung von INSEL-Programmen zu erzielen (s. 2.2.2).

In der vorliegenden Arbeit gilt es, die Ursachen dieser konstanten Ineffizienz zu beseitigen. Die Einflußfaktoren sind allerdings vielfältig und beginnen mit der Zwischensprache C++. Die Angebote von C++ entsprechen den Anforderungen von INSEL nur bedingt. Zum einen existieren einige Fähigkeiten, wie Objektreferenzen, die nicht benötigt werden. Zum anderen finden einige elementare Anforderungen, die sich u. U. effizient auf die Hardware abbilden ließen, kein Pendant in C++ und müssen mit aufwendigen Hilfskonstruktionen umschrieben werden. Die wichtige Zielcode-Optimierung des C++ Übersetzers kann diesen Überhang nachträglich nicht beseitigen. Ein Beispiel ist die Parameterübergabe. Da C++ u. a. kein Gegenstück für *Copy-In-Out* besitzt, werden Order-Parameter auf der Halde anstelle von Registern oder Keller übergeben. Das Problem der inadäquaten Angebote setzt sich mit ähnlichen Folgen an der Schnittstelle DTK und HP-UX fort.

Zu dem Übersetzerproblem gesellt sich mangelnde Flexibilität uniformer Lösungen, z. B. Pfad-Interpretation und Erzeugen eines Threads pro abstraktem Manager. Uniformität bedeutet Allgemeingültigkeit verbunden mit aufwendiger Interpretation. Im Falle von EVA ignorieren die uniformen Verfahren selbst die ungleich höheren Kosten für verteilte Kommunikation, woraus in vielen Fällen gravierende Ineffizienzen resultieren. Das neu zu konstruierende V-PK-Management muß deshalb wesentlich differenzierter und flexibler mit großem Augenmerk auf Details konstruiert werden. So müssen Entscheidungen, wie die Abbildung des Manager-Konzepts auf C++ Objekte zuzüglich Threads, oder die Realisierung der Akteur-Manager-Anbindung mittels synchroner Nachrichten, stärker differenziert werden, damit die gesteckten Ziele erreicht werden können. Differenzieren bedeutet dabei das Einziehen zusätzlicher Abstraktionsebenen, um den schrittweisen Übergang von abstrakt nach konkret systematisch vollziehen zu können.

### 4.3.3 Experimentalsystem 3: AdaM

Der Schwerpunkt der Untersuchungen von AdaM lag auf der Entwicklung von Konzepten und Verfahren zur effizienten Realisierung der Speicherfähigkeit passiver INSEL-Komponenten und dabei im wesentlichen Depots. Mit AdaM wurden erste Erfahrungen mit einem Mikro-Kern-Ansatz gewonnen. Die Wahl des Mach 3.0 Kerns, der zu dem damaligen Zeitpunkt auf der Intel x86 Architektur zur Verfügung stand, ermöglichte es, mit der Speicherverwaltung im Benutzermodus zu experimentieren. Vollständig integriert in den Mach-Kern waren bereits leichtgewichtige Prozesse (C-Threads) mit gemeinsamen Adreßräumen sowie ein Port-Konzept für die Kommunikation zwischen Tasks. Im Gegensatz zu HP-UX war Mach inklusive Kern, Binder und Übersetzer im Quellcode verfügbar. Von den diesbezüglichen Möglichkeiten wurde allerdings kein Gebrauch gemacht.

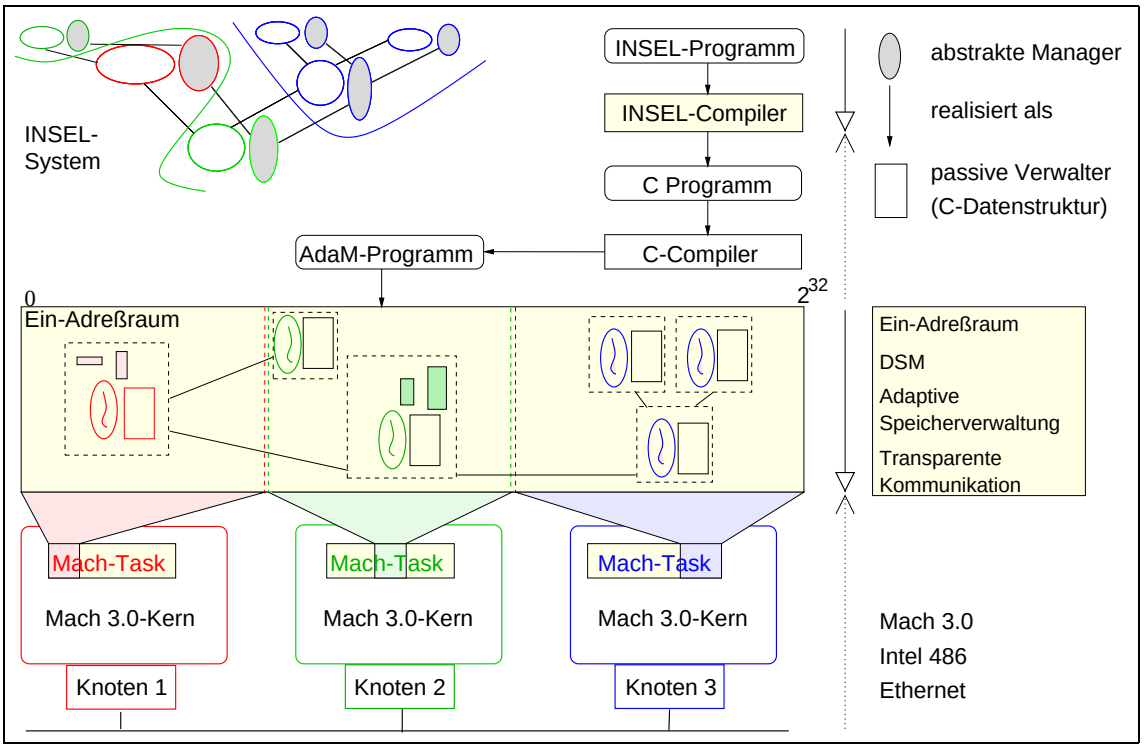


Abbildung 4.2: Mach-basierter Prototyp mit Ein-Adreßraum

In Abbildung 4.2 ist die Architektur von AdaM dargestellt. Basierend auf den negativen Erfahrungen mit C++ als Zwischensprache wurde C als Zwischensprache gewählt, wobei der produzierte C-Code sehr nahe an einem Maschinenprogramm ist. Wie bei EVA wird das übersetzte C-Programm statisch mit einer Laufzeitbibliothek zu einem AdaM-Programm gebunden, das anschließend auf den Mach Kernen aller Stellen zur Ausführung gebracht wird. Die transformatorischen Maßnahmen unterscheiden sich mit der Ausnahme der Zwischensprache kaum von EVA. Das Laufzeitmanagement der beiden Systeme unterscheidet sich allerdings gravierend.

Die abstrakten Manager sind in AdaM als passive C-Datenstrukturen realisiert, deren Erzeugung und Auflösung kaum Aufwand verursacht. Akteure werden durch C-Threads realisiert und dienen als Einheiten der Verteilung, die sich auf eine primitive initiale Zuteilung

beschränkt. K-Order Aufträge erteilt ein aufrufender Akteur mittels dem Mach Port-Konzept direkt dem K-Akteur-Thread des Auftragnehmers. Alle anderen Order mit der Ausnahme von Depot-Zugriffsoperationen werden lokal prinzipiell als C-Unterprogramme ausgeführt. Benannte Datenobjekte werden auf Einträge in Felder (N-Array) der Sprache C abgebildet. N-Arrays sind jeweils lokale Variablen der für DA-Komponenten generierten C-Funktionen und realisieren einen Keller-Rahmen mit explizitem Kellerzeiger et cetera. Diese Zusammenfassung benannter Datenobjekte wird genutzt, um in C Zeiger auf N-Arrays weiterreichen zu können, mit deren Hilfe effiziente Zugriffe auf Datenobjekte entlang der $\delta$-Struktur möglich sind. In den N-Arrays sind zusätzlich Displays auf $\delta$-äußere DA-Komponenten realisiert, wodurch die aufwendige Pfad-Interpretation von EVA auf wenige primitive Schritte reduziert wird. Anonyme Datenobjekte werden in der Halde auf der Stelle ihrer Erzeugung realisiert.

In AdaM wurde ein SVM-System mit sequentieller Konsistenz eingesetzt, der einen 32 Bit weiten gemeinsamen Speicher über die physisch verteilte Konfiguration etabliert. Der globale Adreßraum ist statisch anhand der Stellen partitioniert. Die Kooperation zwischen Akteuren und ihren beigestellten passiven Manager mit anderen Akteuren und Manager ist dadurch wesentlich flexibler als bei EVA. Zum einen können Nachrichten über Ports versendet werden, zum anderen kann direkt auf Speicher zugegriffen werden. Auf lokale Datenobjekte kann durch den gemeinsamen Speicher mit den Displays fast genauso effizient zugegriffen werden, wie es üblicherweise der Fall ist. Entfernte Datenobjekte werden bei Bedarf von dem DSM-System repliziert. Folgezugriffe können sehr effizient ohne zusätzliche Nachrichtentransporte passieren und das mehrfache Versenden von Nachrichten in Folge der Interpretation eines aufwendigen Zugriffspfades entfällt.

Bei AdaM wurde ferner beobachtet, daß Seiten als Transporteinheiten aufgrund der Gefahr von *false sharing* oft nicht geeignet sind und weder Replikation noch Migration als alleiniges Mittel in allen Fällen zu befriedigender Leistung führten. In [Win95, Win96b] wurde deshalb der Transport und die verteilte Ausführung von Operationen auf Datenobjekte flexibilisiert. Anhand statischer Analysen von Nutzungsabhängigkeiten durch den Übersetzer werden Datenobjekte in die Klassen *replizierbar*, *migrierbar* und *dynamisch replizierbar* eingeteilt. Zur Laufzeit wird die Analyseinformation durch Monitoring des Nutzungsverhaltens angereichert und ggf. dynamisch über die Realisierung mittels Migration, Replikation oder Realisierung der Zugriffsoperation mittels RPC entschieden. Die Integration des Übersetzers in das Management wurde im Bereich der Speicherverwaltung weiter genutzt, um einige Realisierungsentscheidungen anhand von Optimierungseigenschaften, wie statisch begrenzbarer Speicherbedarf, besser an die Anforderungen anzupassen.

Für die Zuteilung und Freigabe des verteilten Speichers an die Akteure wurde eine zweistufige Verwalterarchitektur, bestehend aus Stellen-Speicherverwalter (WV) und AS-Speicherverwalter (SV), entwickelt. Ein SV realisiert den Aufgabenbereich Speicherfähigkeit eines abstrakten AS-Managers. Bei der Erzeugung von Keller- oder Haldenobjekten, fordern Akteure Speicher bei ihrem SV an, der als MFLFII-Allokator [Iye93] auf diese feingranularen Allokationen ausgelegt ist. Verfügt der SV selbst nicht über genug Speicher, so wendet er sich an den WV auf dieser Stelle, der die Partition des virtuellen Adreßraumes, die diesem Knoten zugeordnet ist, in einem Pool grobgranularer Speicherbereiche verwaltet.

#### 4.3.3.1 Bewertung

Mit den Arbeiten am Experimentalsystem AdaM wurden einige wichtige Fortschritte erzielt. In aller erster Linie sind die Maßnahmen der Flexibilisierung und der Ansatz zur Nutzung von

Übersetzeranalysen zu nennen. Umfangreiche Meßreihen belegen die Gewinne dieser Maßnahmen. Weitere Leistungen der AdaM-Arbeiten sind die Realisierung und Demonstration der Vorteile des Ein-Adreßraumes und die leichtgewichtige Realisierung von Manager mit passiven Datenstrukturen. Die Wahl des Mach-Kerns sowie der Erzeugung von C-Code auf niedrigem Abstraktionsniveau waren zudem erste wichtige Schritte zur Loslösung von der bis dahin inflexiblen, additiven zu einer adaptiven Vorgehensweise bei der Konstruktion von Systemen.

Wie schon bei EVA wurden auch mit AdaM vielversprechende, relative Beschleunigungen gemessen. Die absolute Beschleunigung bleibt allerdings auch bei AdaM hinter den Erwartungen zurück, wofür abermals die zweistufige Übersetzung verantwortlich zu machen ist. Der generierte, hardwarenahe C-Code kann zwar effizient auf Fähigkeiten der Prozessoren abgebildet werden, verhindert aber gleichzeitig erfolgreiche Optimierung durch den Übersetzer. Ein Beispiel sind die N-Arrays. Sie werden von dem C-Übersetzer unmittelbar auf eine C-Keller-Variable abgebildet. Jede INSEL-Variable wird damit tatsächlich im Speicher realisiert und die Register der Hardware bleiben ungenutzt bzw. werden zur Realisierung des zusätzlichen Kellerzeigers und Operationen auf diesem eingesetzt. Ähnliches gilt für die Realisierung der Abschlußsynchronisation von Order, die zu diesem Zweck immer indirekt über eine Mittlerfunktion gestartet werden, um die Order als Teil der Akteursphäre zu registrieren. Bei der Auflösung der Order wird dies genutzt, um in $\epsilon$ eingeordnete, lebenszeitabhängige Z-Komponenten automatisch finden und auflösen zu können. Diese Detailprobleme führen bereits zu einem lokalen Mehraufwand im Bereich von $l = 4$, der das Erreichen der quantitativen Ziele abermals behindert. Damit zeigt sich deutlich, daß die Ziele des V-PK-Managements nur dann erreicht werden können, wenn das gesamte Management mit allen Instrumenten konsequent an die Anforderungen von V-PK-Systemen angepaßt wird. In Kapitel 7 wird der in dieser Arbeit neu entwickelte Übersetzer beschrieben, der von INSEL unmittelbar in ein hoch optimiertes Maschinenprogramm transformiert und damit die Schwächen der Transformation in AdaM und EVA eliminiert.

Der ermutigende Erfolg, der mit der noch einfachen Integration von Übersetzeranalysen in AdaM erzielt wurde, motiviert die Systematisierung dieses Ansatzes und seine Ausdehnung auf bidirektionale Informationsflüsse (unidirektional in AdaM) und zusätzlicher Koordinierung von Entscheidungen. Gleiches gilt für den Ansatz der Flexibilisierung, der in AdaM exemplarisch exerziert wurde und für das V-PK-Management weiter ausgearbeitet wird. Von Interesse ist ferner eine Generalisierung des zweistufigen Managementkonzepts, das in AdaM zur Speicherallokation entgegen der Top-Down Ausrichtung, an den Stellen orientiert, entwickelt wurde. Das Konzept von Ressourcenpools pro Stelle eignet sich unter bestimmten Voraussetzungen gut, um den Übergang von der abstrakten $\alpha$-Struktur auf die physische Struktur der Hardwarekonfiguration effizient zu gestalten. In der vorliegenden Arbeit wird dieser Ansatz in die Systematik der Realisierung abstrakter Manager eingeordnet.

### 4.3.4 Experimentalsystem 4: Shadow

Das vierte Experimentalsystem wurde auf der SUN V9 Architektur unter SUN Solaris entwickelt. Im Zentrum des Interesses stand – in Fortsetzung der AdaM-Arbeiten – die weitere Flexibilisierung der Verwaltung des physisch verteilten, gemeinsamen Speichers sowie die Modellierung des Ressourcenmanagements mittels Agenten und Graphersetzungssystemen.

Im Bereich der Grundlagen wurde ein Systemmodell entwickelt, das alle Abstraktionsebenen von abstrakt bis zur Hardware einschließt und die Komponenten des Systems nach den Dimensionen Raum und Abstraktionsebene strukturiert. Gemäß dieser Gliederung werden

horizontale und vertikale Ressourcen unterschieden. In beiden Fällen ergeben sich ihre Eigenschaften aus statisch spezifikatorischen und dynamischen Nutzungseigenschaften. Je nach Einordnung entlang der Abstraktionsebenenachse besitzen Ressourcen mehr oder weniger Freiheitsgrade. Die Reduzierung der Freiheitsgrade mit dem Ziel der vollständigen Festlegung einer Realisierung entspricht dem Durchwandern eines Trichters, in dessen Verlauf sukzessive Information hinzugenommen, Entscheidungen getroffen und Freiheitsgrade aufgegeben werden. Ressourcen sind stets entweder gebunden oder ungebunden. Mittels der Bindung von Ressourcen wurde das System als ein Graph modelliert, dessen Knoten Ressourcen und Kanten Bindungen sind.

Bezüglich der partiellen Realisierung der MoDiS-Architektur, wurden die abstrakten Manager als mobile, autonome und kooperierende Agenten realisiert. Faktisch handelt es sich dabei, ähnlich wie bei AdaM, um C-Datenstrukturen, die allerdings aktiv durch einen oder mehrere Solaris-Threads realisiert sind. Im Gegenzug werden Akteure in der Regel nicht durch einen eigenen Thread realisiert, sondern erhalten ihre Rechenfähigkeit von dem Manager-Agenten-Thread. Das Ergebnis ist somit ähnlich aber nicht identisch mit AdaM, denn Shadow sieht zusätzliche Manager-Agenten vor, z. B. den Netz-Kommunikations-Server *Communication Manager*, die nicht mit Akteuren assoziiert sind. Probleme der Übersetzung mittels einer Hochsprache haben sich bei Shadow nicht gestellt, da anstelle von INSEL ein C-Dialekt und ein optimierender C-Übersetzer gewählt wurde. Im Rahmen der Speicherverwaltung wurde der gemeinsame Speicher mit dem effizienten, objektbasierten DSOM-System *Shadow Stack* realisiert, das die Fähigkeiten gewöhnlicher, seitenbasierter Hardware zur Erkennung von Zugriffen auf individuelle, feingranulare Objekte nutzt [GPR97]. Shadow Stack ermöglicht es, pro Objekt ein Konsistenzmodell sowie ein Kohärenzprotokoll festzulegen. Im Gegensatz zur zweistufigen Gliederung der Speicherallokation in AdaM (SV, WV) wurde das Schema von Shadow stark an die abstrakte $\alpha$-Struktur der Akteure angelehnt. Manager-Agenten kommunizieren rekursiv entlang der $\pi$-Struktur mit ihren Vätern und Söhnen um Speicheranforderungen zu erfüllen. Auf diese Weise wird der – auf SUN V9 64 Bit weite – Adreßraum, unabhängig von den Stellen, dynamisch partitioniert.

#### 4.3.4.1 Bewertung

Die uniforme Modellierung des Systems mit Ressourcen und Abhängigkeiten ist eine wichtige Maßnahme, um Ressourcenmanagement-Systeme besser beschreiben und verstehen zu können. Nach wie vor ist auch dieses Modell von der Dreiteilung in Anwendungs-, Management- und Hardwareobjekte geprägt, wenngleich mehrere Ebenen von Managementobjekten vorgesehen sind. Zahlreiche Komponenten und ihre Abhängigkeiten entziehen sich durch diese grobe Einteilung einer systematischen Betrachtung. Die Ursachen für die Tendenz zur Dreiteilung sind jedoch nicht in Shadow zu suchen sondern genereller Art. Dazu gehört auch die Beschränktheit natürlicher Sprache, die das Benennen zahlreicher verschiedener Abstraktionsebenen behindert. In der vorliegenden Arbeit wird das Shadow Systemmodell weiterentwickelt. Es wird ein generischer Namensraum entwickelt, die Begriffe Ressourcen und Ressourcenklassen neu definiert und die wichtige Dimension Zeit in dem neuen Systemmodell mit aufgenommen. Der Agentenansatz wird dahingegen von einer differenzierteren Betrachtung der Realisierung abstrakter Manager ersetzt. Dies erscheint zweckmäßiger als die Einführung einer zweiten separaten Verwaltungshierarchie.

Der objektbasierte Speicher von Shadow ist äußerst effizient und erzeugt keinen nennenswerten konstanten lokalen Mehraufwand. Das Programmiermodell ist allerdings mangels der

Integration des Übersetzers kompliziert und erfordert die Verwaltung eines zweiten Kellers im Programm sowie explizite Abschlußsynchronisation zur Gewährleistung der Lebenszeiten, die von Shadow Stack vorausgesetzt werden. Dies unterstreicht abermals die Notwendigkeit für eine durchgängige Konstruktion von V-PK-Systemen von der Sprache bis zum Kern, um sowohl Qualität als Quantität erzielen zu können. Die Frage, ob das sorgfältig ausgearbeitete Verfahren zur Erkennung von Zugriffen auf entfernte Datenobjekte mittels Hardware Effizienzvorteile gegenüber einer Softwareimplementierung besitzt, blieb offen. Abhängigkeiten zwischen Objekttransporten und dem Scheduling machten zusätzliche Softwarekontrollen erforderlich. Der Unterschied zwischen der hardwarebasierten Lösung und einer wesentlich einfacheren Lösung in Sofware – die vollständige Kontrolle über den Übersetzer erfordert hätte – liegt damit nur noch bei $\pm 2$ Maschinen-Instruktionen pro Zugriff auf ein Datenobjekt. Die Lehren für das V-PK-Management sind zum einen die Beobachtung, daß die Konsequenzen von Entscheidungen auch bei sorgfältiger Planung aufgrund vielfältigster Abhängigkeiten ohne eine geeignete Systematik schwer abzusehen sind und zum anderen seitens transformatorischer Maßnahmen großes Potential existiert, mit dem Probleme in verteilten Systemen effizient gelöst werden können. Der Ansatz der rekursiven Kooperation entlang der $\pi$-Struktur wird in der vorliegenden Arbeit als zweite Alternative zu den Ressourcenpools von AdaM betrachtet. Je nach Anforderung besitzen beide Konzepte Vor- bzw. Nachteile. Beide Ansätze müssen allerdings neben weiteren Möglichkeiten systematisch in das Repertoire der Realisierung abstrakter Manager eingeordnet werden, um sie später anhand fundierter Kriterien adäquat einsetzen zu können.

## 4.4 Fazit

Zum Abschluß der umfangreichen Analysen im Teil I dieser Arbeit und der Recherchen bezüglich dem Stand der Forschung in diesem Kapitel werden hier noch einmal die wichtigsten Beobachtungsergebnisse zusammengefaßt sowie Anforderungen an das V-PK-Management und die Systematik seiner Konstruktion formuliert:

- *Verteilte Systeme sind nach wie vor eine zu nutzende Chance* [Spi89]. Keiner der beschriebenen Ansätze ist in der Lage, verteiltes Rechnen auf das Niveau eines Standardfalles anzuheben. Ergänzungen wie PVM und DCE bieten Effizienz aber keine geeigneten Konzepte zur Spezifikation von Parallelität und Kooperation und werden deshalb als zu kompliziert empfunden. Sprachbasierte Ansätze, wie Orca und EVA, erzielen bislang wiederum keine befriedigende Effizienz. Systeme wie DIPC, Sprite und Comandos überzeugen weder mit geeigneten Sprachkonzepten noch durch Effizienz. Als Gründe sind unklare Zielsetzungen, Kompromisse aufgrund Kompatibilitätszwang und Aufwand, oder das Vernachlässigen von Details zu nennen. Offensichtlich ist der Übergang zu verteilten Systemen schwieriger als vermutet und mit partiellen Erweiterungen nicht zu bewerkstelligen. Benötigt wird stattdessen, eine konsequente und durchgängige Entwicklung geeigneter Sprachkonzepte und zugehöriger Realisierungsverfahren, die sich von bestehendem loslöst.

- Es herrscht ein Mangel an *Integration* von Maßnahmen des Ressourcenmanagements. DTK-Bidding, Shadow Stack DSOM oder Amoeba FLIP bieten leistungsfähige Lösungen zu wichtigen Detailproblemen, deren Potential durch eine horizontal oder vertikal mangelhafte Integration in die Gesamtarchitektur jedoch nicht zur Entfaltung kommt.

Ferner fußen die Ineffizienzen sprachbasierter Ansätze bereits auf einem konstanten lokalen Mehraufwand, der vor allem auf die aufwandsgetriebene, unmodifizierte Übernahme existierender Konzepte und Verfahren basiert (z. B. Zwischenhochsprache + existierender Übersetzer). Erste Versuche der engeren Integration von Managementinstrumentarien, siehe Diamonds, Concert und AdaM, zeigen schon bei unidirektionalen Informationstransfers großen Erfolg.

- Die Überwindung von Stellengrenzen erfordert ein hohes Maß an *Flexibilität*, da die Unterschiede der Kommunikationsleistung zwischen nah und fern gravierend sind. Uniforme Verfahren, die lokal zufriedenstellende Leistung erzielen, wie das Verfolgen von Verweisketten, sind verteilt u. U. inakzeptabel. Umgekehrt darf die Verteilung nicht zum Anlaß genommen werden, bislang effiziente Verfahren durch auf den verteilten Fall verallgemeinerte – zu ersetzen, da auch dies zu einem nicht kompensierbaren Mehraufwand führt, siehe Parameterübergabe über die Halde anstelle in Registern. Ansätze, wie die dynamische Replikation in AdaM, variable Granularität bei Shadow oder der dynamische Austausch von Realisierungen bei Muse/Apertos, belegen die Wichtigkeit der Flexibilisierung über Raum und Zeit für erfolgreiches, verteiltes Ressourcenmanagement.

- Das Potential *transformatorischer Maßnahmen* (Übersetzer & Binder) wird im Bereich der Betriebssysteme generell unterschätzt. Zum einen beruht dies auf der historisch gewachsenen Fehleinschätzung, daß der Kern das Betriebssystem ist – siehe Amoeba, Sprite und Mikrokerne bzgl. Erweiterbarkeit. Zum anderen zwingt Aufwand und mangelnde Information zur Bevorzugung gewohnter Vorgehensweisen: Änderungen an Kernen oder Ergänzung von Laufzeitsystemen. Übersehen wird dabei, daß die Maßnahmen von Kern und Laufzeitsystem in den repetitiven Teil-Interpretierer $i_{std}$ (Abschnitt 2.3.3.2 auf S. 58) eingehen und eine transformatorische Realisierung möglicherweise geeigneter wäre — siehe Pfadinterpretation in EVA.

- Die Wurzel für viele der genannten Probleme ist die *wenig systematische Konstruktion von Betriebssystemen*, die kaum über Empirie, trial & error oder ad hoc Entscheidungen hinausgeht. Alternativen, Abhängigkeiten in Raum und Zeit sowie Wechselwirkungen zwischen Entscheidungen sind meist zunächst unbekannt. Unerwünschte Eigenschaften werden wesentlich später als Phänomene beobachtet, woran sich ggf. Versuche der lokalen Optimierung anschließen. Dies alles ist weit entfernt von einer systematischen und methodisch fundierten Vorgehensweise, mit der die Komplexität der Aufgabe und der verbundene Realisierungsaufwand beherrscht werden könnte. Das Beispiel der Threads illustriert bereits die Wichtigkeit der Entscheidung zwischen Fehlerverhinderung kontra -vermeidung, für die es zunächst keine methodische Grundlage gibt. Nicht weiter differenzierte Versuche der Strukturierung wie Schichtung oder die Dreiteilung Anwendung/Manager/Plattform in den Experimentalsystemen erfassen die vielfältigen Komponenten und Abhängigkeiten nicht im erforderlichen Umfang und induzieren inadäquate Entscheidungen, wie die wenig differenzierte Realisierung abstrakter Parallelität mittels einem starren Thread-Konzept auf Kern-Niveau.

Aus diesen Ergebnissen ergeben sich folgende Anforderungen und Festlegungen für das zu konstruierende V-PK-Management, von dem erwartet wird, daß es trotz des hohen Abstraktionsniveaus von INSEL auch quantitative Leistungsfähigkeit erzielt:

1. Die Instrumente des Managements müssen eng zu einem *integrierten* Gesamtmanagement verwoben werden. Unidirektionale Informationsflüsse sollten systematisch genutzt und auf Bidirektionalität erweitert werden. Weiter müssen die Maßnahmen aller Instrumente des V-PK-Managements sorgfältig aneinander angepaßt werden, um Reibungsverluste zu eliminieren. Die Verbesserung der Informationsgrundlage durch Integration ist zudem eine wichtige Voraussetzung für die Flexibilisierung des Managements.

2. Das V-PK-Management muß systematisch *flexibilisiert* werden. Es werden sowohl lokal effiziente Verfahren als auch ggf. zusätzliche, entfernt effiziente Verfahren benötigt, um zum einen konstanten lokalen Mehraufwand abzuwenden und zum anderen Stellengrenzen zu überwinden. Bei Bedarf ist das Realisierungsspektrum um weitere charakteristische Alternativen zu erweitern. Die Flexibilisierung des Managements muß dediziert erfolgen, um Bindungen mit der geeigneten Effizienz festlegen zu können und ineffiziente, generalisierte Verfahren, wie die stets indirekten Unterprogrammaufrufe in Napier88, zu vermeiden. Die Granularität von Ressourcen sowie deren Kompositionalität muß ferner abgestuft festgelegt werden.

3. Für das V-PK-Management muß ein neues, geeignetes *Transformationsinstrumentarium* entwickelt werden. Der Übersetzungsvorgang darf sich nicht auf eine Zwischensprache plus übernommenen Übersetzer stützen, sondern muß vollständige Kontrolle über die INSEL-Quellcode Transformation in ein Maschinenprogramm ermöglichen. Ähnliches gilt für den Binder, dessen Relokation ebenfalls vollständig kontrolliert und reversibel gestaltet werden muß, um die oben geforderte Flexibilität zu erzielen. Die Zuordnung von Aufgaben an das Transformationsinstrumentarium ist im Zuge der Integration und dem Aufheben der Trennung zwischen Statik und Dynamik zu überdenken.

4. Die Realisierung der abstrakten Manager muß wesentlich stärker als bisher *differenziert* werden. Die intuitive Dreiteilung führt zu uniformen, grobgranularen und schwach strukturierten Managerobjekten. Stattdessen müssen Abstraktionsniveaus festgelegt und sorgfältig getrennt werden, um wichtige Fragen, z. B. aktive oder passive Manager, beantworten und unbewußte Entscheidungen mit weitreichenden Konsequenzen, minimieren zu können. Wie die Parameterübergabe in EVA und die Abschlußsynchronisation in AdaM lehren, besitzen vermeintliche Details weitreichende Konsequenzen und dürfen deshalb trotz des hohen Aufwands nicht außer Acht gelassen werden.

5. Die Frage, *„Wie konstruiert man Management?“* hat in Anbetracht des Defizites an Systematik im Bereich der Betriebssysteme Vorrang vor der Entwicklung weiterer effizienter verteilter Verfahren. Die Bearbeitung dieser Frage ist allerdings umfangreich und ihre Beantwortung ist bestenfalls langfristig zu erwarten. Dennoch eignen sich die Bestrebungen zur engeren Integration und stärkeren Flexibilisierung, um neue Sichtweisen auf Betriebssysteme und deren Konstruktion zu erarbeiten. In diesem Zusammenhang soll ein Ansatz zur Systematisierung erarbeitet werden. Im Zentrum dieses Ansatzes muß die präzise Beschreibung des Systems als dynamisches Netz von Komponenten und Abhängigkeiten mit einer homogenen Gesamtsprache stehen.

Bezüglich der Erweiterbarkeit von Systemen hat sich gezeigt, daß dieses Thema im Bereich der Betriebssysteme bislang meist mechanistisch bearbeitet wird. Die Verankerungen in der Sprache sind, abgesehen von Napier88, schwach ausgearbeitet. Im Kontext der top-down orientierten Vorgehensweise in MoDiS ist zu fordern, daß die Erweiterbarkeit des Systems mit

geeigneten neuen INSEL-Konzepten auf Sprachniveau verankert wird. Die neuen Konzepte müssen sich zudem homogen in die bereits bestehenden einordnen.

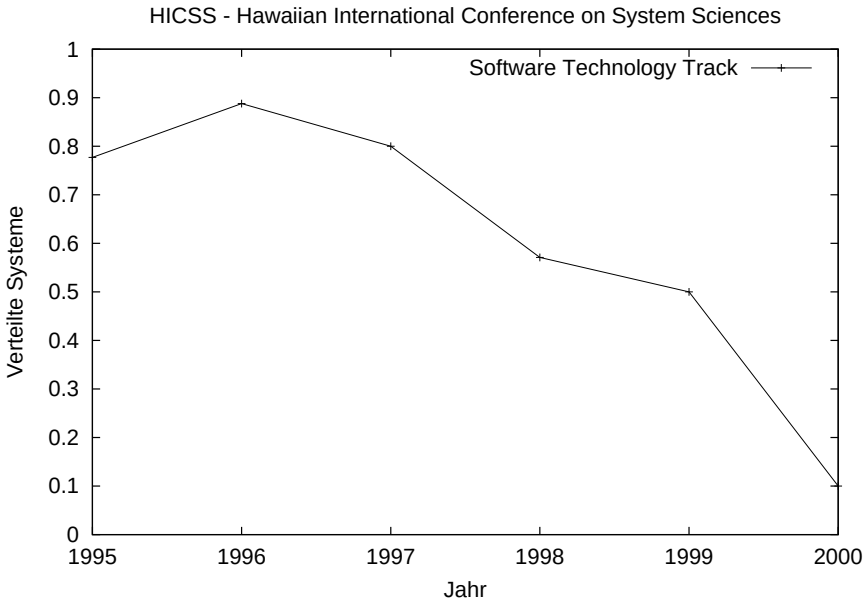


Abbildung 4.3: Interesse an parallelen und verteilten Systemen

Als Schlußbemerkung von Teil I sei darauf hingewiesen, daß die Bemühungen, verteiltes Rechen zu ermöglichen, weltweit rückläufig erscheinen. Diese Vermutung erhärtet sich bei einer Analyse der Themen großer internationaler Konferenzen. Exemplarisch wird hier die Konferenz HICSS (Hawaiian International Conference on System Sciences) betrachtet, die bereits seit 1967 jährlich mit ca. 500 (Stand 1997) Teilnehmern zu den renommierten, großen Veranstaltungen gehört. Abbildung 4.3 stellt für den Zeitraum 1995 bis 2000 den Anteil verteilter Systeme an der Gesamtheit der im HICSS *Software Technology Track* behandelten Themen dar. 1996 wurden in beinahe neun von zehn Sitzungen verteilte Systeme diskutiert. 1998 waren es noch etwa fünf von zehn. Zur Jahrtausendwende wird das nur noch in einer von zehn der Fall sein. Der Rückgang des Interesses liegt in Anbetracht des recherchierten Standes der Forschung nicht daran, daß die Probleme zufriedenstellend gelöst seien. Vielmehr ist Resignation als Ursache zu vermuten. Zusätzlich haben sich neue Themen entwickelt, in deren Kontext schneller Fortschritte erzielt werden können, z. B. Umfeld Internet. Weitere Hinweise auf Resignation sind bei Firmen zu erkennen, die noch Anfang und Mitte der 90er Jahre mit großen Anstrengungen versucht haben, neue, verteilte Betriebssysteme zu realisieren und inzwischen wieder zur Weiterentwicklung zentraler Systeme zurückgekehrt sind. Ein Beispiel ist das objekt-orientierte verteilte Betriebssystem Spring [HK93, RHKP95] der Firma SUN, das nahezu vollständig realisiert wurde. Erst sehr spät stellten sich gravierende Leistungsmängel heraus, die zum Abbruch des Projekts führten.

Offensichtlich konnten die hohen Erwartungen an verteilte Systeme bislang nicht erfüllt werden. Die Wurzeln für das Scheitern sind mangelnde Methodik und enormer Aufwand, die sich konkret u. a. in ungenügender Flexibilität, schwacher Differenzierung und partiellen Lösungen, d. h. Ignoranz wichtiger Abhängigkeiten manifestieren. Die Verbesserung

dieser ungünstigen Situation ist unerläßlich und erfordert eine langfristige Evolution der Betriebssystem-Konstruktion.

# Teil II

# Synthese

# Kapitel 5

# Rahmenkonzepte

Die Analysen im Teil I ergaben, daß Versuche der Konstruktion von Managementsystemen und dabei insbesondere verteilter Managementsysteme, generell an der mangelnden Kenntnis der Komponenten des Systems und ihrer Abhängigkeiten leiden. Der Schlüssel zur Lösung vieler Probleme, angefangen bei Ineffizienzen und qualitativen Mängeln über die unbrauchbar flache Struktur von Design Patterns und den unbefriedigenden Strukturierungsansätzen mittels Schichten und Mikrokernen bis hin zu dem hohen Realisierungsaufwand im Bereich der Systemprogrammierung, liegt somit in dem systematischen Umgang mit hoch dynamischen Netzen von Komponenten.

Zum Abschluß von Teil I wurde deshalb ein Ansatz zur Systematisierung von Managementsystemen gefordert, um langfristig das bislang empirische Vorgehen durch eine fundierte Methodik abzulösen. Managemententscheidungen in komplexen Systemen müssen präzise beschrieben, diskutiert und zielgerichtet gefunden werden können. Mit einer geeigneten Systematik läßt sich auch der Realisierungsaufwand reduzieren, da langwierige Iterationen über Design, Implementierung und Evaluation von einer analytischen Entscheidungsfindung ersetzt werden und zusätzlich ein Anker für generative Realisierungsverfahren vorhanden wäre. Der Weg dorthin ist weit jedoch unerläßlich, um weitere Fortschritte im Bereich der Betriebssysteme erzielen zu können, denn im Gegensatz zur methodischen Konstruktion von Übersetzern oder der Normalformentheorie bei den Datenbanken besteht hier enormer Nachholbedarf. Mit der Festlegung von Rahmenkonzepten und der Entwicklung eines integrierten Systemmodells wird in diesem Kapitel der Versuch unternommen, einen erster Schritt auf diesem Weg zu vollziehen. Die Ziele, die mit dem hier entwickelten Modell verfolgt werden, sind erstens, einen anschaulichen Umgang mit komplexen Systemen zu ermöglichen und somit das Verständnis über das Zusammenwirken vielfältiger, abhängiger Komponenten zu verbessern und zweitens, die Untermauerung der Argumentation in den noch folgenden Kapiteln.

## 5.1 Ressourcen

Ein zentraler Begriff bei der Betrachtung von Managementsystemen ist der Begriff *Ressource*[1], der auch in dieser Niederschrift bereits mehrfach verwendet wurde, ohne zuvor präzise eingeführt worden zu sein. Das Spektrum der Auffassung über das, was Ressourcen sind, ist groß. Oft wird „Ressource“ auf die Betriebsmittel von Geräten beschränkt, wobei diese enge

[1] fr.-lt.: Betriebsmittel

Sichtweise gelegentlich weiter auf bestimmte Hardwareressourcen, z. B. Prozessoren, eingeschränkt wird. Diese Auffassung separiert eine spezifische Abstraktionsebene – die physische – von allen anderen Komponenten des Systems, die ebenso zur Durchführung der Berechnungen beitragen und in einem engen Bezug zu den Hardwareressourcen stehen. Die a priori Isolation einzelner Komponenten anhand intuitiver Kriterien, wie dem Kriterium real oder abstrakt, verleitet dazu, Kontexte dauerhaft von der Betrachtung auszuschließen. Durch das Ignorieren von Kontexten entstehen in Folge Bruchstellen in der Betrachtung der Entscheidungen und Maßnahmen, die in einem System zusammenwirken. Die negativen Konsequenzen sind u. a. Redundanz, Heterogenität, mangelnde Kenntnis über alternative Möglichkeiten und vieles mehr. Die Verallgemeinerung des Begriffs Ressource ist der erste wichtige Schritt, um Probleme dieser Art grundlegend zu vermeiden. „Ressource“ wird in dieser Arbeit benutzt, um sämtliche Komponenten eines Systems, die auf unterschiedliche Weise zur Durchführung der Berechnungen beitragen, einheitlich zu erfassen.

### 5.1.1 Klassenbildung

Eine Grundvoraussetzung, um Ressourcen produzieren und mit ihnen agieren zu können, ist die Bildung von Klassen mit gemeinsamen Eigenschaften. Von den Elementen einer Ressourcenklasse wird gefordert, daß sie auf identische Weise nutzbar sind. Für einen potentiellen Nutzer sind zwei Ressourcen $a$ und $b$ genau dann identisch nutzbar, wenn die für die Nutzung interessierenden Eigenschaften von $a$ und $b$ abgesehen von über die Zeit veränderbaren Unterschieden äquivalent sind. Die Kriterien, die zur Klassenbildung herangezogen werden, beziehen sich deshalb in aller Regel auf die invarianten Eigenschaften von Ressourcen.

**Definition 5.1 (Eigenschaften und Invarianten)**
*Gegeben sei eine Menge $U$ von Elementen eines Systems,*

- *Prädikate mit der Signatur $e : U \rightarrow \mathbb{B}$ definieren Eigenschaften der Elemente aus $U$. Ein $u \in U$ hat die Eigenschaft $e_i$, genau dann, wenn $e_i$ für $u$ gilt, andernfalls besitzt $u$ nicht die Eigenschaft $e_i$.*
- *Eine invariante Eigenschaft, kurz Invariante, ist ein Prädikat, das für das betrachtete $u \in U$ für die Dauer der Existenz von $u$ gilt.*
- *Die Gültigkeit variabler Eigenschaften hängt im Gegensatz zu Invarianten vom Betrachtungszeitpunkt ab.*

Definition 5.1 ist bewußt allgemein gehalten und beschränkt sich nicht auf Ressourcen, da neben den Ressourcen auch die Bindungen, die im Abschnitt 5.2 noch eingeführt werden, ebenfalls Eigenschaften besitzen. Auf ein System sind gleichzeitig unterschiedliche Sichtweisen möglich und sinnvoll. Das Prinzip der Klassenbildung muß deshalb flexibel genug sein, um unterschiedliche Sichtweisen zu ermöglichen. Dieser Flexibilität wird mit Definition 5.2 Rechnung getragen.

**Definition 5.2 (Ressourcenklasse)**
*Sei $\mathcal{R}$ die Menge aller Ressourcen, $K_E \subseteq \mathcal{R}$ eine Teilmenge von $\mathcal{R}$ und $E$ eine Menge invarianter Eigenschaften von Ressourcen,*

$$K_E \text{ ist eine Ressourcenklasse} \Leftrightarrow \forall k \in K_E : \forall e \in E : e(k)$$

Die gewählten Eigenschaften $E$ sind die Kriterien der Zuordnung zu Klassen. Nach Definition 5.2 ist es möglich, daß eine Ressource gleichzeitig mehreren Klassen angehört. So gehört ein Register eines Prozessors sowohl zur Klasse der physischen Speicherressourcen als auch zur Klasse der physischen Prozessorressourcen, wobei die beiden Klassen allerdings nicht identisch sind. Diese mehrfache Zugehörigkeit entspricht dem Einnehmen verschiedener Sichten auf das System.

Wichtige Formen nicht disjunkter Klassen sind *Ressourcensuperklassen* und *Ressourcensubklassen.*

**Definition 5.3 (Ressourcensub- und superklasse)**
*Seien $\mathcal{RK}$ die Menge aller Ressourcenklassen, $rk_1, rk_2 \in \mathcal{RK}$ und $E_{rk_1}, E_{rk_1}$ die Klasseneigenschaften der Ressourcen in $rk_1$ bzw. $rk_2$,*

$$rk_1 \text{ ist eine Ressourcensuperklasse von } \Leftrightarrow E_{rk_1} \subseteq E_{rk_2} \wedge E_{rk_2} \neq E_{rk_1}$$
$$rk_1 \text{ ist eine Ressourcensubklasse von } rk_2 \Leftrightarrow rk_2 \text{ ist eine Ressourcensuperklasse von } rk_1$$

*Superklassen heißen auch Basisklassen.*

Das Prinzip der Bildung von Super- und Subklassen induziert eine Schachtelungshierarchie über Ressourcenklassen, die geeignet ist, um den Detaillierungsgrad der Betrachtung je nach Bedarf zu variieren (s. Abb. 5.1).

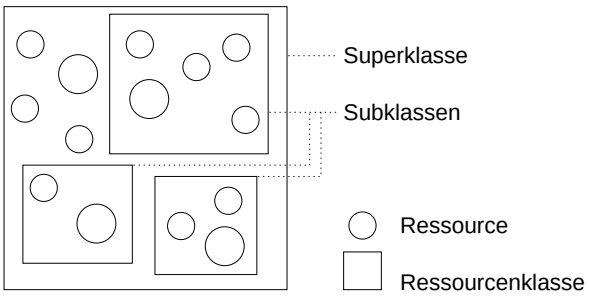


Abbildung 5.1: Ressourcen, Super- und Subklassen

Mit dem bis hierhin erläuterten Prinzip der Klassenbildung können bereits zwei wichtige Klassen eingeführt werden. Aus der Sicht der Betriebssysteme besteht Interesse an Rechensystemen, die auf konkreter Hardware realisiert sind. Die dafür benötigte, ausgezeichnete Klasse von Ressourcen ist die Klasse der physischen - bzw. *Hardwareressourcen.*

**Definition 5.4 (Physische Ressourcen)**
*Die Klasse $\mathcal{H}$ der physischen Ressourcen bzw. Hardwareressourcen umfaßt sämtliche speicher- und rechenfähigen Komponenten, die von den physischen Prozessoren zur Nutzung angeboten werden. Für die Eigenschaften $E$ der Klassenbildung gilt: $E = \{e_{\mathcal{H}}\}$ mit*

$$e_{\mathcal{H}}(r) = \begin{cases} true : r \text{ ist eine nutzbare, physische Komponente des Rechensystems} \\ false : sonst \end{cases}$$

In die Klasse der physischen Ressourcen $\mathcal{H}$ fallen unter anderem:

- Instruktionen der Befehlssätze der Prozessoren
- Register der Prozessoren
- Zellen und Seiten des Arbeitsspeichers
- Blöcke des Hintergrundspeichers
- ggf. virtuelle Adressen, falls diese von der Hardware angeboten werden

Bei der top-down orientierten Vorgehensweise existiert neben den Ressourcen der Hardware auf niedriger Abstraktionsebene eine zweite wichtige Klasse vorgegebener Ressourcen auf höherem Abstraktionsniveau. Dies sind die Mittel, die der Umwelt des Rechensystems zur Spezifikation von Problemlösungen angeboten werden und aus der Sicht der Benutzer operationalisiert sind. Im Falle von MoDiS handelt es sich dabei um die INSEL-Konzepte, mit deren Hilfe INSEL-Problemlösungen formuliert werden können. Da die Ressourcen dieser Klasse die Nutzungsmöglichkeiten des Rechensystems definieren, werden sie im folgenden als *Nutzungsressourcen bezeichnet.*

**Definition 5.5 (Nutzungsressourcen)**
*Die Klasse $\mathcal{N}$ der Nutzungsressourcen eines Systems umfaßt sämtliche abstrakten speicher- und rechenfähigen Komponenten eines Systems, die der Umwelt des Systems zu dessen Nutzung zur Verfügung stehen.*

Bei INSEL gehören zu den Nutzungsressourcen unter anderem die verschiedenen Komponentenarten als Hüllen für die Entwicklung von Komponenten, Kompositionsregeln sowie sämtliche Anweisungen, die zur Definition von Anweisungsteilen in INSEL zur Verfügung stehen. Zusammen definieren die Hardwareressourcen und die Nutzungsressourcen die beiden äußeren Klammern eines Systems. Zur Vereinfachung der noch folgenden Beobachtungen wird von den Mengen $\mathcal{H}$ und $\mathcal{N}$ angenommen, daß sie während der Lebenszeit des Systems konstant sind.

Während die Ressourcen aus $\mathcal{N}$ den Rahmen für mögliche Anforderungen an das Rechensystem definieren, stellen die $r \in \mathcal{H}$ die real vorhandenen Angebote zur Erfüllung von Anforderungen dar. Zwischen diesen beiden Extremen ist in aller Regel eine unmittelbare Korrespondenz weder möglich noch sinnvoll. Sattdessen werden weitere Ressourcen und Ressourcenklassen benötigt, die zwischen den Anforderungen und den Angeboten vermitteln.

Die Festlegung wohldefinierter Klassen von Zwischenressourcen zur anschließenden Produktion der benötigten Bindeglieder ist die primär wichtige Aufgabe der Systemkonstruktion, da ein Großteil der Managemententscheidungen bereits bei dieser Klassenbildung gefällt wird. Im folgenden Abschnitt wird zunächst der Begriff der Ressource präzisiert. Aus dieser Präzisierung leiten sich anschließend die Möglichkeiten, Klassen von Zwischenressourcen kreativ und konstruktiv zu definieren, unmittelbar ab.

### 5.1.2 Produktion von Zwischenressourcen

$\mathcal{H}$ und $\mathcal{N}$ markieren die Schnittstellen des System zu seiner Umwelt. Ressourcen innerhalb des Systems, die nicht Elemente aus $\mathcal{H} \cup \mathcal{N}$ sind, gehören der Klasse der *Zwischenressourcen* $\mathcal{Z}$ an. Offensichtlich werden Ressourcen aus $\mathcal{Z}$ ausschließlich aus den *vorgegeben Ressourcen* $\mathcal{V} = \mathcal{H} \cup \mathcal{N}$ gefertigt. Dies führt zu der rekursiven Definition von Ressourcen gemäß 5.6.

**Definition 5.6 (Ressourcen)**
*Sei $\mathcal{S}$ ein Rechensystem, $\mathcal{H}$ die Menge der Hardwareressourcen und $\mathcal{N}$ die Menge von Nutzungsressourcen von $\mathcal{S}$. Die Menge aller Ressourcen $\mathcal{R}$ in $\mathcal{S}$ ist rekursiv definiert als:*

*1) $\varepsilon$ ist eine Ressource und bezeichnet die leere Ressource*

*2) $\forall r \in \mathcal{H} \cup \mathcal{N} : r$ ist eine vorgegebene Ressource*

*3) $\hat{r}$ ist eine Ressource $\Leftrightarrow r$ ist eine Ressource; $\hat{r}$ bezeichnet die Referenzressource auf $r$*

*4) $\vec{r}_n = (r_1, \ldots, r_n)$ ist eine Ressource $\Leftrightarrow \forall r_i : r_i, i = 1, \ldots, n$ sind Referenzressourcen; $\vec{r}_n$ heißt $n$-Tupelressource*

**Definition 5.7 (Äußere Operationen von Ressourcen)**
*Ressourcen besitzen äußere Operationen, über die sie von außen genutzt werden. Die äußeren Operationen einer Ressource $r$ sind:*

1. *$\emptyset \Leftrightarrow r = \varepsilon$*
2. *gesondert spezifiziert, falls $r \in \mathcal{H} \cup \mathcal{N}$*
3. *{„referenziere“,„löse_auf“} $\Leftrightarrow r$ ist eine Referenzressource*
   *Der Aufruf von „referenziere“ auf $r$ mit Ressource $t$ als Parameter bewirkt $r = \hat{t}$;*
   *„löse_auf“ bewirkt die Auflösung der Ressource.*
4. *{„selektiere“,„löse_auf“} $\Leftrightarrow r$ ist eine Tupelressource*
   *Die Ausführung von „selektiere“ mit Parameter $n \in \mathbb{N}$ liefert als Ergebnis die $n$-te Referenzressource aus $r$;*
   *„löse_auf“ bewirkt die Auflösung der Ressource.*

Referenzressourcen verweisen auf andere Ressourcen des Systems und Tupelressourcen bestehen ausschließlich aus Verweisen auf andere Ressourcen. Der Grund für die Festlegung von Ressourcen auf diese Art ist, daß jede Ressource aus $\mathcal{V}$ per Definition genau einmal im System existiert. Wären beliebige Ressourcen als Elemente von Tupelressourcen erlaubt, dann könnte eine Tupelressource per Definition nur so lange existieren, wie jedem Tupelelement eine Ressource zugeordnet ist. Das würde bedeuten, daß dynamische Abläufe, wie die Seitenverdrängung, als Auflösung bestehender und Erzeugung neuer Tupel modelliert werden müßten. Dies wäre denkbar, würde allerdings dem eigentlichen Ziel, flexibel einsetzbare Ressourcen zu fertigen, widerstreben. Ferner gäbe es dann auch Ressourcen $v \in \mathcal{V}$, die gleichzeitig Element verschiedener Tupelressourcen $t_i$ wären, ohne daß der Zusammenhang zwischen den $t_i$ bezüglich $v$ in dem Modell brauchbar erfaßt wäre. Das Referenzkonzept scheint besser geeignet zu sein, um Abhängigkeiten dieser Art inklusive ihrer Dynamik zu modellieren.

Benötigt wird nun ein konstruktives Verfahren zur Generierung der erforderlichen Ressourcen eines Systems. Die Betrachtung und Festlegung individueller Ressourcen ist dafür nicht geeignet. Die Erzeugung von Ressourcen muß statt dessen strukturiert, d. h. auf Grundlage vorab festgelegter Ressourcenklassen mit wohldefinierten Eigenschaften erfolgen. Um dies zu ermöglichen werden die Eigenschaften von Ressourcen gemäß Definition 5.6 zunächst wie folgt erweitert:

**Postulat 5.8**

1. *Referenzressourcen sind mit Ressourcenklassen qualifiziert.*
   *Die Qualifikation einer Referenzressource legt die Klasse fest, auf deren Elemente die Referenzressource zusätzlich zu $\varepsilon$ verweisen kann.*

2. *Die Art einer Ressource, die Qualifikation im Falle von Referenzressourcen bzw. die Anzahl von Tupelementen im Falle einer Tupelressource sind invariante Eigenschaften einer Ressource.*

3. *Ressourcen werden mit Ausnahme der vorgegebenen Ressourcen aus $\mathcal{V} \cup \varepsilon$ stets auf Grundlage einer Klassenbeschreibung erzeugt, mit der die invarianten Eigenschaften der Ressourcen dieser Klasse präzise festgelegt sind.*

Um der Forderung nach der Determinierung von Klasseneigenschaften und der gezielten Produktion von Ressourcen auf der Grundlage der Klassenbeschreibung nachzukommen, wird das Konzept der Ressourcengeneratoren eingeführt.

**Definition 5.9 (Ressourcensignatur)**
*Eine Ressourcensignatur $S_{\mathcal{K}}$ für Ressourcen der Klasse $\mathcal{K} \in \mathcal{RK}$ legt folgende invarianten Eigenschaften von $\mathcal{K}$ fest:*

1. *Art der Ressource: Referenz oder Tupel inklusive Anzahl der Elemente*

2. *Die Qualifikation der Referenzressource bzw. Tupelelemente*

**Definition 5.10 (Ressourcengenerator)**
*Ein Ressourcengenerator $RG_{\mathcal{K}}$ für Ressourcen der Klasse $\mathcal{K} \in \mathcal{RK}$ ist eine Ressource, die im wesentlich eine Definition der Klasse $\mathcal{K}$ und eine äußere Operation zur Produktion von Ressourcen mit den $\mathcal{K}$-Eigenschaften besitzt. Bestandteile der Klassendefinition sind:*

- *Die Ressourcensignatur $S_{\mathcal{K}}$ für die Klasse $\mathcal{K}$ und*

- *eine möglicherweise leere Menge zusätzlicher Invarianten, die von Elementen aus $\mathcal{K}$ gefordert werden.*

Ressourcensignaturen würden alleine zur Spezifikation der gewünschten Ressourcenklassen nicht genügen. Sollen zum Beispiel die Elemente einer Klasse einer Ordnung gehorchen, so muß diese Forderung mit Klassenbeschreibung formulierbar sein. Ressourcensignaturen sind dazu nicht in der Lage. Mit dem zweiten Beitrag der Klassendefinition neben der Signatur können diese kreativen Festlegungen getroffen werden.

Mit den in diesem Abschnitt eingeführten Konzepten und Regelungen ist es nun möglich, die Menge der Ressourcen eines System auf Grundlage vorgegebener Hardware- und Nutzungsressourcen und kreativ festlegbarer Zusatzeigenschaften konstruktiv zu entwickeln. Die Konstruktion beginnt nach Festlegung der Mengen $\mathcal{H}$ und $\mathcal{N}$ mit der Definition der Klassendefinitionen für die benötigten Zwischenressourcen, mit deren Hilfe Ressourcengeneratoren angefertigt werden, die eine äußere Operation zur Erzeugung von Ressourcen mit den spezifizierten Klasseneigenschaften besitzen. Eine spezielle Kategorie von Resourcengeneratoren wurde im Kapitel 3 mit dem INSEL-Generatorkonzept bereits erläutert.

Nach der expliziten Erzeugung einer Ressource $r$ auf Grundlage eines Ressourcengenerators existiert $r$ und ist nutzbar, das heißt, anderen Ressourcen ist es möglich, auf $r$ zu verweisen. Es bleibt zu klären, wann $r$ aufgelöst wird. $r$ soll auch dann weiter existieren,

wenn sie vorübergehend nicht genutzt wird, d. h. kein $s \in \mathcal{R}$ mit $s = \hat{r}$ existiert. Das System wäre allerdings inkonsistent, wenn sich eine Referenzresource $s$ auf $r$ beziehen würde, während die Operation *löse_auf* von $r$ aufgerufen wird. Dieser Fall tritt in der Praxis auf und wird meist als irreparabler Fehlerzustand gewertet. Dieses Problem wird bei der Modellbildung in diesem Kapitel durch Postulat 5.11 ausgeschlossen, wenngleich dies eine Beschränkung der Allgemeinheit darstellt.

**Postulat 5.11 (Auflösung von Ressourcen)**

1. *Eine Ressource $r \in \mathcal{R}$ darf aufgelöst werden* $\Leftrightarrow \nexists s \in \mathcal{R} : s = \hat{r}$
2. *$r$ wird aufgelöst durch Aufruf seiner äußeren Operation* löse_auf

Diese Forderungen werden durch zusätzliche Einschränkungen für Ressourcen einer bestimmten Klasse durch entsprechende Klasseneigenschaften bei Bedarf weiter eingeschränkt. Ein Beispiel ist die FIFO[2]-artige Auflösung von Komponenten eines Kellers.

**Definition 5.12 (Lebenszeit einer Ressource)**
*Sei $t_1$ der Zeitpunkt der Erzeugung und $t_2$ der Zeitpunkt der Auflösung der Ressource $r \in \mathcal{R}$:*

$$\textit{Das Intervall } \Lambda(r) = [t_1, t_2] \textit{ ist die Lebenszeit der Ressource } r.$$

Die Definition der Lebenszeit von Ressourcen hier entspricht exakt der Definition des Begriffs Lebenszeit von INSEL DA-Komponenten auf Seite 76. INSEL DA-Komponenten sind selbst Ressourcen im Sinne von Definition 5.6.

## 5.2 Bindungen und Pfade

Mit den bisher eingeführten Konzepten können Mengen von Ressourcen spezifiziert und produziert werden. Das Zusammenwirken der Ressourcen ist, bis auf die Aussage, daß es Referenzierungen und Tupelbildung gibt, bislang nicht weiter beschrieben. Ein primitives System $\mathcal{S}_0$ könnte potentiell aus der Menge $\mathcal{V}$ und einer Menge von Zwischenressourcen, die sich alle ausschließlich auf $\varepsilon$ beziehen, bestehen. Da es in diesem System keinen Zusammenhang zwischen den Nutzungsressourcen und den physischen Ressourcen gäbe, wäre $\mathcal{S}_0$ ein nutzloses System. Das Ziel der Produktion von Ressourcen besteht demgegenüber in der Realisierung der mit $\mathcal{N}$ spezifizierten Möglichkeiten mit Hilfe der durch $\mathcal{H}$ gegebenen Angebote. Dazu muß eine Korrespondenz zwischen den $n \in \mathcal{N}$ und den $h \in \mathcal{H}$ hergestellt werden. Die erforderlichen Zusammenhänge zwischen den Ressourcen eines Systems sind vielschichtig und hoch dynamisch und werden mit den Konzepten der Bindungen und Bindungspfade – kurz Pfad – als Interpretation der Referenzressourcen modelliert.

Aus der Definition von Ressourcen folgt, daß jede Zwischenressource $z \in \mathcal{Z}$ neben den mit dem Ressourcengenerator festgeschriebenen invarianten Eigenschaften variable Eigenschaften besitzt, die sich aus den Fähigkeiten der Referenzressourcen ergeben. Durch wiederholte Ausführung ihrer Operation *referenziere*, sind Referenzressourcen in der Lage, im Laufe ihrer Existenz Bezüge zu unterschiedlichen anderen Ressourcen zu besitzen. Dies entspricht der über die Zeit variablen Nutzung von Ressourcen durch andere Ressourcen des Systems.

Hilfreich ist ferner die Vorstellung, daß der Verweis einer Referenzressource zeitweilig „offen“ sein kann, d. h. auf keine Ressource verweist. Der Ansatz, offene Referenzen zuzulassen,

[2] first in first out

ermöglicht die Produktion von Zwischenressourcen als flexibel einsetzbare Hüllen. Referenz- und Tupelressourcen sind somit zum einen auf Vorrat produzierbar, bevor konkreter Bedarf besteht, und sie müssen zum anderen nicht aufgelöst werden, wenn sie temporär keine anderen Ressourcen referenzieren, sondern verbleiben im System als Potential zur Befriedigung absehbarer Anforderungen in der Zukunft. Das Modell, das in diesem Kapitel entwickelt wird, erfaßt damit die Unterscheidung zwischen der Generierung, Auflösung und dem Einsatz von Ressourcen. Bei der Entwicklung des Managements ist dies nützlich für die Beurteilung der quantitativen Kosten der Alternativen a) Produktion auf Vorrat und flexible Nutzung einer wiederverwendbaren Ressource oder b) Generierung, einmaliger Einsatz und Auflösung einer Ressource.

Per Definition 5.6 ist es allerdings nicht möglich, daß eine Referenzressource $r$ zu einem Zeitpunkt $t$ keine andere Ressource referenziert. Zur Modellierung „offener Referenzen“, d. h. Referenzressourcen, wurde deshalb die leere Ressource $\varepsilon$ eingeführt. $\varepsilon$ besitzt außer ihrer Existenz und der Eigenschaft, von jeder Referenzressource ungeachtet der Qualifikation referenzierbar zu sein, keine weiteren interessanten Eigenschaften. Je nachdem, ob eine Referenzressource $r$ auf $\varepsilon$ oder eine andere Ressource verweist, heißt die Referenz $r$ *offen* bzw. frei oder *geschlossen* bzw. gebunden. Definition 5.13 führt auf dieser Grundlage die neuen Eigenschaften *gebunden* und *frei* ein.

**Definition 5.13 (Frei und gebunden)**
*Sei $r = \hat{s}$ eine Referenzressource, $\vec{t_n} = (e_1, \ldots, e_n)$ eine n-Tupelressource:*

$$\begin{array}{ll} r \text{ heißt frei} & \Leftrightarrow s = \varepsilon \\ r \text{ heißt gebunden} & \Leftrightarrow s \neq \varepsilon \\ \vec{t_n} \text{ heißt frei} & \Leftrightarrow \exists e_i, 1 \leq i \leq n \wedge e_i \text{ ist frei} \\ \vec{t_n} \text{ heißt gebunden} & \Leftrightarrow \forall e_i, 1 \leq i \leq n \wedge e_i \text{ ist gebunden} \end{array}$$

*Freie und gebundene Referenzressourcen heißen auch **offene** und **geschlossene** Referenzen.*

Die Eigenschaften geschlossen/offen bzw. frei und gebunden sind über die Zeit variable Eigenschaften. Jede Referenzressource sei nach ihrer Erzeugung zunächst offen. Durch Aufruf ihrer äußeren Operation *referenziere* kann die Referenz anschließend auf eine andere Ressource als die leere Ressource gesetzt und damit geschlossen werden. Mit Folgeaufrufen von *referenziere* wird der jeweils aktuelle Bezug, der durch die geschlossene Referenz mit einer anderen Ressource besteht, aufgehoben und ein neuer Bezug zu einer anderen Ressource initiiert. Aufrufe von *referenziere* bewirken somit stets das **Auflösen** des aktuellen und das Festlegen eines neuen Bezugs.

Geschlossene Referenzen induzieren gerichtete Kanten zwischen den Ressourcen eine Systems, die die temporären Nutzungsabhängigkeiten zwischen den Ressourcen des Systems modellieren. Sämtliche Nutzungsabhängigkeiten aller Ressourcen eines Systems zu einem Zeitpunkt $t$ werden mit der dynamischen Bindungsrelation $\rho_t$ registriert.

**Definition 5.14 (Bindungsrelation)**
*Die Bindungsrelation $\rho \subset \mathcal{R} \times \mathcal{R}$ ist wie folgt definiert:*

$$\rho_t(r, s) \Leftrightarrow r, s \in \mathcal{R} \wedge \text{zum Zeitpunkt } t \text{ gilt: } r = \hat{s}$$

*Ferner seien $\bar{\rho}_t(r, s) \Leftrightarrow \rho_t(r, s) \vee \rho_t(s, r)$ die ungerichtete Variante von $\rho_t$ sowie $\rho_t^*, \bar{\rho}_t^*$ die transitiven Hüllen von $\rho_t$ und $\bar{\rho}_t$.*
*Elemente aus $\rho_t$ bzw. $\bar{\rho}_t$ heißen gerichtete/ungerichtete **Bindungen**.*

Die Begriffe „frei“ und „gebunden“ werden oft auch in der umgekehrten Richtung verwendet, um die Frage zu beantworten, ob eine Ressource $r$ von einer anderen Ressource $s$ referenziert wird, d. h. ob $\rho_t(s, r)$ zutrifft. Um Mißverständnisse auszuschließen ist es zweckmäßig, „frei“ und „gebunden“ bei Bedarf näher zu präzisieren. Gilt $\rho_t(s, r)$, so heißt $r$ **von** $s$ und $s$ **an** $r$ gebunden. Eine Ressource $r$, die in einem Kontext $K \subset \mathcal{R}$ zu einem Zeitpunkt $t$ von keiner anderen Ressource $s$ gebunden wird, heißt ungebunden bzw. frei **in** $K$. Ist die Richtung der Bindung nicht von Interesse sondern lediglich die Tatsache, daß es eine Bindung zwischen zwei Ressourcen $a$ und $b$ gibt, so werden $r$ und $s$ **aneinander** oder kurz „gebunden“ genannt, d. h. $\bar{\rho}_t(r, s)$.

Weiter oben wurde erläutert, daß letztendlich Korrespondenzen zwischen den Nutzungsressourcen und den Hardwareressourcen benötigt werden, die über zahlreiche Zwischenressourcen herzustellen sind. Zur Modellierung dieser vielschichtigen Zusammenhänge dienen Pfade. Pfade zwischen Ressourcen beschreiben komplexe, transitive Abhängigkeiten und variieren über die Zeit.

**Definition 5.15 (Pfad)**
*Seien $r, s \in \mathcal{R}$ zwei Ressourcen:*
*Ein gerichteter Pfad von $r$ nach $s$ zum Zeitpunkt $t$ – $\mathcal{W}_t(r, s)$ – ist eine Folge gebundener Ressourcen, der Art $\rho_t(r, a_1), \ldots, \rho_t(a_n, s)$.*
*Ungerichtete Pfade $\bar{\mathcal{W}}_t$ seien äquivalent mit $\bar{\rho}_t$ anstelle von $\rho_t$ definiert.*
*Die Länge eines Pfades sei die Anzahl der Bindungen auf dem Pfad.*

**Satz 5.16**
*Seien $r, s \in \mathcal{R}$ zwei Ressourcen:*
*$\mathcal{W}_t(r, s)$ existiert $\Leftrightarrow \rho_t^*(r, s)$*

**Beweis**
Folgt unmittelbar aus den Definitionen 5.14, 5.15 sowie der Konstruktion der transitiven Hülle von $\rho$. □

Genauso wie INSEL-Komponenten und andere Ressourcen dynamisch erzeugt und aufgelöst werden, werden auch Bindungen zwischen Ressourcen dynamisch initiiert und aufgelöst. Um verbindliche Vereinbarungen über die Nutzungsabhängigkeiten zwischen Ressourcen zu erzielen, ist es zweckmäßig, den Zeitpunkt der Auflösung einer Bindung gemeinsam mit ihrer Initiierung festzulegen. Mit *Verbindlichkeitsintervallen* werden diese Vereinbarungen getroffen, indem die Spanne von der Initiierung bis zur Auflösung einer Bindung festgelegt wird.

**Definition 5.17 (Verbindlichkeitsintervall)**
*Sei $r$ eine Referenzressource, $s \in \mathcal{R}$ , $t_1$ der Zeitpunkt zu dem die Bindung $\rho(r, s)$ initiiert wird und $t_2$ der Zeitpunkt ihrer Auflösung.*

$$\textit{Das Intervall } \Phi(r_{\dot{s}}) = [t_1, t_2] \textit{ ist das Verbindlichkeitsintervall der Bindung } \rho(r, s).$$

*Verbindlichkeitsintervalle heißen synonym auch Bindungsintervalle.*

Die Einhaltung dieser Vereinbarungen ist u. a. für den Bestand der Pfade und damit der Korrespondenzen zwischen $\mathcal{N}$ und $\mathcal{H}$ wesentlich. Die Festlegung geeigneter Verbindlichkeitsintervalle ist darüber hinaus entscheidend für die Flexibilität der Nutzung, Auflösung und Neuproduktion von Ressourcen. Kurze Intervalle bedeuten hohe Flexibilität aber gleichzeitig auch häufigen, kostspieligen Bedarf für die erneute Festlegung von Bindungsintervallen

und Pfaden. Bei langfristig ausgelegten Verbindlichkeitsintervallen reduziert sich die Notwendigkeit erneut Bindungen festzulegen, etc. stark. Gleichzeitig bestehen dann nur selten Möglichkeiten, Veränderungen am System vorzunehmen, die u. U. auf Grund veränderter Anforderungen erforderlich wären. Die Wahl geeigneter Intervalle ist somit ein entscheidender Faktor für das Erzielen einer Balance zwischen flexibler Anpaßbarkeit an variierende Anforderungen auf der einen und Effizienz auf der anderen Seite.

### 5.2.1 Bindungskomplexe

Mit den Konzepten für Ressourcen und Bindungen ist es möglich, Systeme zu konstruieren, die aus zahlreichen Ressourcenklassen, Ressourcen und vielfältigen Abhängigkeiten zwischen diesen bestehen. Die Komplexität solcher Systeme hängt offensichtlich wesentlich von der Anzahl der Ressourcenklassen, Ressourcen, den Abhängigkeiten sowie der Qualifikation der Ressourcenklassen ab. Nur im Falle sehr einfacher Beispielsysteme ist es möglich, für konkret zu treffende Entscheidungen das gesamte System mit sämtlichen Ressourcen und Bindungen zu erfassen. Werden die Systeme komplizierter, wie es bei Betriebssystemen und insbesondere verteilten Betriebssysteme zweifellos der Fall ist, so müssen sinnvolle Ausschnitte aus dem Gesamtsystem gebildet und betrachtet werden.

**Definition 5.18 (Bindungskomplex)**
*Eine Teilmenge* $BK \subseteq \mathcal{R}$ *heißt Bindungskomplex* $\Leftrightarrow \forall r, s \in \mathcal{R} : \exists \bar{\mathcal{W}}_t(r, s)$ *so, daß alle Knoten des Pfades Elemente des Bindungskomplexes* $BK$ *sind.*

Bindungskomplexe sind eine wichtige Möglichkeit, die benötigten Ausschnitte zu bilden. Ein Bindungskomplex besteht gemäß Definition 5.18 aus einer Menge nutzungsabhängiger Komponenten, die ausnahmslos über (ggf. ungerichtete) Bindungspfade verbunden sind und somit voneinander abhängig sind. Die Abhängigkeiten innerer Ressourcen des Bindungskomplexes zu Ressourcen, die selbst nicht Teil des Komplexes sind, definieren die Abhängigkeiten des Komplexes mit dem Rest des Systems. Bei Veränderungen innerhalb des Komplexes müssen zwei Dinge berücksichtigt werden. Erstens müssen mit den Ressourcen, die einen verbindlichen Bezug nach außen aufweisen, die Nutzungseigenschaften des Komplexes nach außen weiterhin gewährleistet bleiben. Zweitens dringen auch Veränderungen innerhalb des Komplexes, selbst wenn sie die erste Forderung erfüllen, unter Umständen nach außen. Dies bedeutet, daß sich black box Verhalten nicht in allen Fällen durchsetzen läßt. Der Grund dafür ist, daß die Bindungen innerhalb des Komplexes Elemente von Pfaden sind, die durch den Komplex verlaufen und als Ganzes qualitative und quantitative Eigenschaften besitzen, die sich aus ihrer Route und Länge ergeben.

Abbildung 5.2 illustriert die Schemata dreier besonders wichtiger Bindungskomplexe. Auf der linken Seite ist eine Referenzressource $r$ dargestellt, die eine andere Ressource $s$ referenziert. Für den Nutzer von $r$ besitzt $r$ im Intervall $\Phi(r_{\hat{s}})$ die Nutzungseigenschaften von $s$. Bei dem Komplex in der Mitte vereint die Tupelressource $t$ mit ihren Elementen die Eigenschaften von $s_1$ und $s_2$. Im dritten Schema (c) selektieren $r_1$ und $r_2$ separate Tupelelemente aus $t$. Die rekursive Definition von Ressourcen in Def. 5.6 legt die Vermutung nahe, daß mit Kompositionskonzepten für Ressourcen, ausgehend von elementaren Ressourcen, durch Tupelproduktionen stets größere Einheiten gebildet werden. Dies allein würde allerdings nicht zur Modellierung eines komplexen V-PK-Systems genügen. Aus Richtung der INSEL-Nutzungskonzepte werden aus elementaren Konzepten, wie Fallunterscheidungen, etc. größere Einheiten, wie Akteursphären, konstruiert. Aus Richtung der Hardware werden ebenfalls aus elementaren

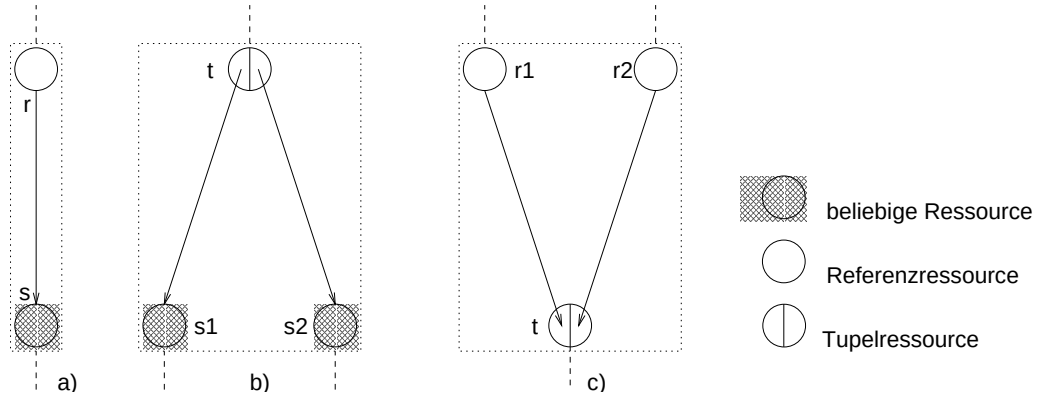


Abbildung 5.2: Identität, Kombination und Projektion

Ressourcen, wie Register und Speicherzellen, komplexere Ressourcen, wie Keller und abstrakte Prozessoren gefertigt. Ungeachtet der Richtung müssen allerdings Pfade zu den elementaren Bausteinen der Gegenseite hergestellt werden. Dies ist offensichtlich nur dann möglich, wenn auf den Pfaden nicht ausschließlich Vergröberungen sondern auch Aufspaltungen stattfinden (s. Abbildung 5.3). Mit den Schemata der Bindungskomplexe aus Abbildung (5.2) sind die dafür erforderlichen Konzepte mit den Charakteristika Identität (a), Kombination (b) und Projektion (c) erklärt.

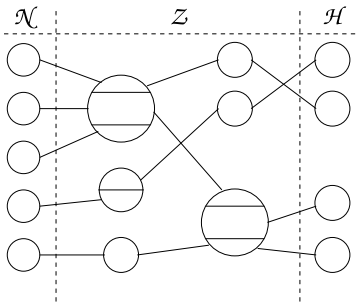


Abbildung 5.3: Beispiel-System

### 5.2.2 Bindungsintensitäten

Die Gestaltung der Pfade erlaubt die Variation der Intensität der Abhängigkeiten zwischen Ressourcen. Mit der Anzahl der Referenzressourcen, die auf einem gerichteten Pfad zwischen zwei Ressourcen $r$ und $s$ liegen, steigt die Anzahl der Verbindlichkeitsintervalle, die über die Gültigkeit des Pfades bestimmen. Vorausgesetzt, daß nicht der triviale Fall zutrifft, daß alle Verbindlichkeitsintervalle auf dem Pfad identisch festgelegt sind, steigt daher mit der Länge des Pfades, die Flexibilität der Bindung. Eine transitive Bindung wird deswegen stärker(enger)/schwächer(loser) genannt, je kürzer/länger der Pfad zwischen den betrachteten Ressourcen ist. Die Abstufung der Intensität von Bindungen ist neben der Festlegung der Verbindlichkeitsintervalle das zentrale Mittel zur Variation der Flexibilität bezüglich der Nutzung von Ressourcen.

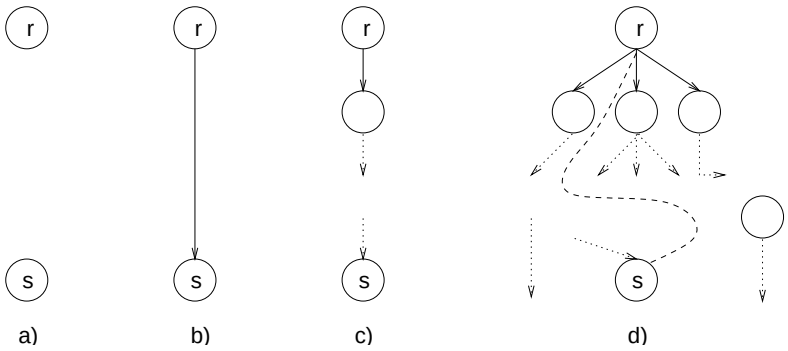


Abbildung 5.4: Abgestufte Bindungsintensitäten

Abbildung 5.4 illustriert vier charakteristische Bindungsintensitäten, die hier qualitativ klassifiziert und später bei der integrierten Instrumentierung des V-PK-Managements wieder aufgegriffen werden. Betrachtet wird in allen vier Fällen der Pfad zwischen den beiden Ressourcen $r$ und $s$. Abbildung a) zeigt den entarteten Fall, in welchem $r$ und $s$ aktuell **nicht** gebunden sind – es existiert kein Pfad. Wird ein Bezug zwischen den beiden Ressourcen benötigt, so muß ein brauchbarer Pfad erst hergestellt werden. Qualitativ *nicht* gebunden schließt hier nicht aus, daß außerhalb des betrachteten Komplexes ein Pfad zwischen $r$ und $s$ existiert. Von diesem wird allerdings angenommen, daß er äußerst schwach ist. Z. B. die im Abschnitt 2.3.3.2 erläuterte Relokationsauflösung operiert auf in diesem Sinne nicht gebundenen Bausteinen von Maschinenprogrammen. In b) sind die $r$ und $s$ **unmittelbar** ohne weitere Zwischenressourcen aneinander gebunden. Dies ist die stärkste Form der Bindung. Bei der Variante c) handelt es sich um eine mittelbare Bindung. Das Kriterium für die Klassifikation der Bindungsintensität als **mittelbar** lautet: auf dem Pfad finden sich wenige Zwischenressourcen. Mittelbare Bindung erweitert die Freiheitsgrade im Umgang mit $r$ und $s$ ohne gravierende Auswirkungen auf die Pfadlänge und damit die Effizienz zu besitzen. Ein typisches Beispiel für mittelbare Bindung aus realen Systemen ist die indirekte Adressierung. Eine deutlich schwächere Intensität findet sich mit der **schwachen** Bindung. Hier verläuft der Pfad über eine ganze Reihe von Zwischenressourcen innerhalb des Bindungskomplexes. Die Anpaßbarkeit des Pfades ist ebenso hoch wie der zu betreibende Aufwand. Ein typisches Beispiel für schwache Bindung sind zum Beispiel tabellengesteuerte Unterprogrammaufrufe in Napier88 (Seite 106 ff.) oder die Anbindung von Serverprozessen im Benutzermodus in Mikrokernarchitekturen.

### 5.2.3 Eigenschaften und Abhängigkeiten

Zwei zentrale Begriffe, die in den vorhergehenden Abschnitten wiederholt verwendet wurden, ohne präzise eingeführt worden zu sein, sind „Eigenschaften“ und „Abhängigkeiten“. Die vollständige und abschließende Klärung dieser Begriffe ist schwierig und wird hier nicht in Anspruch genommen. Auf Grundlage der erklärten Ressourcenkompositions- und Bindungskonzepte soll mit folgenden Aussagen allerdings ein Beitrag zu ihrer Präzisierung geleistet werden.

Eine Eigenschaft ist ein Merkmal eines Elements aus dem Universum eines Systems, das mit dem beschriebenem Konzepterepertoire konstruiert ist. Ressourcenklassen, Ressourcen,

Bindungen, Bindungskomplexe, Pfade, et cetera besitzen Merkmale. Gemäß Definition 5.1 gibt es **invariante** und über die Zeit **variable** Eigenschaften. Ressourcenklassen sind auf Grundlage invarianter Eigenschaften spezifiziert. Instanzen der Klasse erben die Eigenschaften der Klasse und besitzen zusätzlich eigene. Die Eigenschaften der Ressourcen aus der Menge $\mathcal{H}$ sind durch verbindliche **physische** Wirkungen auf die Umwelt des Rechensystems sowie auf andere Ressourcen aus $\mathcal{H}$ gegeben. Die Eigenschaften der Elemente aus $\mathcal{N}$ sind **definitorisch** festgelegt, d. h. durch Bindung der Konzepte an eine abstrakte Semantik. Im Falle von INSEL ist diese Semantik wie auch bei vielen anderen imperativen Programmiersprachen nicht vollständig formalisiert sondern partiell prosaisch angegeben. Bei den Eigenschaften der Ressourcen aus $\mathcal{Z}$ sind zwei Aspekte festzuhalten. Zum einen werden mit den Ressourcengeneratoren der Zwischenressourcen **geforderte** invariante Eigenschaften kreativ festgelegt. Zum anderen müssen diese Forderungen mit der geeigneten Festlegung von Lebenszeit- und Verbindlichkeitsintervallen durchgesetzt werden, da sich die Eigenschaften von Zwischenressourcen faktisch ausschließlich aus der **Komposition** von Eigenschaften referenzierter Ressourcen zuzüglich der Bindungen ergeben. So könnte für eine Klasse $K$ aus $\mathcal{Z}$ die Eigenschaft gefordert werden: $\forall k \in K : k$ wird maximal von $n$ anderen Ressourcen gleichzeitig referenziert. Dabei würde es sich um eine invariante, definitorisch geforderte Eigenschaft der Klasse handeln, die mit den variablen Eigenschaften der anderen Ressourcen und Bindungen des Systems durchgesetzt werden muß. Die variablen Eigenschaften der Zwischenressourcen beschränken sich auf die dynamischen Bindungen von und an Referenzressourcen.
Basierend auf der Kenntnis der Eigenschaftskomposition sowie der Differenzierung vorgegebener Ressourcen aus $\mathcal{V}$ in Rechenaufträge (z. B. Addition, Schleifen, Order) und Speicher bzw. Werte (siehe Abschnitt 3.2), ist es nützlich, folgende zwei Ressourcenklassen zu unterscheiden:

**Daten:** Ein Datum ist ein Speicher (abstrakt oder real), ein Wert oder eine Zwischenressource, die ausschließlich aus Referenzen auf Daten zusammengesetzt ist.

**Rechenvorschrift:** Eine Rechenvorschrift ist ein Rechenauftrag oder eine Zwischenressource, die Rechenaufträge und ggf. Daten als Hilfsmittel für diese referenziert.

Daten sind die Grundlage für die Verarbeitung abstrakter Information durch das Rechensystem. Rechenvorschriften sorgen zusammen mit der Aktivität der Prozessoren für die dynamische Entwicklung des Systems.

Eines der Ziele des vorliegenden Kapitels ist die Verbesserung des Verständnisses für Abhängigkeiten. Mit den Bindungen werden temporäre Nutzungsabhängigkeiten erfaßt. Es sei darauf hingewiesen, daß darüber hinaus zahlreiche weitere Abhängigkeiten existieren, die zu berücksichtigen, aber mit den bis hierhin erläuterten Konzepten noch nicht erfaßt sind. Weitere Abhängigkeiten werden u. a. etabliert durch: die Zusammenfassung in Tupelressourcen, qualifizierte Ressourcenklassen, die Intensität von Pfaden sowie der Existenz von Ressourcen, die als alternative Bindungsziele dienen können. Diese Aspekte werden im folgenden nicht weiter vertieft. Die noch folgenden Untersuchungen beschränken sich bezüglich der Systematisierung des verteilten Ressourcenmanagements, in Anbetracht des konkreten Realisierungszieles eines effizienten V-PK-Managements, auf die ohnehin komplizierten Nutzungsabhängigkeiten.

## 5.3 STK-Systemmodell

Integriertes Vorgehen erfordert ein integriertes Modell zur gesamtheitlichen und präzisen Beschreibung komplexer Systeme inklusive ihrer Komponenten und Abhängigkeiten mit einer homogenen Sprache. Mit den Konzepten der vorangegangenen Abschnitte wurden die konzeptionellen Grundlagen dafür geschaffen.

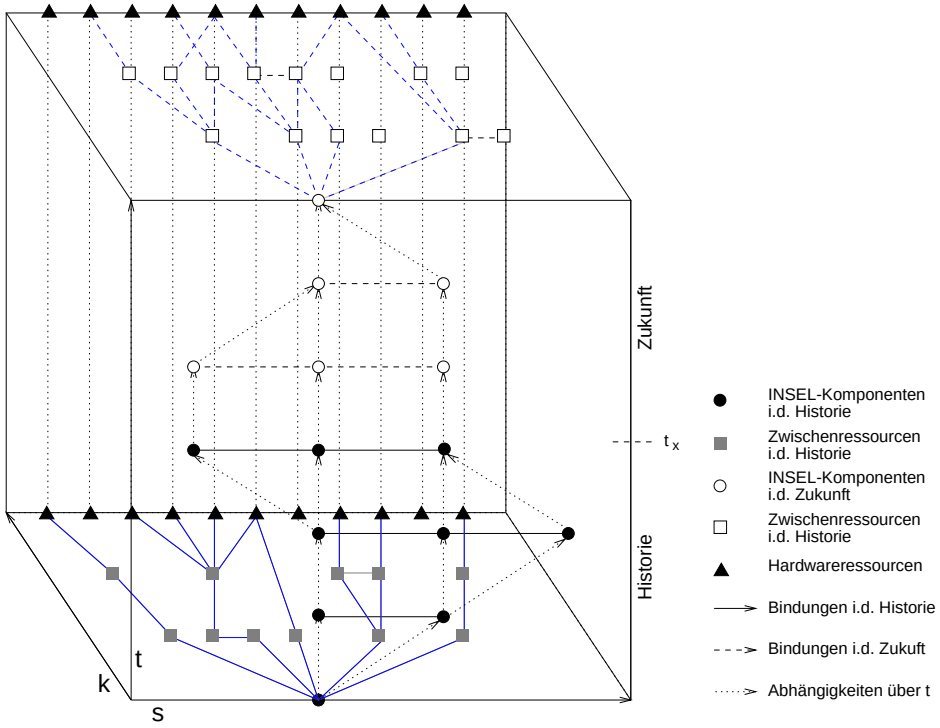


Abbildung 5.5: STK-Systemmodell

Ressourcen und Bindung induzieren einen gerichteten Graph $G_t$, der einen Schnappschuß des Systems $\mathcal{S}_t$ zum Zeitpunkt $t$ darstellt (siehe Abb. 5.3).

$$\text{Systemgraph } G_t = (\mathcal{R}_t, \rho_t)$$

Die Modellierung des Systems mit Hilfe des Systemgraphen $G_t$ gibt die Abhängigkeiten zwischen den Ressourcen eines Systems bereits wesentlich differenzierter wieder, als dies bei den üblichen groben Modellen, wie z. B. einer Schichtenarchitektur der Fall wäre. Was dieser Vorstellung allerdings nach wie vor fehlt, ist die Berücksichtigung des Verhaltens von Ressourcen und Bindungen über die Zeit. Das primäre Interesse an Systemen richtet sich auf die Fähigkeit Berechnungen durchzuführen. Die Durchführung von Berechnungen erfolgt mit hoher Dynamik über die Zeit. Entscheidungen bezüglich der Erzeugung/Auflösung von Ressourcen und Bindungen korrespondieren mit dieser Dynamik. Bei der Betrachtung von Systemgraphen, die Schnappschüsse repräsentieren, werden diese wichtigen Zusammenhänge ignoriert. Für die Entwicklung eines integrierten, leistungsfähigen V-PK-Managements ist das nicht akzeptabel, da abermals wichtige Information über das System sowie Entscheidungskriterien der systematischen Betrachtung entzogen werden. Die Betrachtung statischer Systeme wird hier

deshalb durch die Betrachtung hoch dynamischer und evolutionsfähiger Geflechte ersetzt. Die erste Maßnahme in diese Richtung war die Einführung der Intervalle $\Lambda$ und $\Phi$, die Graphen $G_t$ und $G_{t'}, t' > t$ mittels wohldefinierter Abhängigkeiten verbinden. Das STK-Systemmodell verfügt damit neben den in gängigen Architekturmodellen üblichen Dimensionen **Raum** (S) und **Konkretisierungsniveau** (K) zusätzlich über die Dimension **Zeit** (T).

Abbildung 5.5 stellt das komplettierte STK-Modell eines INSEL-Rechensystems dar, das nun aus drei anstatt der gewohnten zwei Dimensionen besteht. Ohne Beschränkung der Allgemeinheit ist in der Darstellung die Menge der Nutzungsressourcen $\mathcal{N}$ ausgelassen. Stattdessen wurde auf der $k$-Achse bereits ein Schritt Richtung $\mathcal{H}$ zu den INSEL-Komponenten vollzogen. An die Stelle der Nutzungsressourcen tritt die Menge $\mathcal{I}$ der INSEL-Komponenten. Diese Sichtweise ist gerechtfertigt und wird im folgenden beibehalten, da ausschließlich INSEL-Systeme betrachtet werden und sich damit keine interessanten Veränderungen bezüglich der Menge $\mathcal{N}$ ergeben.

Zur Diskussion von Ausschnitten aus dem STK-Modell wird folgende Notation vereinbart. Seien $I_s, I_t, I_k$ Intervalle bezüglich der Achsen $s, t$ und $k$,

$$STK(I_s, I_t, I_k) \text{ bezeichnet den Ausschnitt gemäß } I_s, I_t, I_k.$$

Auf die präzise Angabe der Intervalle wird zur Vereinfachung so weit wie möglich verzichtet. Das Symbol „$\cdot$“ bezeichne das Intervall, das die jeweilige Dimension vollständig abdeckt. Mit $STK(\cdot, I_t, \mathcal{H})$ ist z. B. der Ausschnitt bezeichnet, der sämtliche Ressourcen der Hardware im Zeitintervall $t$ beinhaltet.

### 5.3.1 Dimension t – Zeit

Die Dimension $t$ des STK-Modells beschreibt die Entwicklung des Systems über die Zeit. Die Zeit sei in Anlehnung an [Rad95] als physische Zeit festgelegt, die eine Linearisierung der für das INSEL-System relevanten Ereignisse festlegt, die mit der abstrakten Halbordnung über der Menge dieser Ereignisse konsistent ist. Die Halbordnung ergibt sich aus den Gesetzmäßigkeiten, die mit den Ressourcenklassen, u. a. mit dem Akteur- oder Order-Konzept auf INSEL-Niveau, definiert sind und erlaubt Aussagen der Art $a$ tritt *vor* $b$ ein, $a$ tritt *nach* $b$ ein und $a$ ist nebenläufig zu $b$ (d. h. nicht enthalten in der Halbordnung). Die integrierte Betrachtung der abstrakten Halbordnung und der physischen Zeit reflektiert den Übergang von abstrakt nach konkret, der von dem STK-Systemmodell in der Dimension $k$ (s. u.) erfaßt ist. Für den Rest der Arbeit beziehen sich Aussagen über Nebenläufigkeit und Parallelität auf die abstrakte Halbordnung. Mit „gleichzeitig“ seien Ereignisse bezeichnet, die in ein gemeinsames Intervall bezüglich der physischen Zeit fallen.

Der aktuelle Betrachtungszeitpunkt $t_x$ auf der Achse $t$ gliedert das Modell des Systems in **Historie** ($\forall t_h \in t : t_h < t_x$) und **Zukunft** ($\forall t_z \in t : t_z > t_x$). Die zukünftige Entwicklung des Systems ist u. a. aufgrund der $\Phi$ und $\Lambda$ Festlegungen abhängig von der Historie. Wissen über die Historie ermöglicht deshalb Aussagen und Entscheidungen, die die Zukunft des Systems betreffen. In Managementsystemen wird dieser Zusammenhang auf unterschiedliche Weise genutzt. Ein einfaches Beispiel ist die verbindliche Belegung von Kellerspeicher für eine Zeitspanne. Weitergehende Zusammenhänge wurden mit der adaptiven Replikation von Depots im Experimentalsystem AdaM gezeigt

Ein Schnitt durch das System entlang der Zeit und dem Raum zeigt die Erzeugung und Auflösung von Ressourcen eines Konkretisierungsniveaus sowie die Initiierung und Auflösung

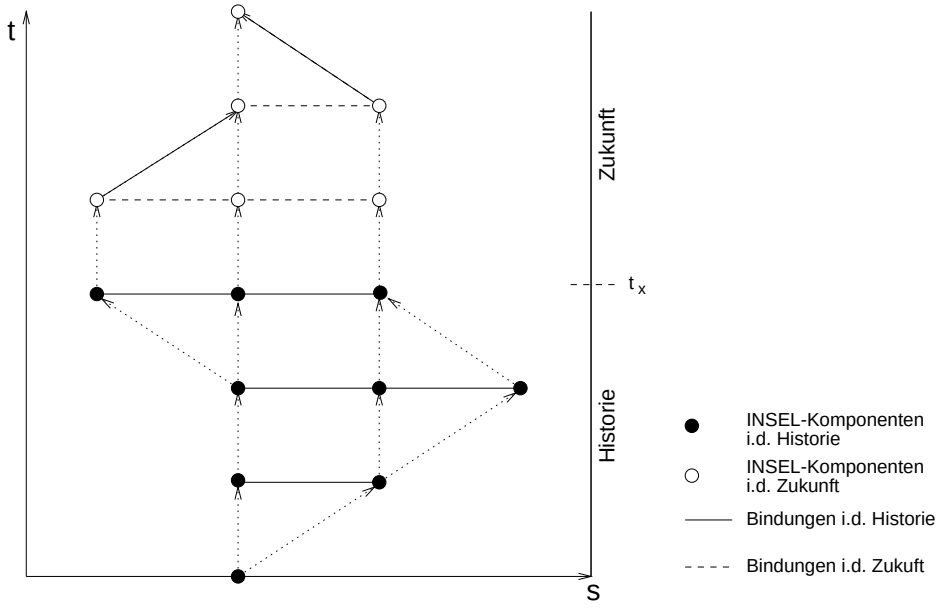


Abbildung 5.6: Raum-/Zeitebene aus dem STK-Systemmodell

von Bindungen zwischen diesen. In Abbildung 5.6 ist ein solcher Ausschnitt $STK(\cdot,\cdot,\mathcal{I})$ dargestellt, für den die strengen Gesetzmäßigkeiten der INSEL-Konzepte gelten. Sämtliche Ressourcen sind INSEL-Komponenten und Teil eines Berechnungsverbandes. Die Strukturrelationen aus 3.2.2 spiegeln sich ausnahmslos in Bindungen in der Horizontalen wider. Die Bezüge entlang $t$ entsprechen den $\Lambda$-Festlegungen zuzüglich der Erzeugungs- und Auflösungsphasen. Verbindlichkeitsintervalle $\Phi$ sind an den Parallelverschiebungen von Bindungen entlang $t$ zu erkennen.

Ein Schnitt entlang $t$ und $k$ durch das Modell würde mit $STK(I_s,\cdot,\cdot)$ in erster Linie die dynamische Erzeugung und Auflösung von Zwischenressourcen sowie die Dynamik der Pfade und Teilpfade zwischen dynamisch erzeugten und aufgelösten $i \in \mathcal{I}$ und einer dauerhaft existierenden Ressource $h \in \mathcal{H}$ zeigen.

### 5.3.2 Dimension k – Konkretisierungsniveau

Die Dimension $k$ des STK-Systemmodells beschreibt den kontinuierlichen Übergang von dem hohen Abstraktionsniveau der INSEL-Komponenten aus der Menge $\mathcal{I}$ zu den konkreten Ressourcen der physischen Geräte aus $\mathcal{H}$. $k$ ersetzt die Ebenen, die in klassischen Schichtenansätzen eingezogen werden. Der Übergang zu einem nahezu kontinuierlichen Spektrum bezüglich $k$ im Gegensatz zu wenigen, rigiden Abstraktionsebenen hat große Bedeutung für die Systemkonstruktion, die sich nicht zuletzt auch auf das Leistungsvermögen des Managements auswirken.

Die Tragweite dieser modifizierten Betrachtungsweise von Systemen wird deutlich, wenn man die übliche Methodik der Systemkonstruktion rekapituliert und mit den Rahmenkonzepten aus diesem Kapitel gegenüberstellt (siehe Abbildung 5.7). Während im STK-Systemmodell $k$ quasi nachträglich zur Strukturierung des dynamisierten Systemgraphen und seiner Darstellung eingesetzt wird, ist die übliche Methodik der Systemkonstruktion genau

umgekehrt. Mit dem Ziel, Pfade zwischen $\mathcal{I}$ und $\mathcal{H}$ zu fertigen, werden im ersten Schritt Ebenen zwischen $\mathcal{I}$ und $\mathcal{H}$ festgelegt. An diesen Abstraktionsebenen orientiert sich anschließend die Festlegung von Ressourcenklassen innerhalb der Ebenen, Ressourcen und so fort. Dabei wird übersehen, daß mit der Festlegung von Ebenen uniforme Entscheidungen getroffen werden, die nicht die Anforderungen reflektieren. Unter anderem wird die Länge aller Pfade durch die Anzahl der zu überwindenden Ebenen a priori vorbestimmt und eine angemessene Differenzierung der Quell-Ziel Transformationen erheblich beeinträchtigt. Oft wird nachträglich versucht, dieses Defizit durch „Optimierung“ – Verkürzung einiger Pfade ungeachtet der Ebenen – abzuschwächen. Ohne diese Nachteile weiter zu vertiefen, bleibt festzustellen, daß die entstehenden Systemgraphen durch das Denken in Abstraktionsebenen partitioniert sind. Dies schlägt sich folgerichtig in der Darstellung von Systemen mit den üblichen Dimensionen Raum und Abstraktionsebenen nieder. Letztere Dimension ist diskret und besitzt in aller Regel nur wenige grobe Abstufungen. Die Dimension Zeit fehlt völlig, obwohl alle Entscheidungen über das System von dem Verhalten über die Zeit abhängen.

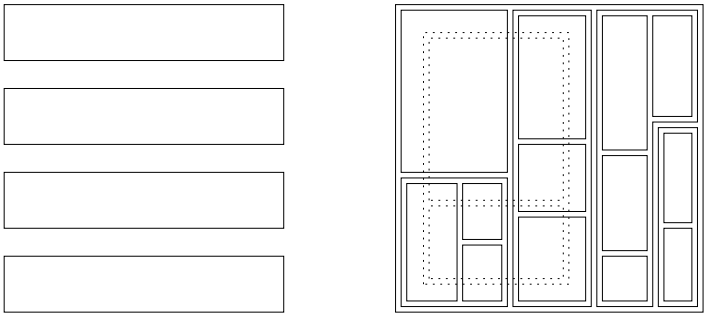

Abbildung 5.7: Abstraktionsebenen und geschachtelte Abstraktionenräume

Die Sicht auf Systeme, die in diesem Kapitel mit den Konzepten für Ressourcenklassen, Ressourcen und Bindungen erarbeitet wurde, ist grundlegend anders. Ressourcenklassen werden als geschachtelte Sub- und Superklassen frei von dem Zwang, horizontale Schnitte durch das gesamte System vornehmen zu müssen, entwickelt. Durch Schachtelung von Klassen, die abgesehen von Sub- und Superklassen paarweise disjunkte Mengen sind, werden rekursive **Abstraktionenräume** gebildet. Jeder Abstraktionenraum ist entweder in eine äußere Schachtel eingebettet, die die Eigenschaften des gesamten Systems in sich vereint oder ist selbst der Abstraktionenraum, der die Ressourcen des gesamten System umfaßt. Abstraktionenräume werden frei in dem durch $s, t$ und $k$ aufgespannten Raum mit der Granularität festgelegt, die den Anforderungen an die Pfade zwischen $\mathcal{I}$ und $\mathcal{H}$ entspricht. Ferner sind in einem System verschiedene überlappende und nicht identische, geschachtelte Abstraktionenräume möglich. Dies ist sinnvoll, um völlig unterschiedliche alternative Pfade in einem System zu modellieren, und entspricht darüber hinaus besser der tatsächlichen Vorgehensweise bei der Systemkonstruktion. So definiert die Abstraktion „Speicher“ einen Abstraktionenraum, in dem virtueller Speicher, Arbeitsspeicher, Register und weiteres geschachtelt sind. Daneben gibt es den Abstraktionenraum Stellenressourcen, mit Prozessoren, Stellenspeicher, et cetera. Offensichtlich sind die beiden Räume weder identisch noch disjunkt (siehe gepunktete Linie in Abbildung 5.7). Die Möglichkeit, überlappende Abstraktionenräume festzulegen und diese zu wechseln, ermöglicht es, die gewünschten kurzen und effizienten Pfade von der Quelle $\mathcal{I}$ zum Ziel $\mathcal{H}$ zu finden. Das Wechseln von Abstraktionenräumen mit benannten Abstraktionen

entspricht dem Wechseln zwischen Sprachen mit den damit einhergehenden Problemen, insbesondere Synonyme und Homonyme. Die Mißverständnisse, die durch die mangelnde Kenntnis von Abstraktionenräumen und dem Bewußtsein für das Wechseln zwischen diesen entstehen, sind allenthalben bei der Diskussion von Entscheidungen bezüglich Systemarchitekturen sowie eventueller Vor- und Nachteile deutlich zu beobachten.

Die eingeführten Rahmenkonzepte und das STK-Systemmodell ermöglichen gegenüber dem Ebenen-Ansatz, diese Vielfalt und Flexibilität im erforderlichen Umfang systematisch zu konstruieren, zu erfassen und weiter zu verarbeiten. Das Konkretisierungsniveau als Dimension $k$ des STK-Modells definiert somit ein fein abgestuftes Raster zur Beurteilung der relativen Nähe/Entfernung von Ressourcen zu $\mathcal{I}$ bzw. $\mathcal{H}$. Die einzigen beiden Schnitte, die ausgehend von einem Punkt auf der $k$-Achse über den gesamten Raum $s$ gezogen werden können, sind auf dem Niveau der invarianten elementaren Ressourcen $\mathcal{N}^3$ und $\mathcal{H}$.

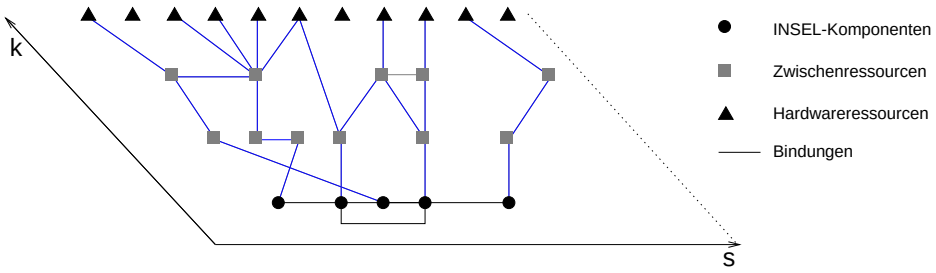


Abbildung 5.8: Ebene Raum-/Konkretisierungsniveau aus dem STK-Systemmodell

Eine von $s$ und $r$ zu einem Zeitpunkt $t$ aufgespannte Ebene zeigt, wie in Abbildung 5.8, die INSEL-Komponenten, Zwischenressourcen, Hardwareressourcen und Bindungen zwischen diesen, die den Systemgraphen $G_t(\mathcal{R}, \rho_t)$ zum Zeitpunkt $t$ bilden.

### 5.3.3 Dimension s – Raum

Die verbleibende Dimension Raum beschreibt die Komponenten und Bindungen, die sich gemäß $k$ zu einem Zeitpunkt $t$ auf dem selben Konkretisierungsniveau befinden. Aus den obigen Aussagen über $t$ und $k$ folgt, daß Intervalle aus $t$ und $k$ herangezogen werden müssen, um Ressourcenmengen entlang der $s$-Achse bilden zu können. In dem Ausschnitt $STK(\cdot, t, \mathcal{I})$, der dem Konkretisierungsniveau der INSEL-Komponenten entspricht, finden sich entlang $s$ die INSEL-Komponenten und strukturelle Abhängigkeiten zwischen diesen, die zum Zeitpunkt $t$ existieren. Auf dem Konkretisierungsniveau $\mathcal{H}$ der Hardwareangebote, findet sich in der Dimension $s$ zu jedem Zeitpunkt die Menge der Hardwareressourcen $\mathcal{H}$, die über die Zeit $t$ im Gegensatz zu $\mathcal{I}$ und $\mathcal{Z}$ nicht variiert. Zu den Elementen dieser Menge gehören auch die Netzwerkressourcen der Stellen, wodurch die physische Verteilung im STK-Systemmodell repräsentiert ist.

[3] für $\mathcal{I}$ gilt dies nicht

## 5.4 Zusammenfassung und Bemerkungen

Mit dem STK-Systemmodell, das in diesem Kapitel auf der Grundlage einer präzisen Definition der Konzepte, Ressourcen und Bindungen entwickelt wurde, werden die vielfältigen Komponenten und Abhängigkeiten innerhalb eines Systems verdeutlicht und die Integration von Entscheidungen über die Dimensionen Raum, Zeit und Konkretisierungsniveau ermöglicht. Ferner wurde ein Ansatz zur Systematisierung der Konstruktion komplexer Systeme erarbeitet, der sich nicht mehr an konkreten Begebenheiten seitens der Hardware oder groben Schichtungen orientiert, sondern die Verklammerung von Anforderungen und Angeboten durch flexible, dynamische Pfade in den Vordergrund stellt. Dieses Modell wird bei der noch folgenden Konkretisierung der MoDiS-Managementarchitektur als Anschauungsobjekt herangezogen, um die Argumentation pro und kontra Realisierungsalternativen zu unterstützen.

Offensichtlich wirft das STK-Systemmodell selbst einige neue Fragen auf, die im Rahmen der vorliegenden Arbeit nicht weiter vertieft werden können. Ein Beispiel sind die Konsequenzen des Überganges von dem Schichtenmodell zu geschachtelten Abstraktionenräumen in Bezug auf die Austauschbarkeit und Wiederverwendbarkeit von Komponenten. Eine andere offene Frage betrifft die Vollständigkeit des Modells. Diese und weitere Fragen sind wichtig und müssen langfristig im Zuge der weiteren Systematisierung der Systemkonstruktion selbstverständlich beantwortet werden. Für die Zielsetzung der systematischen Konstruktion eines integrierten, leistungsfähigen V-PK-Managements genügen allerdings die bis hierhin erarbeiteten Konzepte und Sichten.

**Sprache**
Mit den Beobachtungen aus diesem Kapitel ist es interessant, am Rande die Bedeutung von Sprache für die Konstruktion von Systemen zu diskutieren. Zur Beschreibung von Systemen mit Ressourcen und Abhängigkeiten wird Sprache benötigt. Die Grammatik $G(N, T, P, S)$ einer möglichen Sprache für einen vollständigen Abstraktionenraum, dessen äußere Schachtel das gesamte System beinhaltet, könnte stark vereinfacht etwa wie folgt aussehen:

- $T = \mathcal{H} \cup \mathcal{N}$, $N = \mathcal{RK}$
- $S =$ die Ressourcenklasse $rk \in \mathcal{RK}$, die der äußersten Schachtel des Abstraktionenraumes entspricht.
- $P$ wird über die Menge der Ressourcenklassen konstruiert:
  - Zu Beginn gilt $P = \emptyset$
  - Sei $\Sigma_{rkr}$ die Signatur der mit $rks$ qualifizierten Referenzressourcenklasse:
    $P = P \cup rkr \rightarrow rks$
  - Sei $\Sigma_{rkt}$ die Signatur der $n$-Tupelklasse $(rkr_1, \ldots, rkr_n)$:
    $P = P \cup rkt \rightarrow rkr_1 rkr_2 \ldots rkr_n$

Offensichtlich wird die Grammatik auf Grund der vielen verschiedenen Ressourcenklassen und Bezüge sehr umfangreich und verfügt über eine enorme Menge an Symbolen und Produktionen. Ferner müssen nicht nur genau eine, sondern mehrere Abstraktionenräume und damit die kontextsensitive Konjunktion von Sprachen beherrscht werden. Sinnvoll wäre die Generierung der Namen der Grammatiksymbole nach einem festen Schema. Abstrakt ist das keine Schwierigkeit, in der Praxis der Betriebssystemkonstruktion durchaus. Es ist zu vermuten, daß die

Kapazität eines menschlichen Konstrukteurs nicht groß genug ist, um mit einem Wortschatz von diesem Umfang agieren zu können. Rechnergestützte Verfahren existieren ebenfalls bislang nicht. In der Praxis resultieren daraus zwangsläufig Mißverständnisse und Unkenntnis, die zu Entwurfsfehlern und einer starken Aberration von der Zielsetzung führen. Ein typisches Beispiel für dieses Problem wurde mit der Parameterübergabe bereits in Abschnitt 2.2.2 erwähnt. Die Limitation der natürlichen Sprache ohne systematische und maschinelle Unterstützung fördert diese Schwierigkeiten.

# Kapitel 6

# Inkrementelle Systemkonstruktion

Mit dem Ein-Programm-Ansatz, der im Projekt MoDiS verfolgt wird, erfolgt eine Integration des Systems über die Dimension *s*. Das Programm beschreibt gesamtheitlich das in Ausführung befindliche V-PK-System, das aus INSEL-Komponenten zusammengesetzt ist und gemäß den INSEL-Strukturrelationen über wohldefinierte globale Abhängigkeiten verfügt. Auf diese Weise liegt Information über Zusammenhänge vor, die für erfolgreiches verteiltes Management, z. B. Plazierung von Akteuren und Replikationsentscheidungen, von größter Bedeutung ist. Darüber hinaus ist sie die Grundlage zur Behebung qualitativer Defizite, wie redundante Bibliotheken oder mangelnde Integrität wechselseitiger Referenzen zwischen Benutzerprogrammen.

Nun sollen mit dem Ein-Programm-Ansatz keine kurzlebigen, abgeschlossenen Anwendungswelten geschaffen werden, sondern ein langlebiges System, in das Problemlösungen von Benutzern als Erweiterungen dynamisch eingebracht, ausgeführt und später ggf. wieder entfernt werden können. Herkömmliche Systeme ermöglichen dies mechanistisch mit Hilfe von Dateien und der schmalen Kerndienst-Schnittstelle, über die Benutzerprogramme in das System eingebracht aber nur sehr schwach mit den bereits bestehenden Komponenten gebunden werden können. Da dies nicht von Maßnahmen auf Sprachniveau begleitet wird, sind die Bezüge zwischen Programmen im allgemeinen weder spezifiziert noch dem System bekannt. In Folge kann das System Zusammenhänge nicht automatisiert gewährleisten oder zur Steigerung der Leistung des Managements nutzen. Ferner ist die Weiterentwicklung des Systems auf das Hinzufügen grobgranularer Benutzerprogramme beschränkt. Anpassungen an den bereits bestehenden und in Ausführung befindlichen Komponenten sind nicht möglich. Mit den Konzepten, die in diesem Kapitel erarbeitet werden, wird wesentlich feiner granulare Adaptions- und Evolutionsfähigkeit des Systems und seines Programmes auf Sprachniveau verankert. Die Gesetzmäßigkeiten der INSEL-Konzepte gelten damit über die Grenzen einzelner Problemlösungen hinweg.

Das System wird somit einerseits so betrachtet, als ob mit dem Programm vollständiges Wissen über das System und sein Entwicklungspotential vorhanden wäre. Tatsächlich ist die zukünftige Entwicklung des Systems zu einem gewissen Grad jedoch unbekannt, da keine Kenntnis über zukünftige Veränderungen der Anforderungen, wie im speziellen Fall von Problemlösungen von Benutzern, vorliegt. In dem bisher als vollständig betrachteten Programm müssen deshalb Freiheitsgrade vorgesehen werden, die einen definierten Rahmen für die Weiterentwicklung des Programms über die Zeit festlegen. Der Schlüssel zu dieser Vorgehensweise ist die Systematisierung und Flexibilisierung der **Unvollständigkeit** von Programmen,

die, wie unten argumentiert wird, in einem engen Rahmen ohnehin stets praktiziert wird. Das unvollständige Programm wird schrittweise erweitert, wodurch der Konstruktionsprozeß insgesamt einen inkrementellen Charakter erhält und gemeinsam mit Teilausführungen des Programmes über die Dimension Zeit flexibilisiert wird.

In den folgenden Abschnitten werden zunächst einige generelle Aspekte der unvollständigen Spezifikation erweiterbarer Programme diskutiert. Anschließend werden die konzeptionellen Grundlagen erarbeitet, die benötigt werden, um das verteilt in Ausführung befindliche INSEL-Programm gemäß der in 1.1.1 formulierten Zielsetzung inkrementell erweitern zu können. Die Darstellungen in diesem Kapitel konzentrieren sich dabei auf das Niveau $STK(\cdot,\cdot,\mathcal{I})$ der INSEL-Komponenten, die Zwischenressourcen im Sinne der Rahmenkonzepte aus dem vorgehenden Kapitel sind. Die Konsequenzen, die sich daraus auf realisierungsnähere Konkretisierungsniveaus ergeben, wie z. B. Lockerung der Bindung zu den Realisierungen der INSEL-Komponenten sowie Auswirkungen auf den Übersetzer und den Binder, werden später im Rahmen der Realisierung des V-PK-Managements weiter präzisiert. Wegen des hohen konzeptionellen Abstraktionsniveaus kommt ferner die physische Verteilung der Ausführungsumgebung in diesem Kapitel nicht zum Tragen. [1]

## 6.1 Flexibel wiederverwendbare Schablonen

Die Konzepte der unvollständigen Spezifikation und inkrementellen Vervollständigung des INSEL-Systems sind in der Kombination Ein-Programm-Ansatz und Systemdynamik neu. Insgesamt ordnet sich dieses Thema allerdings in den ganz allgemeinen Problemkomplex der Wiederverwendung bzw. Weiterverwendung nach Anpassung ein, der für die Informatik von zentraler Bedeutung ist. In diesem Abschnitt werden zur Hinführung auf die Erweiterbarkeit von INSEL-Systemen einige Parallelen zu ähnlichen Konzepten und Verfahren aufgezeigt, mit deren Hilfe die generelle Tragweite der inkrementellen Systemkonstruktion illustriert und die Flexibilisierung der Entwicklung von Programmen vorbereitet wird.

Im allgemeinen wird mit Programmen angestrebt, einmal gelöste Probleme nicht erneut lösen zu müssen, sondern auf bereits formulierte und operationalisierte Verfahren zurückgreifen zu können. Da Probleme in der Realität in Variationen auftreten, werden bekannte Problemlösungen durch die Integration von Freiheitsgraden flexibilisiert, um sie im Bedarfsfall in ihren Grundzügen beibehalten und im Detail schnell an den aktuellen Nutzungskontext anpassen zu können. Der Prozeß der Flexibilisierung einer Problemlösung durch Hinzunahme von Freiheitsgraden entspricht seiner Verallgemeinerung durch Abstraktion von Details.

> Eine Verallgemeinerung einer Problemlösung $P$ ist die Lösung von Bindungen in $P$, d. h. das Ersetzen gebundener durch freier Variablen. Die resultierende unvollständigere Problemlösung $P^u$ ist eine Abstraktion von $P$ mit zusätzlichen Freiheitsgraden.

Durch Abstraktion entstehen somit abstrakte Schablonen von Problemlösungsverfahren, die im Bedarfsfall mit dem Wissen über den konkreten Nutzungskontext wieder vervollständigt werden.

[1] Es sei bemerkt, daß davon auszugehen ist, daß sinnvolle Programme häufig gelesen und selten geschrieben werden und damit für Replikation prädestiniert ist. In einem Ein-Adreßraum gibt es neben der Replikation wenige interessante Fragestellung bzgl. der Verteilung des Programmcodes, weshalb dieses Thema in der vorliegenden Arbeit nicht behandelt wird.

> Eine Vervollständigung einer Problemlösung $P$ ist das Binden freier Bezüge von $P$, wodurch Freiheitsgrade aus $P$ entfernt werden und eine konkretere Problemlösung $P^v$ entsteht.

Schrittweises Vervollständigen entspricht somit schrittweiser Konkretisierung, mit dem Ziel, Freiheitsgrade sukzessive zu eliminieren, um eine vollständig festgelegte und ausführbare Problemlösung zu produzieren. Abbildung 6.1 stellt exemplarisch die Evolution eines Pro-

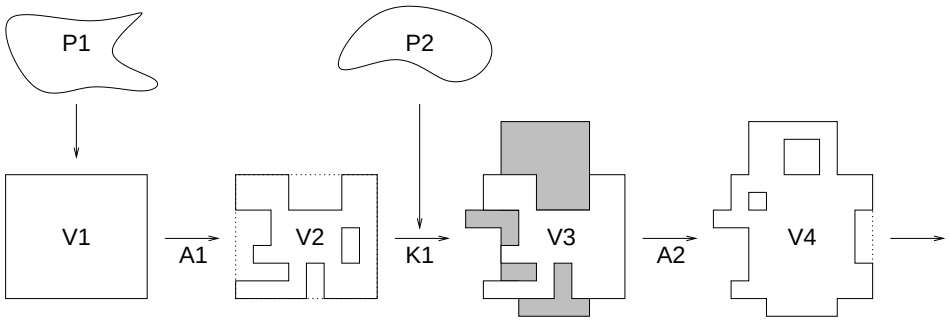


Abbildung 6.1: Evolution eines Problemlösungsverfahrens

blemlösungsverfahrens dar. Für das konkrete Problem $P_1$ wird das Lösungsverfahren $V_1$ entwickelt, das anschließend im Schritt $A_1$ durch Reduktion der Menge der seiner Komponenten, zu dem Verfahren $V_2$ verallgemeinert wird. Im Schritt $K_1$ wird das allgemeine Verfahren $V_2$ im Hinblick auf die Lösung des Problems $P_2$ konkretisiert. Das Ergebnis der vervollständigenden Erweiterung ist $V_3$, das nach seiner möglichen Flexibilisierung nicht zwangsweise $V_2$ entsprechen muß und neben oder statt diesem weiter entwickelt und eingesetzt werden kann.

Die Spezifikation der Freiheitsgrade mittels freier Bezüge sowie die Festlegung gewünschter Bindungen im Zuge der Vervollständigung erfolgt mittels formaler Sprachen. In der Praxis ist das skizzierte Prinzip der Verallgemeinerung und Konkretisierung in zahlreichen Ausprägungen zu beobachten, wobei „formale Sprache“ nicht beschränkt auf Programmiersprachen zu verstehen ist. Beispiele sind unter anderem, sortiert nach der Tragweite:

- Eingabe von Daten während der Laufzeit
- Textsubstitution durch Präprozessoren
- Parametrisierte Unterprogramme
- Das Schachtelungsprinzip
- Vererbung in objektorientierten Sprachen
- Design Patterns und Modellierungssprachen wie UML [Rat97]

Unvollständige Spezifikation und Vervollständigung von Problemlösungen ist somit ein integraler Bestandteil von Bemühungen, wieder- und weiterverwendbare Programme, Komponenten und Systeme zu konstruieren. Die eigentlichen Probleme und Ziele der inkrementellen Erweiterung von Problemlösungen sind über die Zeit allerdings in den Hintergrund getreten. Dies zeigt sich daran, daß zwar Details der verschiedenen Ansätze verbessert werden, wie z. B. ein zusätzliches Feature bei der Makrosubstitution, die tatsächlichen Charakteristika

der Problemstellung jedoch kaum systematisch aufgearbeitet werden. So leiden alle der oben genannten Ansätze an einer stark limitierten Ausdrucksfähigkeit für Freiheitsgrade. Ferner bleibt die Einordnung der Zeit in dem Prozeß der Konkretisierung und Abstraktion außen vor. Dies zeigt sich insbesondere daran, daß kaum ein Zusammenhang zwischen statischen und dynamischen Methoden hergestellt wird. Von den oben angeführten Ansätzen dienen einzig Eingabe sowie parametrisierte Unterprogramme dem Zweck der Vervollständigung zur Ausführungszeit. Alle anderen Vorgehensweisen beziehen sich auf Maßnahmen, die lediglich entkoppelt von der Ausführung durchführbar sind.

Die schwache Einordnung der Dynamik des Systems hat zur Folge, daß in der Praxis auf Veränderungen der Anforderungen, die über die Eingabe und Parametrisierung hinausgehen, nur reagiert werden kann, indem Zustände gesichert und das laufende System angehalten wird. Anschließend erfolgen statische Rekonfigurationen wonach das System neu gestartet und der alte Zustand wieder hergestellt werden muß. Diese Vorgehensweise ist insgesamt unbefriedigend und für Systeme deren ständige Verfügbarkeit von vitaler Bedeutung ist, wie z.B. Produktions-Datenbanken von Großunternehmen, ebenso unakzeptabel wie in dem verteilten Gesamtsystem-Ansatz von MoDiS. Die Ursache hierfür liegt in der rigiden Separation von Konstruktion und Ausführung, die sich im Laufe der Jahre durch die limitierten Fähigkeiten gängiger Realisierungstechniken, insbesondere Übersetzer und Binder, entwickelt hat. Interpretative Techniken stellen dabei eine Ausnahme dar, die wegen ihrer schwachen Effizienz allerdings nur geringe Relevanz besitzen, wenngleich in diesem Fall Veränderungen des Programmes in einem sehr flexiblen Rahmen ohne Unterbrechung seiner Ausführung möglich sind, siehe Smalltalk [GR85]. Allerdings ist auch in diesen Umgebungen inkrementelle Konstruktion nicht konzeptionell verankert, sondern tritt eher als Phänomen auf.

Mit der inkrementellen Erweiterbarkeit, die in diesem Kapitel für INSEL-Systeme erarbeitet wird, werden Freiheitsgrade zur Planung des Entwicklungspotentials des gesamtheitlichen V-PK-Systems als elementare Bestandteile seines Programmes und in einen dynamischen Konstruktionsprozeß auf konzeptioneller anstelle mechanistischer Ebene verankert. Dies erfolgt losgelöst von der Frage, ob die Realisierung interpretativ oder transformatorisch erfolgt. Fragen dieser Art werden im Zuge der Realisierung des Managements beantwortet. Gemäß den drei Dimensionen der Erweiterbarkeit aus Kapitel 2 Seite 69, wird angestrebt, Erweiterbarkeit in der Dimension eins – Zeit – dynamisch inkrementell zu ermöglichen. In der Dimension zwei – Abstraktionsniveau – (bzw. $k$ im STK-Systemmodell) konzentrieren sich die Überlegungen hier auf das INSEL-Niveau. Offenkundig zieht dies entsprechend große Flexibilität auf niedrigeren Abstraktionsniveaus in zweierlei Hinsicht nach sich. Erstens müssen Veränderungen des Ausschnittes $STK(\cdot,\cdot,\mathcal{I})$ durch entsprechende Veränderungen der Ressourcenmengen und Bindungen auf konkreteren Niveaus durchgesetzt werden. Zweitens ist die Realisierung des Managements selbst Teil des Gesamtsystems und kann daher mit den hier erarbeiteten Konzepten selbst inkrementell konstruiert und flexibel an veränderliche Anforderungen angepaßt werden. Ein mögliches Szenario wäre z. B. der Austausch einer verteilten Lastverteilungsstrategie ohne Unterbrechung der Ausführung des Systems, wobei die ausgetauschten Komponenten nicht, wie üblich per Indirektion, schwach an das restliche System gebunden würden, sondern in der Intensität ihrer Integration nicht von anderen Komponenten unterscheidbar wären. Das weite Spektrum der Möglichkeiten, das in diesem Abschnitt aufgespannt wurde, muß allerdings in der verbleibenden Dimension Raum eingeschränkt werden, um demonstrable Ergebnisse erzielen zu können. Vorrangig werden im Folgenden deshalb DA-Komponenten und dabei insbesondere unvollständige M-Akteure und Order zur Flexibilisierung der Systemkonstruktion herangezogen.

## 6.2 INSEL Komponentenkategorien

Zunächst seien folgende INSEL-Rahmenbedingungen rekapituliert:

1. Jede DA-Komponente $a \in X_t$ wird durch Ausführung der Operation *erzeuge* auf dem entsprechenden Generator $G(a)$ inkarniert.

2. Generatoren sind DE-Komponenten und entstehen durch Erarbeitung der Deklaration im Deklarationsteil von DA-Komponenten; d. h. $\exists b \in X_t : G(a) \in L_0(b)$.

3. Die Schachtelungsstruktur $\delta$ ist die invariante definitorische Basis der Ausführungsumgebung jeder DA-Komponente $a \in X_t$.

Insbesondere Punkt 2 bedarf Ergänzungen. Zum einen müssen offensichtlich unvollständige sowie inkrementell zu vervollständigende Generatoren eingeführt werden und zum anderen ist zu klären, auf welcher Grundlage Generatoren erarbeitet werden. Generatoren unterliegen als DE-Komponenten der hohen Dynamik der DA-Komponenten. Als lokale Elemente von DA-Komponenten korreliert die Existenz von Generatoren mit deren Lebenszeiten $\Lambda$. Im Falle eines rekursiven Programmes wird die Dynamik der Erarbeitung und Auflösung von Generatoren besonders deutlich.

```
MACTOR system IS                                  – Hauptakteur, Deklarationsteil (1)
   PROCEDURE a(I : IN INTEGER) IS
      PROCEDURE b(J : IN INTEGER) IS              – geschachtelt in a (2)
      BEGIN
         ...
      END;
   BEGIN
      b(I);                                       – Erzeugung einer δ-inneren Order (3)
      IF I > 0 THEN
         a(I-1);                                  – Rekursionsschritt (4)
      END IF;
   END;
BEGIN                                             – Anweisungsteil des Hauptakteures (5)
   a(42);
END;
```

Abbildung 6.2: Dynamik der Generatoren – Rekursion

**Beispiel 6.1** *In Abbildung 6.2 ist ein rekursives INSEL-Programm dargestellt. Bei seiner Ausführung wird zunächst der Deklarationsteil des Haupt-M-Akteures* `system` *erarbeitet.* $L_0($`system`$)$ *besitzt anschließend ein Element, den Generator* $G(a)$ *für PS-Order der mit dem Text von* `a` *beschriebenen Eigenschaften. In der Hauptphase von op(*`system`*) wird eine Order-Inkarnation* $a_1$ *bezüglich* $G(a)$ *erzeugt.* $a_1$ *erarbeitet seinerseits in der Aufbauphase die Deklaration eines Generators* $G_{a_1}(b) \in L_0(a_1)$*. Im Anweisungsteil von* $a_1$ *wird eine Inkarnation bezüglich* $G_{a_1}(b)$ *erzeugt und anschließend rekursiv* $a_2$ *auf Grundlage von* $G(a)$ *erzeugt. Der interessante Schritt ist nun, daß* $a_2$ *wiederum per Definition 3.1 einen eigenen Deklarationsteil besitzt und mit* $L_0(a_2) = \{G_{a_2}(b)\}$ *einen neuen Generator erarbeitet, wobei* $G_{a_1}(b) \neq G_{a_2}(b)$*. Erreicht* $a_2$ *in seinem Anweisungsteil den* `b`*-Aufruf, so wird b bezüglich* $G_{a_2}(b)$ *erzeugt! Werden nun im Rahmen der inkrementellen Systemkonstruktion Veränderungen an dem* `b`*-Programm*

*vorgenommen, so ist zu klären, welche(r) Generator(en) $G_{a_i}(b)$ bzw. Inkarnationen von der Erweiterung betroffen sein sollen.*

Sinnvoll sind sowohl Erweiterungen einer spezifischen `b`-Inkarnation oder eines einzelnen Generators $G_{a_i}(b)$ als auch sämtlicher Generatoren $G_{a_i}(b)$, die in der Zukunft in Folge weiterer Rekursionsschritte noch erarbeitet werden. Neben diesem Phänomen der Erweiterbarkeit im Falle von Rekursion, das die Dynamik der Generatoren besonders deutlich aufzeigt, ist ganz allgemein zu beobachten, daß die Generatoren $G_{a_i}(b)$ auf der einen Seite zwar Individuen sind, auf der anderen Seite aber eine gemeinsame Wurzel besitzen, die bisher weder mit dem Generator- noch dem Inkarnationen-Konzept von INSEL erfaßt ist.

### 6.2.1 Generatorfamilien

Diese Beobachtungen motivieren die stärkere Differenzierung von INSEL-Komponenten gemäß Komponentenkategorien und die Erweiterung dieser um das neue Konzept der *Generatorfamilien*[2], die ebenfalls Ressourcengeneratoren sind.

**Definition 6.2 (Komponentenkategorien)**
*Ein INSEL-System $\mathcal{S}_t$ besteht zum Zeitpunkt $t$ aus Komponenten folgender drei Kategorien:*

- Generatorfamilie
  *$\mathcal{G}_t$ bezeichnet die Menge der Generatorfamilien von $\mathcal{S}_t$*
- Generatoren
  *$G_t$ bezeichnet die Menge der Generatoren von $\mathcal{S}_t$*
- Inkarnationen
  *$I_t = X_t \cup Y_t$ bezeichnet die Vereinigung der Mengen der DA-Inkarnationen $X_t$ und der DE-Inkarnationen $Y_t$ von $\mathcal{S}_t$*

*Zur Vereinfachung wird auf den Index $t$ verzichtet, falls lediglich genau ein Schnappschuß des Systems betrachtet wird oder der Zeitpunkt aus dem Kontext klar ist.*

Generatorfamilien lassen sich weder als DE- noch DA-Komponenten in die bisher festgelegten INSEL-Komponentenarten einordnen, sondern bilden eine neue, dritte Komponentenkategorie. Sie unterscheiden sich in ihrer Lebenszeit von den Generatoren und Inkarnationen und sind nach ihrer Definition nutzbar bzw. sichtbar, soweit dies der Schachtelungsstruktur entspricht, ohne daß sie zuvor im Rahmen der Dynamik des Systems von einem Akteur erarbeitet werden. Eine Generatorfamilie legt als Ressourcengenerator mit der Signatur, den Deklarations- und Anweisungsteil wichtige Invarianten der jeweiligen Generatoren fest. Ferner sind die Generatorfamilien des Systems statisch in die Schachtelungsstruktur $\delta_t^{\mathcal{G}} \subset \mathcal{G}_t \times \mathcal{G}_t$ eingeordnet, die gleichzeitig eine Lebenszeitabhängigkeit zwischen den Generatorfamilien induziert. $\delta_t^{\mathcal{G}}$ ähnelt der $\delta_t$-Struktur, wobei letztere allerdings auf der Menge $X_t$ der DA-Komponenten definiert ist und wesentlich höherer Dynamik unterliegt.

Im mathematischen Sinn wäre es präziser, statt des Begriffes Generatorfamilien von Generatorfamiliengeneratoren zu sprechen, da Generatorfamilien bereits Mengen erarbeiteter Generatoren sind und damit das Erarbeitungsproblem der Generatoren nicht gelöst werden

[2] Definition 6.2 sowie einige der noch folgenden Definitionen in diesem Kapitel basieren auf Definitionen aus der Diplomarbeit [Reh98b], die im Rahmen der vorliegenden Dissertation entstand.

kann. So müßten korrekter Weise die beiden Generatoren $G_{a_1}(b)$ und $G_{a_2}(b)$, aus dem Beispiel oben, nach ihrer Erarbeitung genau einer gemeinsamen Generatorfamilie angehören, wobei dies lediglich eine Strukturierung bereits existierender Generatoren darstellen würde aber keinen Aufschluß über deren Entstehung gäbe. Da im Hinblick auf die inkrementelle Systemkonstruktion die Entwicklungsgeschichte von Komponenten von Interesse ist, wird in dieser Arbeit auf die Unterscheidung der Familie und dem Familiengenerator verzichtet und der kürzere Begriff Generatorfamilie verwendet, sofern keine anderweitigen Erläuterungen ergehen.

### 6.2.2 Erzeugungs- und Auflösungsabhängigkeiten

Mit den Generatorfamilien kann die Frage nach der Erarbeitung von Generatoren beantwortet werden. Mit den INSEL-Konzepten wurde bereits festgelegt, daß Generatoren eine implizit definierte, äußere Operation *erzeuge* zu Erzeugung von Inkarnationen anbieten. Generatorfamilien besitzen als Ressourcengeneratoren eine äußere Operation zur Produktion von Ressourcen, wobei es sich in diesem Fall um Generatoren handelt. Definition 6.3 faßt diese Abhängigkeiten bezüglich der Entstehung von INSEL-Komponenten zusammen.

**Definition 6.3 (Erzeugungsordnung)**

- *Sei $\mathcal{A} \in \mathcal{G}$ eine Generatorfamilie und $b \in X$ eine DA-Komponente. Wird die $\mathcal{A}$-Beschreibung durch $b$ erarbeitet, so entsteht der Generator $A \in L_0(b)$. $A$ ist ein Element der Menge der Generatoren bezüglich der Generatorfamilie $\mathcal{A}$, $G_{\mathcal{A}} \subseteq G$.*
  *Die Erarbeitungsabhängigkeit des Generators $A$ von der Generatorfamilie $\mathcal{A}$ sei symbolisiert durch $\mathcal{A} \overset{def}{\Rightarrow} A$; gesprochen: Generator $A$ wurde auf Grundlage der Generatorfamilie $\mathcal{A}$ erarbeitet.*

- *Sei $A \in G_{\mathcal{A}}$ ein Generator der Generatorfamilie $\mathcal{A} \in \mathcal{G}$. Durch die Ausführung von „erzeuge“ auf $A$ wird die Inkarnation $a$ als Element der Inkarnationen bezüglich Generator $A$, d. h. $a \in I_A \subseteq I$, erzeugt.*
  *Die Erzeugungsabhängigkeit zwischen Inkarnation $a$ und dem Generator $A$ sei symbolisiert durch $A \overset{erz}{\Rightarrow} a$; gesprochen: $a$ wurde bezüglich $A$ erzeugt.*

Definition 6.3 besagt, daß die Erarbeitung der Deklaration eines Generators $A$ der Ausführung einer äußeren Operation *erarbeite* auf der Generatorfamilie $\mathcal{A}$ entspricht, mit deren Hilfe $A$ erzeugt und in $L_0$ eingeordnet wird. Genauso wie Inkarnationen stets auf Grundlage eines Generators erzeugt werden, werden auch Generatoren auf Basis einer Generatorfamilie erarbeitet

**Satz 6.4**
$\forall t \in \mathbb{R}_+ : \forall a \in X_t : \exists_1 A \in G_t : A \overset{erz}{\Rightarrow} a$
*Das bedeutet, daß für jede DA-Inkarnation der Generator, auf dessen Grundlage sie erzeugt wurde, während ihrer gesamten Lebenszeit eine Komponente des Systems ist.*

**Beweis**
Für die Auflösung von Generatoren gelten als benannte DE-Komponenten die gleichen strengen Gesetzmäßigkeiten wie für lokale N-Komponenten ($\lambda$-Struktur). Da in der Abschlußsynchronisation einer Komponente $a \in X$ auf die Terminierung und Auflösung aller $\alpha$-inneren Inkarnationen gewartet wird, bevor $a$ und damit $L_0(a)$ aufgelöst wird, trifft somit Satz 6.4

auf N-Komponenten und ihre Generatoren zu. Für Z-Komponenten ergibt sich die Aussage unmittelbar aus der Definition der $\gamma$-Struktur vermöge 3.8. □

Gemäß Satz 6.4 ist gewährleistet, daß für jede Inkarnation $x \in X$ der zugehörige Generator als definitorische Basis für das gesamte Intervall $\Lambda(x)$ Bestand hat. Zur Durchsetzung der Klasseneigenschaften von Inkarnationen ist dies mehr als zweckmäßig. Würde dies nicht gelten, so gäbe es verwaiste Inkarnationen, für die es nicht möglich wäre, eindeutig zu klären, ob sie die geforderten Klasseneigenschaften tatsächlich besitzen. Eine Regelung dieser Art sollte dementsprechend auch für Generatorfamilien gelten und ist in Postulat 6.5 formuliert.

**Postulat 6.5 (Auflösungsordnung)**
*Seien $a \in X, A \in G, \mathcal{A} \in \mathcal{G}$ eine Inkarnation, ein Generator und eine Generatorfamilie mit $\mathcal{A} \overset{def}{\Rightarrow} A \overset{erz}{\Rightarrow} a$ und den Lebenszeiten $\Lambda(a) = [t_a, t'_a], \Lambda(A) = [t_A, t'_A], \Lambda(\mathcal{A}) = [t_\mathcal{A}, t'_\mathcal{A}]$, dann muß zusätzlich zu $t_a \geq t_A \geq t_\mathcal{A}$ auch $t'_a \leq t'_A \leq t'_\mathcal{A}$ gelten.*

Auf Grundlage der Rahmenkonzepte des STK-Systemmodells sind Inkarnation, Generatoren und Generatorfamilien Ressourcen. Die Relationen $\overset{def}{\Rightarrow}$ und $\overset{erz}{\Rightarrow}$ spezifizieren Bindungen zwischen diesen, mit Verbindlichkeitsintervallen $\Phi$, die jeweils exakt den Lebenszeiten des erarbeiteten Generators bzw. der erzeugten Inkarnation entsprechen.

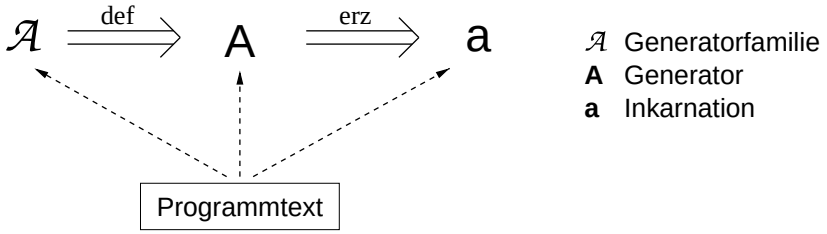


Abbildung 6.3: Komponentenkategorien, Komponentenentwicklung und Programmtext

In der Darstellung 6.3 sind die Zusammenhänge zwischen den Komponentenarten veranschaulicht. Eine Inkarnation $a$ wird mit Hilfe eines Generators $A$ für $a$-Inkarnationen erzeugt. Der Generator $A$ wurde selbst auf Grundlage einer Generatorfamilie $\mathcal{A}$ im Deklarationsteil einer anderen Inkarnation $b$ erarbeitet und befindet sich seither in $L_0(b)$ und somit auch in der Ausführungsumgebung $U(b)$ der Komponente $b$. Deshalb war es $b$ möglich, eine $a$-Inkarnation zu erzeugen.

Programmtext in INSEL-Syntax dient ganz allgemein der Spezifikation der Eigenschaften von Komponenten und ist nicht einer spezifischen Komponentenart zugeordnet. Ungeachtet der Komponentenart sind die Eigenschaften jeder Komponente durch Programmtext spezifiziert. Die exakte Einordnung des Programmtext in die Systematik der Komponenten und Komponentenarten sowie die gezielte Spezifikation von Komponenteneigenschaften ist Gegenstand des nächsten Abschnitts.

## 6.3 Unvollständige Spezifikation

Aus den oben erläuterten und in 6.3 dargestellten Abhängigkeiten ergibt sich zusammen mit der Definition von Ressourcengeneratoren in 5.10, daß Generatoren zum Zeitpunkt ihrer Erarbeitung die mit der Generatorfamilie festgelegten Klasseneigenschaften erben und diese

ggf. angereichert an die später erzeugten Inkarnationen weiter vererben. Für die inkrementelle Weiterentwicklung von Komponenten ist ein konzeptionelles Rahmenwerk erforderlich, mit dem die Eigenschaften einer Komponente zunächst unvollständig festgelegt und zu einem späteren Zeitpunkt vervollständigt werden können. Im wesentlichen bedeutet dies, in den INSEL-Komponenten, die Ressourcen aus $\mathcal{I}$ und damit Zwischenressourcen sind, offene Referenzen zuzulassen, die vorübergehend nicht an eine Bedeutung auf abstrakterem Konkretisierungsniveau gebunden sind. In einem ersten Schritt würde es somit genügen, bei der Konstruktion der Menge $\mathcal{I}$ aus der Menge $\mathcal{N}$ Bezüge auf $\varepsilon$ zuzulassen.

Größere Klarheit bezüglich der Freiheitsgrade wird jedoch erzielt, indem die Menge $\mathcal{N}$ der Nutzungsressourcen selbst um Ressourcen angereichert wird, die lediglich einen Rahmen für ihre Substitution festlegen. Auf den Begriff der Substitution sind dabei zwei Sichtweisen möglich und gleichermaßen zulässig. Bei einer gedachten Expansion des STK-Systemmodells in der Dimension $k$ in Richtung eines höheren Abstraktionsniveaus werden die Ressourcen aus $\mathcal{N}$ zu Tupel- und Referenzressourcen, die sich auf andere Ressourcen mit einer noch abstrakteren Semantik als die INSEL Nutzungsressourcen beziehen. Bei den substituierbaren Ressourcen handelt es sich in dieser Betrachtung um Referenzressourcen, deren variable Eigenschaften durch verzögerte Festlegung der Bindung bestimmt werden. Die jeweilige Qualifikation einer offenen Referenzressource legt den Rahmen für ihre spätere Bindung an eine Bedeutung fest. Der gleiche Effekt wird erzielt, indem man das STK-Systemmodell nicht gedanklich expandiert, sondern Substitution als das tatsächliche Ersetzen der als substituierbar ausgezeichneten Ressource betrachtet. Dies erfordert allerdings eine gesonderte Spezifikation des Spektrums möglicher Ersetzungen.

Unabhängig von der Modellierung kann sich die Substitution auf andere Klassen von Ressourcen aus $\mathcal{N}$ (z. B. unvollständiger Anweisungsteil) oder auch auf Ressourcen, die derzeit noch nicht existieren und erst später zu $\mathcal{N}$ hinzugefügt werden (Erweiterung der Sprache selbst), beziehen. Der letztere Fall wird in dieser Arbeit noch nicht betrachtet. Beispiele für den ersteren existieren bereits seit geraumer Zeit, z. B. mit gewöhnlichen formalen Parametern eines Unterprogrammes oder noch deutlicher mit den variabel – unvollständig – spezifizierten Parameterlisten in C-Programmen mit Hilfe der „`(...)`“ Notation.

$\mathcal{N}^u \subset \mathcal{N}$ sei die Menge der *unvollständigen Nutzungsressourcen*, die zu $\mathcal{N}$ als substituierbare Platzhalter zum Zweck der unvollständigen Spezifikation der Ressourcen aus $\mathcal{I}$ und damit der INSEL-Komponenten hinzugefügt werden. $\mathcal{N}^v = \mathcal{N} \backslash \mathcal{N}^u$ sei die Menge der *vollständigen Nutzungsressourcen*. Beispiele sind die arithmetischen Operatoren sowie Schleifen oder Erzeugungsanweisungen für DA-Komponenten. Für $n \in \mathcal{N}^u$ trifft dies nicht zu. Sie sind entsprechend der Erläuterungen oben Ressourcen, deren Qualifikation und zugehörige Klassendefinition den Rahmen für ihre Substitution festlegen.

Mit diesen Erklärungen und der Differenzierung von $\mathcal{N}$ in $\mathcal{N}^u$ und $\mathcal{N}^v$ ist es nun möglich, Vollständigkeit bzw. Unvollständigkeit von INSEL Komponenten zu definieren.

**Definition 6.6 (Vollständig spezifiziert)**

- *Jede Ressource besitzt die Eigenschaft* vollständig *oder* unvollständig. *Es besagt, ob die Menge ihrer Eigenschaften auf abstrakterem Niveau vollständig festgelegt ist oder Ergänzungen notwendig sind, um alle diesbezüglichen Freiheitsgrade zu determinieren.*
- *$a \in X$ heißt vollständig spezifiziert – $a^v$ – wenn sie ausschließlich aus $n_i \in \mathcal{N}^v$ komponiert ist.*
- *$a \in X$ heißt unvollständig spezifiziert – $a^u$ – wenn sie nicht vollständig spezifiziert ist.*

- *Ein vollständig bzw. unvollständig spezifizierter Ressourcengenerator generiert vollständig bzw. unvollständig spezifizierte Ressourcen.*

### 6.3.1 Freiheitsgrade

Die Freiheitsgrade, die sich grundsätzlich aus Definition 6.6 ergeben, hängen von der Festlegung der Menge $\mathcal{N}^u$ ab und können dementsprechend vielfältig sein. Zum einen ist das Attribut der Vollständigkeit bezüglich der betrachteten Komponentenart zu differenzieren. Möglich sind unvollständige DA-Inkarnationen, unvollständige Generatoren, unvollständige Generatorfamilien sowie vollständige Varianten selbiger drei Kategorien. Ferner sind verschiedene Formen der Unvollständigkeit zu unterscheiden (siehe Abbildung 6.4). Grundsätzlich ist

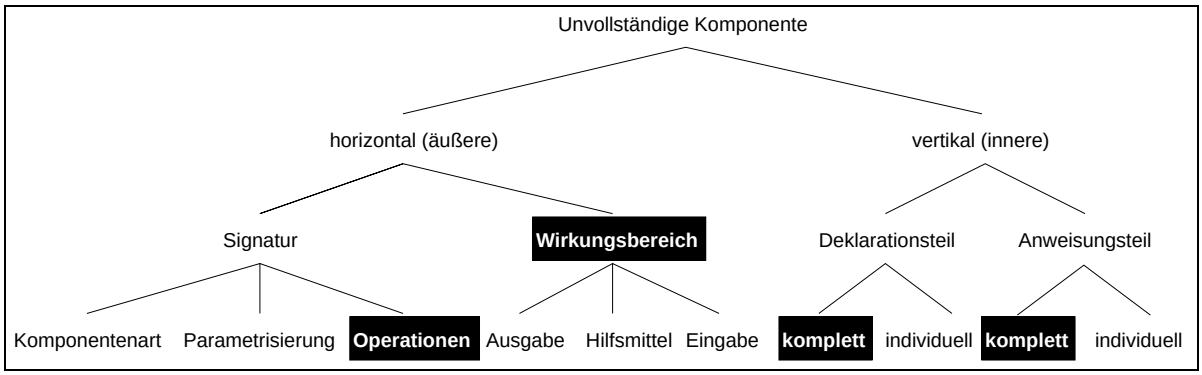


Abbildung 6.4: Taxonomie der Unvollständigkeit von DA-Komponenten

horizontale bzw. äußere - von vertikaler bzw. innerer Unvollständigkeit zu separieren. Im ersteren Fall besitzt die Schnittstelle der Komponenten nach außen Freiheitsgrade. Dies kann bedeuten, daß die Signatur der Komponente noch nicht spezifiziert oder ihr Wirkungsbereich bezüglich lesender und schreibender Zugriffe innerhalb der Ausführungsumgebung $U$ noch unbekannt ist. Bezüglich der Signatur ist zu unterscheiden, ob die Komponentenart, ihre Parametrisierung oder die Menge der explizit definierten äußeren Operationen flexibilisiert sind. Vertikale Unvollständigkeit bezieht sich auf die inneren Eigenschaften der betrachteten Komponente, die sich aus dem Deklarations- und dem Anweisungsteil ergeben. Beide können als Ganzes – komplett – unvollständig spezifiziert sein, oder es bestehen Freiheitsgrade bezüglich einzelner Deklarationen und Anweisungen, deren Festlegung auf einen späteren Zeitpunkt aufgeschoben ist.

#### 6.3.1.1 Festlegung der Menge $\mathcal{N}^u$

Dieses weite Feld an Möglichkeiten wird in der vorliegenden Arbeit insbesondere im Hinblick auf die exemplarische Realisierung auf die in Abbildung 6.4 hervorgehobenen Formen der Unvollständigkeit eingeschränkt. In INSEL sind mit formalen Parametern, Ein-/Ausgabe-Anweisungen und Identifikatoren bereits einige Elemente der Menge $\mathcal{N}^u$ vorhanden. In dieses Repertoire werden zusätzliche Elemente aufgenommen, die als Platzhalter für Deklarationen in Deklarationsteilen von M-Akteuren, S-Order und Depots sowie deren Rumpf bzw. Anweisungsteil dienen können. Programmiersprachlich seien diese neuen Elemente aus $\mathcal{N}^u$ gemeinsam mit dem Terminal `INCOMPLETE` bezeichnet. Aus dem jeweiligen Kontext ist ersichtlich,

ob damit eine Deklaration oder ein Anweisungsteil substituiert wird. Dieses Repertoire ist mächtig genug, um neue Problemlösungen von Benutzern dynamisch in das bereits existierende System und sein Programm integrieren zu können. Weitere Möglichkeiten die sich daraus bereits ergeben, sind die inkrementelle Konstruktion der Menge explizit definierter, äußerer Operationen eines Depots sowie die dynamische Festlegung von PS-Order-Rümpfen, z. B. zur Anpassung von Managementmechanismen.

### 6.3.2 Abhängigkeiten der Eigenschaften

Zwischen den Eigenschaften von Komponenten der verschiedenen Kategorien herrschen Gesetzmäßigkeiten, die den Rahmen für die konsistente Vervollständigung unvollständiger Komponenten festgelegen.

Wohldefinierte Eigenschaftsabhängigkeiten sind sowohl für die Qualität als auch die quantitative Leistung des Systems entscheidend. Ohne sie müßten Komponenten jeweils im Detail betrachtet werden, um Aussagen machen zu können. Für das Management würde dies bedeuten, daß Entscheidungen pro Individuum und nicht für Mengen von Komponenten getroffen werden können, woraus inakzeptabler Mehraufwand entstehen würde. Vermieden wird diese Situation durch die Bildung von Klassen. Im Rahmen der INSEL-Konzepte in Kapitel 3 wurden deshalb Generatoren für Inkarnationen und in Kapitel 5 das verallgemeinerte Konzept der Ressourcengeneratoren eingeführt, in das sich das Generatorfamilienkonzept homogen einordnet. In diese Systematik der Klassenbildung muß die konzeptionelle Unvollständigkeit ebenfalls eingeordnet werden. Die Menge der Eigenschaften einer Komponente $x$ sei im Folgenden durch $E(x)$ symbolisiert.

**Satz 6.7**

- *Die Eigenschaften $E(\mathcal{A})$ einer Generatorfamilie $\mathcal{A} \in \mathcal{G}$ sind in der Sprache INSEL spezifiziert und definieren Invarianten der Generatoren $A \in G_{\mathcal{A}}$, d. h. $\forall A : \mathcal{A} \overset{def}{\Rightarrow} A$.*
- *Die Eigenschaften $E(A)$ eines Generator $A \in G$ sind in der Sprache INSEL spezifiziert und definieren Invarianten der Inkarnationen $a \in I_A$, d. h. $\forall a : A \overset{erz}{\Rightarrow} a$.*
- *Die Eigenschaften $E(a)$ einer Inkarnation sind in der Sprache INSEL spezifiziert.*

**Beweis**
Folgt aus der Definition der klassendefinierenden Ressourcengeneratoren in 5.10 und der Einführung von Generatorfamilien und Generatoren als solche. Ferner wurde INSEL als Beschreibungsinstrumentarium festgelegt. □

Satz 6.7 bedeutet gleichzeitig, daß jede Inkarnation die Klasseneigenschaften des zugehörigen Generators und jeder Generator die seiner Generatorfamilie ererbt. Jede Inkarnation bzw. jeder Generator besitzt überdies individuelle Eigenschaften, z. B. Nutzung, die nicht mit ihrem Generator bzw. seiner Generatorfamilie festgelegt sind. Daraus ergibt sich folgendes Korollar.

**Korollar 6.8 (Eigenschaftsabhängigkeit)**
*Seien $a \in X, A \in G, \mathcal{A} \in \mathcal{G}$ eine Inkarnation, ein Generator und eine Generatorfamilie mit $\mathcal{A} \overset{def}{\Rightarrow} A \overset{erz}{\Rightarrow} a$, dann gilt:*

$$E(\mathcal{A}) \sqsubseteq E(A) \sqsubseteq E(a)$$

Diese konzeptionelle Abhängigkeit zwischen den Eigenschaften von Komponenten unterschiedlicher Kategorien muß auch im Falle unvollständiger Spezifikation gewährleistet bleiben und definiert eine Minimalanforderung an konsistente Übergänge von unvollständig nach vollständig und umgekehrt.

### 6.3.3 Transitionen

Konzeptionelle Unvollständigkeit erlaubt die Festlegung unvollständiger Generatorfamilien und die Produktion unvollständiger Generatoren und Inkarnationen, die zu einem späteren Zeitpunkt vervollständigt werden. Der spätere Übergang von unvollständig nach vollständig wird *Vervollständigung* genannt und entspricht dem Einschränken der Freiheitsgrade einer Komponente. Dabei ist zwischen totaler und partieller Vervollständigung zu unterscheiden. Während im ersteren Fall sämtliche Freiheitsgrade bezüglich der Spezifikation der Eigenschaften in einem Schritt eliminiert werden, so werden im letzteren die abstrakten Eigenschaften einer individuelle Komponente sehr viel aufwendiger schrittweise ergänzt. Für die inkrementelle Konstruktion des V-PK-Systems genügt es, die totale Vervollständigung von DA-Komponenten, DA-Generatoren und -familien zu betrachten, so daß im Hinblick auf die Realisierung von partiellen Vervollständigungen abgesehen wird. Je nach Kategorie der Komponente verbleiben demnach drei verschiedene Arten der Vervollständigung.

**Definition 6.9 (Vervollständigung)**
*Seien $\mathcal{A} \in \mathcal{G}, A \in G, a \in I$, $\prec$ symbolisiere den Übergang von unvollständig nach vollständig. Es existieren drei verschiedene Arten der Vervollständigung von Komponenten:*

*Generatorfamilienvervollständigung (GFV):* $\mathcal{A}^u \succ \mathcal{A}^v$
*Generatorvervollständigung (GV):* $A^u \succ A^v$
*Inkarnationsvervollständigung (IV):* $a^u \succ a^v$

Dies bedeutet, daß abgesehen von Konsistenzbedingungen, jede Komponente ungeachtet ihrer Art individuell vervollständigt werden kann. Die Eigenschaften, die bei einer IV festgelegt werden, gelten ausschließlich für die betrachtete Inkarnation. Die Tragweite einer GV ist größer. Zu dem modifizierten Generator selbst erhalten auch alle Inkarnationen, die auf seiner Grundlage im Anschluß erzeugt werden, die nachträglich spezifizierten Eigenschaften. Entsprechend weitreichender sind die Auswirkungen einer GFV, da sämtliche im Anschluß erarbeiteten Generatoren dieser Familie und die von ihnen erzeugten Inkarnation die Eigenschaften der Vervollständigung erhalten. Unabhängig von der Art Komponente $x$, die vervollständigt wird, werden Ressourcen aus $\mathcal{N}^u$ durch Ressourcen aus $\mathcal{N}^v$ substituiert, wodurch $x$ um zusätzliche Eigenschaften *erweitert* wird. Praktisch erfordert dies die Spezifikation der Erweiterung des $x$-Programmes in INSEL und bedient sich somit der P-Erweiterbarkeit aus Definition 1.1. Hierdurch wird nochmals deutlich, daß jede Komponente ungeachtet ihrer Art durch ihren eigenen INSEL-Programmtext beschrieben ist, andernfalls wäre es nicht möglich, die verschiedenen Arten der Vervollständigung zu separieren.

Für die Evolutions- und Adaptionsfähigkeit des Systems ist die Möglichkeit der Revision bereits getroffener Entscheidungen zusätzlich zu geplanten Freiheitsgraden essentiell, da vermutlich nur dann im erforderlichen Umfang auf unvorhergesehene Veränderungen reagiert werden kann. Das Revidieren setzt voraus, daß bereits getroffene Entscheidungen zurückgenommen und durch neue Freiheitsgrade ersetzt werden können. Die Verallgemeinerung der Komponenteneigenschaften ist somit das Gegenstück zur Vervollständigung.

**Definition 6.10 (Verallgemeinerung)**
*Seien $\mathcal{A} \in \mathcal{G}, A \in G, a \in I$, $\succ$ symbolisiere den Übergang von vollständig nach unvollständig. Es existieren drei verschiedene Arten der Vervollständigung von Komponenten:*

*Generatorfamilienverallgemeinerung (GFR):* $\mathcal{A}^v \prec \mathcal{A}^u$
*Generatorverallgemeinerung (GR):* $A^v \prec A^u$
*Inkarnationsverallgemeinerung (IR):* $a^v \prec a^u$

Weitere Definitionen sowie ein Ansatz zur Realisierung der Verallgemeinerung sind in [Reh98b] nachzulesen und werden hier nicht weiter vertieft als dies für die Gesamtthematik der inkrementellen Systemkonstruktion erforderlich ist.

#### 6.3.3.1 Zeitpunkt der Vervollständigung

Eine Vervollständigung muß offensichtlich spätestens dann erfolgen, wenn im Zuge des Fortschritts einer Akteur-Berechnung eine noch nicht spezifizierte Eigenschaft einer Inkarnation benötigt wird. Im Kontext des oben erklärten `INCOMPLETE`–Konzeptes sind die Erarbeitung eines noch unvollständigen Deklarationsteiles bzw. der Übergang in die Hauptphase der Ausführung der kanonischen Operation, bei einem „unvollständig“ spezifizierten Anweisungsteil mögliche Situationen. Die Vervollständigung ist dann zwingend erforderlich, um die Berechnung des Akteurs fortsetzen zu können. Selbstverständlich sind Vervollständigungen vor dieser konkreten Zwangssituation sinnvoll und gestattet. Der frühestmögliche Zeitpunkt der vollständigen Spezifikation einer Komponente ist naturgemäß ihre Erzeugung.

#### 6.3.3.2 Konsistenz

Zu klären verbleibt die wichtige Frage, welche Kriterien konsistente Vervollständigungen und Verallgemeinerungen erfüllen müssen. Die Abhängigkeit zwischen den Eigenschaften der Komponenten gemäß Korollar 6.8 induzieren eine Reihenfolgerestriktion auf den verschiedenen Arten der Vervollständigungen und Verallgemeinerungen.

**Postulat 6.11**
*Das Zeichen $\bowtie$ symbolisiere „ist zulässig, wenn gilt“:*

$$
\begin{array}{lllll}
GFV & - \mathcal{A}^u \succ \mathcal{A}^v & \bowtie & \nexists A{:}\mathcal{A}^u \overset{def}{\Rightarrow} A \\
GV & - A^u \succ A^v & \bowtie & \nexists a : A^u \overset{erz}{\Rightarrow} a \\
GFR & - \mathcal{A}^v \prec \mathcal{A}^u & \bowtie & \nexists A{:}\mathcal{A}^v \overset{def}{\Rightarrow} A \\
GR & - A^v \prec A^u & \bowtie & \nexists A{:}\mathcal{A}^v \overset{def}{\Rightarrow} A \wedge \nexists a : A^v \overset{erz}{\Rightarrow} a \\
IR & - a^v \prec a^u & \bowtie & \nexists a : A^v \overset{erz}{\Rightarrow} a
\end{array}
$$

*IV sind stets zulässig.*

Postulat 6.11 besagt prinzipiell, daß einerseits die Eigenschaften eines unvollständigen Ressourcengenerators nicht individuell vervollständigt werden können, so lange Ressourcen existieren, die auf seiner Grundlage erzeugt wurden und andererseits eine Ressource nicht verallgemeinert werden kann, wenn der Ressourcengenerator festlegt, daß die Eigenschaften der von ihm erzeugten Ressourcen vollständig spezifiziert sind. Die Aussage, daß IV stets möglich sind, ist erklärungsbedürftig. Die Voraussetzung für eine IV einer Inkarnation $a^u$ ist, daß es einen unvollständigen Generator $A^u \overset{erz}{\Rightarrow} a^u$ gibt. Wird nun $a^u$ zu einem beliebigen Zeitpunkt

vervollständigt, so bleibt die Eigenschaftsabhängigkeit aus Korollar 6.8 auch anschließend in der Form $E(A^u) \sqsubseteq E(a^v)$ gewährleistet, da lediglich $a$ um Eigenschaften erweitert wird, aber immer noch die Eigenschaften des Generators $A$ besitzt.

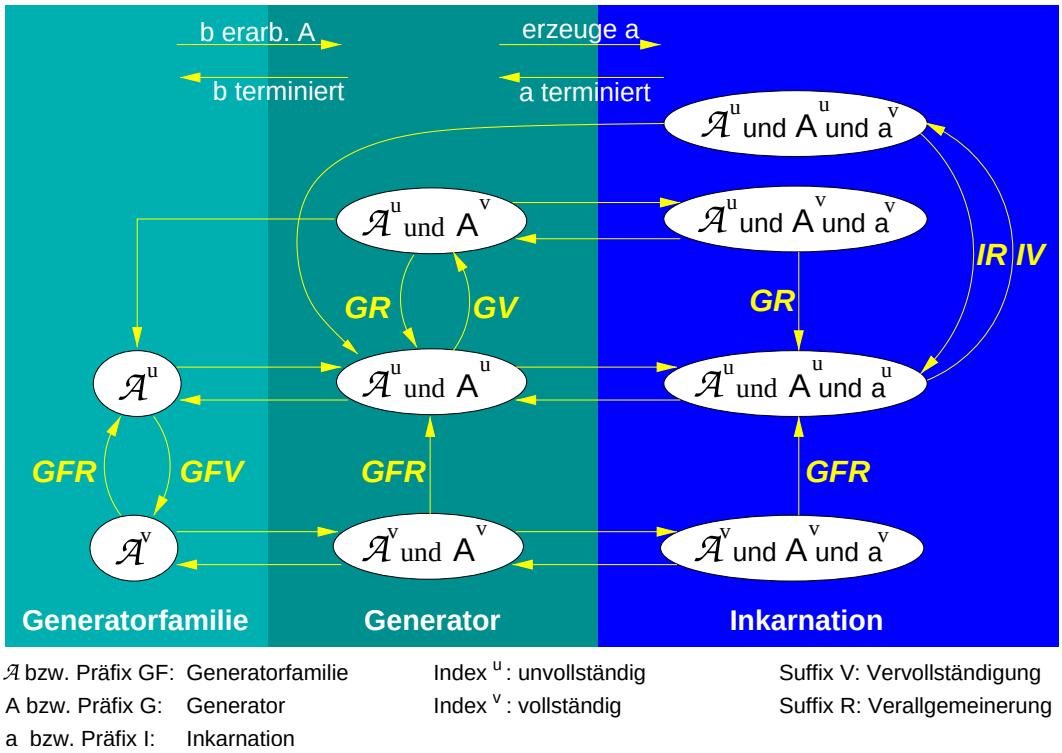


Abbildung 6.5: Zustandsübergänge der Komponentenevolution

In Abbildung 6.5 sind die Möglichkeiten der Entwicklung von INSEL-Komponenten die sich aus dieser Forderung ergeben in einem Zustandsübergangsdiagramm dargestellt. Hierbei sei $b \in X$ eine DA-Komponente, die den Generator $A$ auf Grundlage der Familie $\mathcal{A}$ erarbeitet und eine Inkarnation $a$ erzeugt. Die Knoten des Diagramms sind die Elemente des kartesischen Produkts der Komponentenarten jeweils mit der Eigenschaft vollständig bzw. unvollständig. Horizontale Kanten sind die Erarbeitungs-, Erzeugungs- und Terminierungsereignisse; vertikale Kanten sind Übergänge bezüglich der Vollständigkeit. Dabei ist zu beachten, daß das Diagramm auf der zusätzlichen Annahme beruht, daß Inkarnation und ihre Generatoren bzw. Generatoren und ihre -familien in einem Schritt gemeinsam verallgemeinert werden können. Dies erlaubt eine Abschwächung der strengen Forderung aus 6.11. Ressourcengeneratoren können demnach auch dann verallgemeinert werden, wenn von ihnen erzeugte Ressourcen existieren, indem diese Ressourcen gemeinsam mit ihrem -generator verallgemeinert werden. Die grundlegende Eigenschaftsabhängigkeit aus 6.8 bleibt dadurch erhalten und die Flexibilität wird erhöht. Im Bild ist dies deutlich daran zu sehen, daß nun auch GFR stets möglich sind und GR nur von der Vollständigkeit der zugehörigen Generatorfamilie abhängen.

Mit Postulat 6.11 ist eine zentrale Forderung an die konsistente Vervollständigung von Komponenten des Systems formuliert. Ihre Einhaltung genügt alleine jedoch nicht, um die Konsistenz des Systems zu gewährleisten. Hierfür sind einige weitere Bedingungen insbeson-

dere an die Semantik von Vervollständigungen zu knüpfen. So wäre bei einer Vervollständigung der Schnittstelle zu prüfen, inwiefern diese mit der verbindlich festgelegten Nutzung der Komponente durch andere vollständig spezifizierte Komponenten übereinstimmt. Im Detail bedeutet das die Überprüfung der Parametrisierung bzw. der Signaturen nach außen exportierter Komponenten. Bei einer Vervollständigung des Anweisungsteiles ist zu hinterfragen, ob die nachträglich festgelegten Zugriffe auf Komponenten in der Ausführungsumgebung mit den ursprünglich vereinbarten Freiheitsgraden verträglich sind und vieles mehr. Zusätzliche Anforderungen entspringen der Parallelität der Akteur-Berechnungen sowie deren Kooperationsverhalten. Die Gewährleistung der semantischen Korrektheit ist dementsprechend wichtig aber auch vielschichtig und aufwendig. Ferner hängt sie von der Präzision der Spezifikation der Freiheitsgrade mittels unvollständiger Nutzungsressourcen und nicht zuletzt von technischen Randbedingungen ab.

In der vorliegenden Arbeit müssen einige dieser generellen Fragen im Hinblick auf die noch folgende Konstruktion eines leistungsfähigen V-PK-Managements offen bleiben. Die konkrete Festlegung der unvollständigen Nutzungsressourcen mit der syntaktischen Ausprägung `INCOMPLETE` (s.o.) vereinfacht allerdings die Anforderungen an semantisch korrekte Vervollständigungen stark, so daß einige Überprüfung von dem INSEL-Übersetzer auf Sprachniveau tatsächlich vorgenommen werden. Dazu gehören:

- Komponentenkategorie
- Komponentenart: M-Akteur, S-Order, Depot
- Komponentenabschnitt: Anweisungs- oder Deklarationsteil
- Syntaktische Korrektheit der Spezifikation
- Gewährleistung der Integrität der Ausführungsumgebung

## 6.4 Views

Im Kapitel 3.2.2 wurde die Ausführungsumgebung $U(a), a \in X$ erläutert, die auf Grundlage der $\delta$-Struktur die Menge der Komponenten inklusive Generatoren definiert, die von der DA-Komponente $a$ genutzt werden können. Vergleichbare Festlegungen müssen auch für die neue Komponentenkategorie Generatorfamilie getroffen werden. Ferner ist zu klären, wie sich Vervollständigungen der Eigenschaften unvollständiger Komponenten auf den Rest des INSEL-Systems auswirken. Dazu wird als Erweiterung der Ausführungsumgebung das Konzept der *Views* eingeführt, in die zusätzlich zu DA- und DE-Komponenten auch die Generatorfamilien eingeordnet sind.

**Definition 6.12 (View einer Inkarnation)**
*Sei* $a \in X_t$,

$$\mathcal{G}_t(a) := \{\mathcal{A} \in \mathcal{G}_t : \mathcal{A} \textit{ wurde in } A \textit{ plaziert } \vee \mathcal{A} \in V_t(b) : (a,b) \in \alpha\}$$
$$V_t(a) := \{\mathcal{G}_t(a) \cup U_t(a)\}$$

*Der View* $V_t(a)$ *einer Inkarnation* $a \in X_t$ *beinhaltet die Komponenten aller Kategorien, die* $a$ *zum Zeitpunkt* $t$ *zur Erarbeitung von Generatoren, zum Erzeugen von Inkarnationen oder zur Durchführung von Berechnungsschritten zur Verfügung stehen.*
$\mathcal{G}_t(a)$ *ist die Menge der von* $a$ *nutzbaren Generatorfamilien.*

Diese Definition besitzt zwei besonders wichtige Aspekte. Der erste ist die *Plazierung* von Generatorfamilien im System. Dadurch werden auch Generatorfamilien mit ihrem Programmtext in die Dynamik des Systems eingeordnet. Wird eine Inkarnation aufgelöst in die eine Generatorfamilie $\mathcal{A}$ plaziert wurde, so wird auch die Generatorfamilie aus dem System entfernt. Die Lebenszeit einer Generatorfamilie $\mathcal{A}$ – $\Lambda(\mathcal{A})$ – reicht somit von dem Zeitpunkt ihrer Plazierung in einer DA-Komponente $a$ bis zur Auflösung von $a$. Diese Bindung von Generatorfamilien an Inkarnationen gibt dem Management Aufschluß über die adäquate Verwaltung des INSEL-Quellcodes durch das Management.

Der zweite Aspekt von Definition 6.12 ist die Miteinbeziehung der $\alpha$-*Struktur* in das Erarbeiten und Erzeugen von Komponenten zusätzlich zur $\delta$-Struktur, die mittelbar durch $U$ die Gestalt der Views mitbestimmt. Generatorfamilien werden inklusive ihrer Eigenschaft der Vollständigkeit oder Unvollständigkeit sukzessive entsprechend der Dynamik der Berechnungen nach $\alpha$-innen vererbt, wodurch die Tragweite einer Vervollständigung sehr flexibel variiert werden kann. IV besitzen die geringste Zeit-Granularität, da individuelle Inkarnationen angepaßt werden und die Wirkung einer solchen Vervollständigung auf die Lebenszeit der Inkarnationen beschränkt ist. Eine GV eines Generators $A$ ist von wesentlich größerer Tragweite, da die Vervollständigung in die Views der Inkarnation $a \in X : A \in L_0(a)$ und alle $\delta^*$-inneren Komponenten von $a$ einfließt. Eine GFV wirkt sich darüber hinaus auf den View der Komponente $a \in X$, in der die Generatorfamilie vervollständigt wird, sowie die Views aller $\alpha^*$-Nachfolger von $a$ aus.

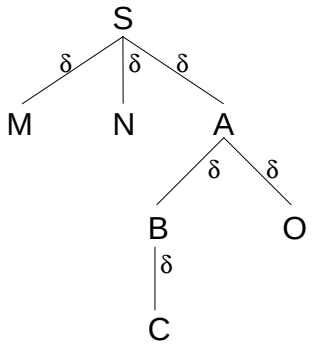


Abbildung 6.6: Beispielszenario

**Beispiel 6.13** *In Abbildung 6.6 ist ein Beispielszenario angegeben, das zur Illustration der unterschiedlichen Auswirkungen verschiedener Arten von Vervollständigungen dient. In die Generatorfamilie $S$ sind die Familien $M, N$ und $A$ $\delta^{\mathcal{G}}$-geschachtelt, wobei $B, O$ $\delta^{\mathcal{G}}$-innen zu $A$ sind und $C$ weiter zu $B$. Sämtliche Familien seien zunächst in der Inkarnation $s$ plaziert. Bild 6.7 zeigt eine mögliche dynamische Entwicklung des Systems und die Tragweiten von IV, GV und GFV.*

**Satz 6.14**
*Der View $V_t(a)$ von $a \in X_t$ ergibt sich aus den GV und GR der unmittelbar $\delta$-äußeren Inkarnation, modifiziert um die GFV und GFR der $\alpha$-Vorgänger.*

Satz 6.14 (in abgewandelter Form aus [Reh98b], S. 33) ist eine andere Formulierung der Definition 6.12, in der die hier erklärten Zusammenhänge von $\delta$- und $\alpha$-Vererbung sowie GV, GR,

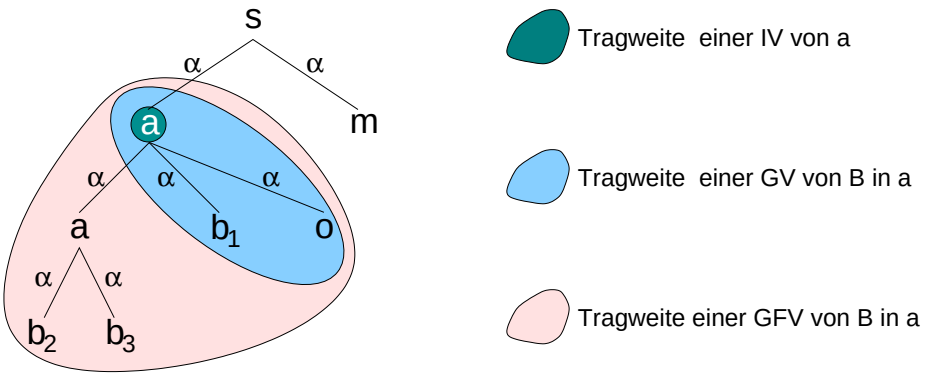


Abbildung 6.7: Tragweiten von Vervollständigungen im $\alpha$-Graphen

GFV und GFR prägnant zusammengefaßt sind. In Abbildung 6.8 ist die Komposition eines Views einer Inkarnation $a_2 \in X$ dargestellt. Auf der linken Seite ist die Schachtelungsstruktur angegeben, in der Mitte der Schnappschuß des $\alpha$-Graphen einer Ausführung, bei der Rekursionen bezüglich $M$ und $A$ stattgefunden haben. $V_t(a_2)$ besitzt einen $\delta$-ererbten Bestandteil von dem jüngsten statischen Vorgänger $m_2$ sowie $\alpha$-ererbte Bestandteile von dem unmittelbarem $\alpha$-Vorgänger $a_1$. In $V_t(a_2)$ wurden überdies bereits eigene Modifikationen vorgenommen.

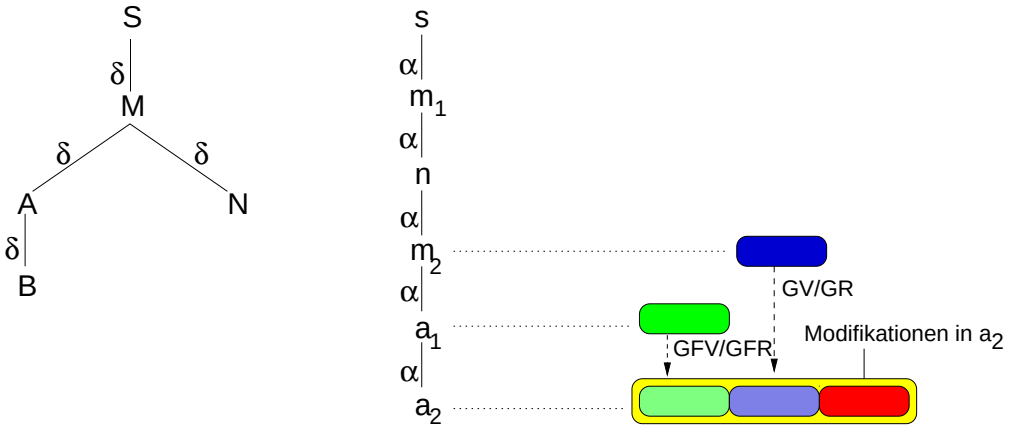


Abbildung 6.8: View einer Inkarnation

Es sei bemerkt, daß nur durch das Einbeziehen der wesentlich dynamischeren Ausführungsstruktur Rekursionsprobleme, die geringfügig komplexer als im Beispiel 6.1 sind, zufriedenstellend gelöst werden können! Man stelle sich in dem dortigen Beispiel vor, daß der Rumpf von $b$ mit der Generatorfamilie als `INCOMPLETE` spezifiziert wäre und in der vorletzten Zeile zusätzlich zu `a(42)` ein weiterer Aufruf `a(43)` stattfände. Würde $\alpha$ nicht miteinbezogen, so gäbe es lediglich die Wahl, entweder die Generatorfamilie von $b$ bereits bei Aufruf von `a(42)` zu vervollständigen, oder den $b$-Generator in jedem Rekursionsschritt bezüglich $a$ erneut zu vervollständigen. Letztere Alternative wurde bereits als inpraktikabel verworfen und die erste würde dazu führen, daß nach `a(42)` und vor `a(43)` alle Freiheitsgrade bezüglich $b$ bereits verloren wären und höchstens durch eine aufwendige Verallgemeinerung rückgewonnen werden

könnten. Beide Alternativen wären unbefriedigend. Durch die Integration von $\alpha$ steht nun die geeignete Alternative zur Verfügung, GFV von $b$ individuell jeweils in den $a$-Inkarnationen vorzunehmen, die wegen Definition 6.12 nicht nach außen an den M-Akteur `system` getragen werden.

## 6.5 Realisierungsaspekte

In der Einleitung zu dem vorliegenden Kapitel wurde darauf hingewiesen, daß sich die Überlegungen auf ein hohes konzeptionelles Niveau und die Sprache INSEL konzentrieren werden. Dessen ungeachtet wurde ein Großteil der hier erläuterten Konzepte bereits in einem einsatzfähigen System erfolgreich realisiert. Details der Realisierung mittels Übersetzer und flexibler Bindertechniken finden sich in [Reh98b, Jek, Piz97] und werden überdies in der vorliegenden Arbeit im eingeschränkten Umfang bei der Realisierung des V-PK-Managements in Kapitel 7 erläutert. Im Folgenden werden zusätzlich die wichtigsten Aspekte der Realisierung skizziert.

INSEL-Programme und ihre zugehörigen Übersetzerprodukte – Maschinenprogramme – sind als INSEL-Depots in dem System repräsentiert, die als wurzelnahe Komponenten dynamisch erzeugt werden. Die Granularität der separat transformierbaren Einheiten des INSEL-Übersetzers entspricht den individuellen DA-Komponenten S-Order, Depot und M-Akteur, deren Generatoren und Generatorfamilien. Die Information über die Vollständigkeit der Spezifikation wird vom Übersetzer auf Grundlage des Programmtextes analysiert und an die nachfolgenden Managementinstrumente Binder, etc. mittels zusätzlicher Sektionen in der ELF-Objektdatei weitergereicht. Der Binder wurde in die Lage versetzt, flexibel Bindungen zwischen in Ausführung befindlichen Übersetzerprodukten festzulegen und wieder aufzulösen. Den Managern wurde ferner die Aufgabe übertragen, während der Ausführung von Akteurberechnungen die Vollständigkeit der aktuellen Ausführungskomponente $\varphi(x)$ des assoziierten Akteures $x$ zu prüfen. Wird eine unvollständige Inkarnation erzeugt, so wird die Ausführung des Akteures angehalten, ein INSEL-System-Editor gestartet und der Benutzer zur Vervollständigung und ggf. Plazierung der Komponente aufgefordert. Anschließend wird der Übersetzer und Binder inkrementell reaktiviert und die vervollständigte Komponente je nach Wahl des Benutzers als IV, GV oder GFV in das System integriert und die Ausführung von $x$ fortgesetzt. Es sei darauf hingewiesen, daß dies alles nur durch die ohnehin geforderte enge Integration sämtlicher Managementmaßnahmen möglich wurde. Durch diese Vorgehensweise entspricht die Flexibilität etwa interpretierenden Verfahren, die hier allerdings mit ungleich größerer Effizienz erzielt wird.

Der Aufwand für die Realisierung zwang allerdings zu Kompromissen und die technische Machbarkeit mit effizienten Verfahren ist eingeschränkt. So wurde u. a. auf die Realisierung einer tatsächlich inkrementellen Transformation geschachtelter Komponenten bei der Realisierung des Übersetzers verzichtet. Stattdessen werden stets die $\delta^{\mathcal{G}}$-äußeren Komponenten des V-PK-Systems mitübersetzt und der Binder selektiert anschließend die modifizierte Komponente. Ebenso mußte bislang insbesondere, wie bereits weiter oben erwähnt, von der partiellen Vervollständigung von Inkarnationen abgesehen werden, da dies aufwendige Zustandstransfers erfordern und enorme technische Probleme aufwerfen würde.

## 6.6 Zusammenfassung

Das Ziel der in diesem Kapitel erarbeiteten Konzepte war, die Langlebigkeit des Systems durch den Übergang von Anwendungswelten zu einer inkrementell erweiterbaren Systemarchitektur zu gewährleisten und dabei die globale Strukturierung des gesamtheitlichen V-PK-Systems aufrecht zu erhalten, um möglichst präzises Wissen über die Anforderungen zu besitzen. Auf hohem Abstraktionsniveau wurde zu diesem Zweck die neue Komponentenkategorie der Generatorfamilien und die unvollständige Spezifikation von Komponenten aller Kategorien auf konzeptionellem Niveau eingeführt und in der Sprache INSEL durch unvollständige Nutzungsressourcen verankert. Für den Übergang von unvollständig nach vollständig wurden Regelungen getroffen, die einerseits eingehalten werden müssen um die Konsistenz des Systems zu gewährleisten und andererseits partiell automatisiert überprüft werden. Mit dem neuen Konzept der Views wurde die Ausführungsumgebung auf Generatorfamilien ausgedehnt und die Systematik der inkrementellen Vervollständigung miteinbezogen. Auf diese Weise sind flexible und abgestufte Erweiterungen an dem dynamischen System möglich. Schließlich wurde der Programmtext in die Dynamik der Systemberechnungen eingeordnet und die Realisierung der entwickelten Konzepte mittels dem integrierten Instrumentarium des V-PK-Managements skizziert.

Das globalstrukturierte V-PK-System kann nun vielfältig durch neue Benutzerprogramme erweitert werden, die im primitivsten Fall unvollständig spezifizierte M-Akteur-Inkarnationen jeweils für eine Ausführung vervollständigen. Darüber hinaus können diese Konzepte auch zur Flexibilisierung des V-PK-Managements, dessen Realisierung ebenfalls Teil des Gesamtsystems ist, herangezogen werden. Die konsequente Integration in die Gesamtarchitektur von V-PK-Systemen, angefangen von der Sprache über sämtliche Managementebenen hinweg, verhindert, daß durch diese Flexibilisierung neue Fehlerquellen eingeführt werden, die möglicherweise während der Ausführung zu nicht behebbaren Fehlerzuständen führen könnten.

# Kapitel 7

# Konkretisierung der MoDiS Managementarchitektur

V-PK-Systeme werden mit den Konzepten von INSEL auf hohem Abstraktionsniveau, bei vollständiger Opazität der physischen Verteilung, inkrementell und gesamtheitlich konstruiert. Damit wurde eine erste, wichtige Voraussetzung für die universelle Nutzung der physisch verteilten Rechen- und Speicherkapazitäten von NOW-Konfigurationen erfüllt. Der zweite wichtige Schritt zum Erreichen dieses übergeordneten Zieles ist die Konstruktion eines quantitativ leistungsfähigen V-PK-Ressourcenmanagements, das abstrakt verteilte INSEL-Systeme automatisiert auf die physisch verteilten Hardwareressourcen der NOW-Konfigurationen abbildet. Gemäß der Analysen in Teil I dieser Arbeit kann dies nur dann erreicht werden, wenn das V-PK-Management den in Abschnitt 4.4 formulierten Anforderungen genügt und sich in drei Merkmalen grundlegend von herkömmlichen Management- bzw. Betriebssystemen abhebt: Systematische *Integration* von Information und Maßnahmen des Managements, abgestufte *Flexibilisierung* und intensive Nutzung des Potentials *transformatorischer* Maßnahmen.

Als Ausgangspunkt für die Entwicklung des V-PK-Managements mit diesen Merkmalen dient die in Abschnitt 3.4 erläuterte Management-Grobarchitektur von MoDiS, die sich top-down an den Akteursphären (AS) orientiert. Die abstrakte Dezentralisierung des Gesamtmanagements in multiple, autonom und kooperativ agierende AS-Manager wird als vorgegeben betrachtet.

In den folgenden Abschnitten wird zunächst die Aufgabe, Management zu konstruieren, anhand der in Kapitel 5 eingeführten Rahmenkonzepte und dem Modell eines idealisierten Produktionssystems diskutiert und der Bezug zwischen den Begriffen Gesamtsystem, AS-Manager, Instrumentierung, etc. geklärt. Dabei wird eine neue Perspektive eingenommen, die dem Zweck dient, die Fehler der empirischen und bottom-up orientierten Konstruktion von Betriebssystemen zu vermeiden. Im Anschluß an diese generellen Betrachtungen wird der im Rahmen der vorliegenden Arbeit verfolgte Ansatz zur Konkretisierung der MoDiS-Manager, mittels dediziertem und konzertiertem Einsatz eines Repertoires von Realisierungsinstrumenten, erläutert. Dabei wird auch auf die Implementierung des Instrumentariums, mit Schwerpunkt auf den INSEL-Übersetzer und den inkrementellen Binder, eingegangen und aufgezeigt, wie vorhandene, hochwertige Software an die veränderten Anforderungen in V-PK-Systemen angepaßt und somit der enorme Aufwand für die Realisierung der neuen Architektur mit vertretbarem Aufwand und akzeptablen Kompromissen bewältigt werden konnte. Das Aufspalten der Manager im Zuge ihrer Realisierung limitiert die Möglichkeiten, Bindungen zwi-

schen Ressourcen des Systems festzulegen und führt somit zu einer starken Separation von Maßnahmen und Information innerhalb des V-PK-Managements. Die unerwünschten Nebeneffekte dieser Separation wurden bereits mehrfach diskutiert und erfordern Gegenmaßnahmen, um die geforderte Integration des Gesamtmanagements bewerkstelligen zu können. In diesem Zusammenhang werden Veränderungen an den Instrumenten erklärt, mit deren Hilfe der bidirektionale Informationsfluß sowie die Abstimmung von Maßnahmen verbessert wurde.

## 7.1 Leitlinien zur Konstruktion des V-PK-Managements

In der Einleitung der vorliegenden Arbeit wurde bei der Erläuterung übergeordneter, langfristiger Ziele in 1.2 darauf hingewiesen, daß die Methodik des Konstruierens von Betriebssystemen, im Gegensatz zu mechanistischen Details, bislang schwach ausgearbeitet ist und wesentlicher Fortschritte bedarf, damit die qualitativen und quantitativen Mängel, die in Bezug auf verteilte Rechensysteme in 4 erläutert wurden, wirksam vermieden werden können und zusätzlich der steigende Realisierungsaufwand immer komplexerer Systeme bewältigt werden kann.

Im Zuge der Entwicklung des V-PK-Managements wurde der kreative Spielraum des top-down orientierten, gesamtheitlichen MoDiS-Ansatzes bereits mehrfach genutzt, um Beiträge zur langfristigen Systematisierung der Konstruktion von Ressourcenmanagementsystemen zu liefern. Dazu gehören die Betrachtungen von Realisierungsinstrumenten in 2.3.3.2, die Manager-Grobarchitektur in 3.4 sowie die erklärten Ressourcen- und Bindungskonzepte. Vor der konkreten Realisierung des V-PK-Managements werden in diesem Abschnitt die beiden wichtigen Fragen diskutiert, was Ressourcenmanagement ist und wie es grundsätzlich zu konstruieren ist. Diese Überlegungen sind eine top-down orientierte Fortführung der systematisierenden Maßnahmen in Kapitel 5. Im Hinblick auf die noch zu leistende Realisierung des V-PK-Managements wird die abschließende Beantwortung dieser grundlegenden Fragen weder in Anspruch genommen noch angestrebt. Das Ziel dieser Vorüberlegungen ist, ein besseres Verständnis von Managementsystemen und ihrer Konstruktion zu gewinnen, um auf dieser Grundlage signifikante Fortschritte relativ zu anderen Ansätzen sowie den Experimentalsystemen zu erzielen.

### 7.1.1 Das Management eines Produktionssystems

Durch die erweiterte Betrachtung von Betriebssystemen, die sich nicht wie üblich auf die Funktionalität von Kernen beschränkt, sondern den gesamten Entscheidungsgang von der abstrakten Problemlösung bis zu ihrer konkreten, physischen Realisierung betrachtet, sind im Laufe der Entwicklung des V-PK-Managements Probleme deutlich geworden, deren Tragweite nicht auf den Kontext verteilter Betriebssysteme beschränkt ist, sondern für Managementsysteme allgemein von Interesse sind und einen engen Bezug zu Fragestellungen aus dem Bereich der Wirtschaftswissenschaften aufweisen. Allerdings konnten dort keine Antworten auf die aufgetretenen Fragen, z. B. wie werden geeigneter Klassen von Zwischenressourcen geplant, gefunden werden.

Im Gegenzug wurde in der vorliegenden Arbeit ein allgemeines Modell eines Produktionssystems entworfen, das einige wichtige Aspekte veranschaulicht (siehe Abbildung 7.1). Schematisch betrachtet besteht das Produktionssystem (PS) aus vier Komponenten. Erstens sind mit Anforderungen die zu erreichenden Ziele der Produktion festgelegt. Zweitens muß ein gewisses Repertoire an Ressourcen existieren, mit deren Hilfe die Anforderungen erfüllt werden

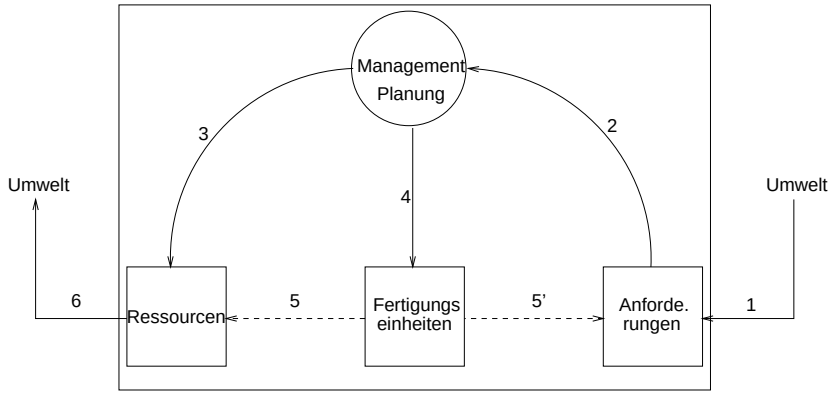


Abbildung 7.1: Schema eines Produktionssystems

können. Drittens verfügt das PS über Fertigungseinrichtungen zur Produktion der Zielressourcen und alles was dazu benötigt wird. Viertens existiert eine übergeordnete Planungsinstanz, die Abläufe kreativ plant, steuert und überwacht. Das PS ist ein offenes System. Von außen werden Anforderungen an das PS herangetragen, die durch die Produktion von Ressourcen befriedigt werden. Unter der Annahme der Langlebigkeit des PS handelt es sich insbesondere bei den ersten drei Komponenten des Systems, inklusive der Anforderungen, um dynamische Mengen. Dies motiviert die Betrachtung von PS-*Konfigurationen*, die Schnappschüsse des PS zu einem Zeitpunkt $t$ beschreiben.

**Bemerkung 7.1 (PS-Konfiguration)**
*Seien $\mathcal{R}, \mathcal{H}, \mathcal{N}, \rho$ wie in Kapitel 5 vereinbart, $\mathcal{RG} \subset \mathcal{R}$ die Menge der Ressourcengeneratoren von PS, $A$ die Anforderungen an das PS und $M$ seine Planungsinstanz.*

$$PS_t = (\mathcal{R}, \rho, \mathcal{RG}, M, A)$$

*beschreibt eine Konfiguration des Produktionssystems PS zum Zeitpunkt $t$.*
*Die **initiale** Konfiguration $PS_{init}$ sei wie folgt definiert:*

$$PS_{init} := (\mathcal{V}, \emptyset, \emptyset, M, \emptyset), \quad \mathcal{V} = \mathcal{H} \cup \mathcal{N}$$

Die Ressourcenkomponente wurde in Bemerkung 7.1 im Hinblick auf das V-PK-Management und die dafür entworfenen Rahmenkonzepte durch Ressourcen und Bindungen präzisiert. Die Fertigungseinrichtungen (FE) entsprechen der Menge $\mathcal{RG}$ der Ressourcengeneratoren, durch die gemäß Definition 5.10 auch die Beschreibungen aller Ressourcenklassen aus $\mathcal{RK}$ mit einer PS-Konfiguration erfaßt sind.

#### 7.1.1.1 Klassifikation von Managementinstanzen und ihrer Aufgaben

Für das Verständnis von Management genügt es, das Fortschreiten der PS-Konfigurationen bei $\Delta t$ Übergängen informell zu beschreiben. Das PS ruht nach Festlegung der initialen Mengen $\mathcal{H}, \mathcal{N}$ sowie der Bestimmung von $M$ zunächst in der durch $PS_{init}$ beschriebenen Konfiguration, bis Anforderungen an PS herangetragen und zu $A$ hinzugefügt werden (1). Aus den Rahmenkonzepten für Ressourcen gemäß Kapitel 5 folgt, daß es sich bei den Anforderungen selbst um Zwischenressourcen handelt, die aus den Nutzungsressourcen $\mathcal{N}$ gefertigt

werden. Der Managementinstanz kommt die Aufgabe zu, diese Anforderungen mit den innerhalb des Produktionssystems zur Verfügung stehenden Ressourcen zu erfüllen (2). Sofern die Anforderungen nicht trivialer Weise unmittelbar mit den bereits vorhandenen Ressourcen befriedigt werden können, sind zwei wesentliche Schritte erforderlich. Erstens müssen geeignete Ressourcenklassen durch Festlegung ihrer invarianten Eigenschaften festgelegt werden. Zweitens müssen die Ressourcen gefertigt werden. Dies kann in Spezialfällen durch $M$ erfolgen (3) oder es werden von $M$ Fertigungseinheiten (FE) geplant und konstruiert, die Ressourcen mit den festgelegten Klasseneigenschaften produzieren (4). Die gefertigten FEs erweitern als Ressourcengeneratoren nochmals die Menge der Ressourcen (5). Zusätzlich entstehen durch die Produktion der FEs neue Anforderungen an das Produktionssystem (5'). So ist schon allein die Gewährleistung der Existenz einer FE eine neue Anforderung an das PS. Der Zyklus fährt somit mit (2) fort, bis die Anforderungen letztendlich erfüllt sind. Die Rekursion kommt zu einem Ende sobald die im Schritt (5') hinzukommenden Anforderungen mit den bereits existierenden Ressourcen oder vorhandenen FEs erfüllt werden können.

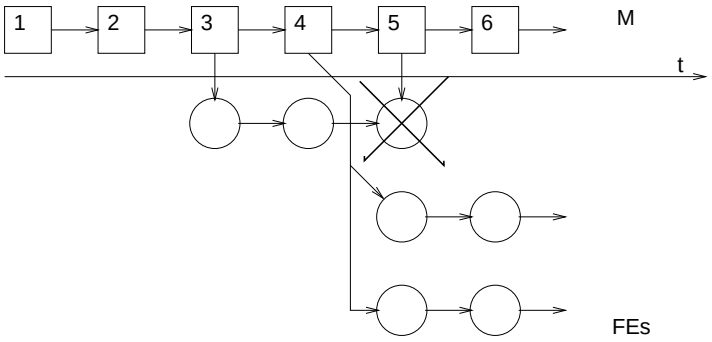


Abbildung 7.2: Delegation von Managemententscheidungen

Aus dieser Betrachtung ergeben sich zwei Kategorien von Managemententscheidungen, deren Zusammenhang in 7.2 dargestellt ist. Zur Befriedigung einer Anforderung besitzt die Managementinstanz $M$ die Alternativen, Ressourcen selbst zu produzieren (1 und 2) oder FEs zu produzieren und die entsprechenden Anforderungen an sie zu *delegieren* (3 und 4). Je nach Bedarf werden FEs von $M$ aus PS wieder entfernt (5). FEs sind in dem durch $M$ geplanten Umfang in der Lage, einfache oder komplexe Aufgaben selbständig zur erfüllen, d. h. die dafür erforderlichen Entscheidungen selbst zu treffen. Mit der Erweiterung der Menge der FEs und dem Delegieren von Anforderungen an diese, wird das Management des Produktionssystems um zusätzliche Instanzen mit dem Ziel der Leistungssteigerung erweitert.

**Bemerkung 7.2 (Managementinstanzen)**
*Eine Managementinstanz ist charakterisiert durch das Treffen von Managemententscheidungen (s. Def. 2.11). Die Menge der Managementinstanzen ist gegeben durch* $\{M\} \cup \mathcal{RG}$.

Zwischen den Fähigkeiten der FEs und der Managementinstanz $M$ ist allerdings eine klare Trennlinie zu ziehen.

- Eine FE $\in \mathcal{RG}$ produziert nach einem festgelegten Verfahren ausschließlich Ressourcen mit den Klasseneigenschaften, die mit dem Ressourcengenerator FE als Invarianten festgelegt wurden.

- M besitzt die zusätzliche Fähigkeit, neue Ressourcenklassen und FEs $\in \mathcal{RG}$ kreativ zu entwerfen.

$M$ ist die kreative Komponente eines Produktionssystems und seines Managements, in der das gesamte Potential zur Erfüllung von Anforderungen an PS vereint ist. Ihre Fähigkeiten sind für die Leistung des Produktionssystems maßgeblich und durch FEs nicht vollständig substituierbar. Das Management eines PS gliedert sich anhand dieser Unterscheidung in das

- **Management** im engeren Sinn, das der übergeordneten Planungsinstanz $M$ entspricht und der

- **Administration** von Ressourcen durch die FEs, die mehr oder weniger komplexes Ressourcenmanagement im weiteren Sinn durchführen, dabei aber keine neue Ressourcenklassen, die noch nicht im System existieren, festlegen.

Ein weiteres Ergebnis dieser Überlegungen ist die Kaskadierung von Management und Administration bei dem Übergang von abstrakt nach konkret, die sich aus dem oben beschriebenen, rekursiven bzw. spiralförmigen Verhalten der Produktion ergibt.

**Bemerkung 7.3 (Produktionsspirale)**
*$k_i$ mit $i \in \mathbb{N}_0$ bezeichne ein Konkretisierungsniveau, wobei $i = 0$ das Niveau der Physik bezeichnet und mit $i$ das Abstraktionsniveau zunimmt.*

1. *Anforderungen auf Niveau $k_n$ müssen erfüllt werden. Existieren auf $k_{n-1}$ Ressourcen, die passende Angebote repräsentieren, so sind keine weiteren Schritte als die Festlegung von Bindungen zwischen $k_n$ und $k_{n-1}$ erforderlich. Andernfalls muß das Management neue Ressourcenklassen für $k_{n-1}$ sowie Verfahren für den Umgang mit diesen planen. Dafür sind kreative Ideen erforderlich.*

2. (a) *Ressourcen werden anschließend auf $k_{n-1}$ administriert, d. h. produziert und aufgelöst.*

   (b) *Mit den Ressourcen aus Schritt 2a entstehen neue Anforderungen auf $k_{n-1}$, deren Erfüllung Fortführung von Schritt 1 mit $k_{n-1}$ erfordert.*

Die Beobachtung dieses Verhaltens ist wichtig, um die Funktion von Produktionssystemen und ihrer Managementinstanzen verstehen und systematisieren zu können. Es wird behauptet, daß die erläuterte Produktionsspirale in jedem System stattfindet bzw. bei dessen Konstruktion stattgefunden hat, um die Anforderungen Schritt für Schritt auf das Rohmaterial der vorgegebenen Ressourcen abzubilden.

Das Problem Management zu konstruieren ist somit im wesentlichen die Frage nach der Konstruktion der administrativen Einheiten und damit das Management im engeren Sinn, während das Ressourcenmanagement eines Betriebssystems zum Zeitpunkt der Ausführung überwiegend als Administration der Ressourcen zu betrachten ist. Auf Grundlage dieser Unterscheidung beschäftigt sich der folgende Absatz kurz mit den Problemen der Planungsinstanz $M$, um sich dann auf den Entwurf flexibler, administrativer Einheiten für das V-PK-Management zu konzentrieren.

#### 7.1.1.2 Entscheidungsfindung des Managements

Da $M$ das gesamte Leistungspotential des Systems in sich vereint, sind höchste Anforderungen an seine Fähigkeiten zu stellen. Unter anderem wäre zu fordern, daß $M$ vollständige Kenntnis über die Angebote der Ressourcen des PS besitzt. Dies entspricht allerdings nicht der Praxis und ist mit den derzeitigen Methoden unrealistisch. Durch den hohen Realisierungsaufwand werden Verfahren nicht systematisch und angepaßt an Anforderungen und Angebote entworfen, sondern ohne präzise Kenntnis ihrer Eigenschaften aus anderen PS übernommen. Ein Beispiel dafür ist die Übernahme einer Zwischenhochsprache und zugehörigem Übersetzer in sprachbasierten Ansätzen wie Orca und den Experimentalsystemen AdaM und EVA. Durch diese Übernahme werden im großen Umfang Ressourcenklassen und FEs in das Produktionssystem und sein Management übernommen, für die es, mit Ausnahme der Reduktion des Aufwands, meist keine Argumentation gibt. Die wichtige, kreative Tätigkeit des Managements läuft dabei Gefahr, zur Administration der Ressourcen der übernommenen Klassenkonzepte und Verfahren zu entarten.

Die zentrale Aufgabe von $M$ wäre jedoch die Festlegung geeigneter Ressourcenklassen. Von der Frage, wie $M$ die adäquaten Kriterien zur Klassenbildung finden kann, hängt die Lösung des auf Seite 50 beschriebenen Suchproblems nach einem Pfad von $\mathcal{N}$ nach $\mathcal{H}$ und damit die Leistung des PS allgemein, bzw. eines (verteilten) Rechensystems im Speziellen, ab. Das Finden dieser Entscheidungen ist äußerst kompliziert und wird im Bereich der Betriebssysteme bislang kaum behandelt. Aufgrund der hohen Komplexität dieser Thematik und dem konkreten Realisierungsziel der vorliegenden Arbeit müssen sich die Überlegungen auch hier auf wesentliche, grundlegende Aspekte beschränken. Die drei zentralen Bestandteile einer leistungsfähigen Planungsinstanz sind wohl:

- Wissen über die Anforderungen und (alternativer) Angebote
- Metriken
- Kriterien

Im ersten Schritt ist möglichst vollständige Information über die Anforderungen und Alternativen für deren Realisierung zu erarbeiten. Mit Metriken werden die möglichen Realisierungsalternativen anschließend bewertet und geprüft, ob eines oder mehrere der Kriterien zündet und eine Entscheidung für oder gegen eine Alternative erlaubt. Ist das nicht der Fall, weil z. B. noch nicht genug Information vorliegt, fährt der Entscheidungsprozeß fort, bis ein Kriterium zündet, das eine Entscheidung mit befriedigender Schärfe anzeigt. Betrachtet man die vielschichtige und abgestufte Suche nach einem „optimalen“ Pfad von $\mathcal{N}$ nach $\mathcal{H}$ (siehe Abb. 2.10), so ist deutlich zu erkennen, daß wegen der kombinatorischen Komplexität eine simple Tiefensuche abwegig ist. Dies würde bedeuten, einen kompletten Pfad von den Anforderungen zu den Angeboten zu konstruieren, um ihn anschließend zu bewerten und – im häufigsten Fall – letztendlich zu verwerfen. Leider ähnelt diese Vorstellung allerdings sehr der Vorgehensweise, mit der Betriebssysteme in der Praxis konstruiert werden. Die abschließende Bewertung ist das Messen von Ausführungszeiten, in deren Anschluß Optimierungen nahe an den Blättern des Entscheidungsbaumes durchgeführt werden.

Benötigt wird dahingegen ein abgestuftes Suchverfahren, bei dem schon früh die Konsequenzen von Entscheidungen mit hinreichender Präzision bewertet und Revisionen durchgeführt werden, um bereits konstruierte Teilpfade zu verwerfen und den Entscheidungsprozeß mit der nun präziseren Information an höherer Position fortzusetzen. Um geeignete und

ungeeignete Pfade früh unterscheiden zu können, muß das Management Entscheidungsmuster bzw. charakteristische Bindungskomplexe, die von Details abstrahieren, zur Beurteilung der aktuellen Situation einsetzen. So sollte es, wie im Beispiel der Verwaltung der Keller in Multi-Threaded Systemen (S. 4.1.1.2), schon früh möglich sein, eine Entscheidung zwischen grundlegenden Varianten, wie Fehlererkennung kontra -verhinderung, oder lazy evaluation und prefetching zu treffen und nachträglich schwer bis nicht behebbare Fehlentwürfe auszuschließen.

Strategische Planung wird hier in seiner Allgemeinheit nicht weiter vertieft. Die weitere Konkretisierung des V-PK-Managements konzentriert sich auf das Erhalten und Schaffen von Freiheitsgraden, die eine Grundvoraussetzung sind, damit das Management über ein ausreichendes Repertoire an Angeboten verfügt und dieses bei Bedarf erweitern kann. Einerseits wird angestrebt, die Freiheitsgrade der top-down Vorgehensweise weitestgehend zu erhalten, indem das gesamte Repertoire an Managementinstanzen von Übersetzer bis Kern in das V-PK-Management miteinbezogen und zudem durch Anpassung existierender Systemprogramme der Realisierungsaufwand und sein negativer Einfluß auf die Entscheidungsfindung reduziert wird. Andererseits werden neue Realisierungsverfahren kreativ entwickelt, wodurch das Repertoire der Angebote vergrößert und neue Freiheitsgrade geschaffen werden.

Der Begriff V-PK-Management hat bislang das Management im weiteren Sinn inklusive der Ressourcenadministration bezeichnet. In den noch verbleibenden Erläuterungen wird dies nicht mehr geändert. Von der expliziten Unterscheidung zwischen Management und Administration wird nur dann Gebrauch gemacht, wenn die Differenzierung relevant erscheint und nicht aus dem Kontext hervorgeht.

### 7.1.2 Das INSEL-Produktionssystem

Durch die Einordnung des INSEL-Systems und seines Managements in das allgemeine Modell des Produktionssystems, klären sich einige Zusammenhänge zwischen den verschiedenen Sichten auf das INSEL-Gesamtsystem. Ferner ist es nun möglich, die Ziele des V-PK-Managements mit den Rahmenkonzepten für Ressourcen und Bindungen sowie den Erkenntnissen aus der Betrachtung des Produktionssystems zu präzisieren, ohne sich dabei an physischen Betriebsmitteln oder bekannten Verfahren orientieren zu müssen.

#### 7.1.2.1 Sichten auf das System

In Abbildung 7.3 sind analog zu dem allgemeinen Modell aus Abbildung 7.1, mit Ausnahme der Planungsinstanz, alle Komponenten des INSEL-Produktionssystems sowie deren Zusammenhänge skizziert.

1. An die Position der Anforderungen tritt das abstrakte INSEL-Gesamtsystem, das sich aus INSEL-Komponenten – Akteure, Order, Depots, etc. – und strukturellen Abhängigkeiten zwischen den Komponenten zusammensetzt. Im rechten Teil der Darstellung ist der $\alpha$-Verband des INSEL-Systems dargestellt.

2. Im linken Teil der Darstellung ist die Sicht auf die Ressourcen und Ressourcenklassen des Produktionssystems in der Allgemeinheit abgebildet, wie der Ressourcenbegriff im Laufe der vorliegenden Niederschrift erarbeitet wurde. Neben den physischen Ressourcen, sind weiter oben abstraktere Ressourcen, wie Keller und Halde, bis hin zu abstrakter Rechenfähigkeit (T) und Speicherfähigkeit (S) dargestellt. Ebenso sind die Trennlinien

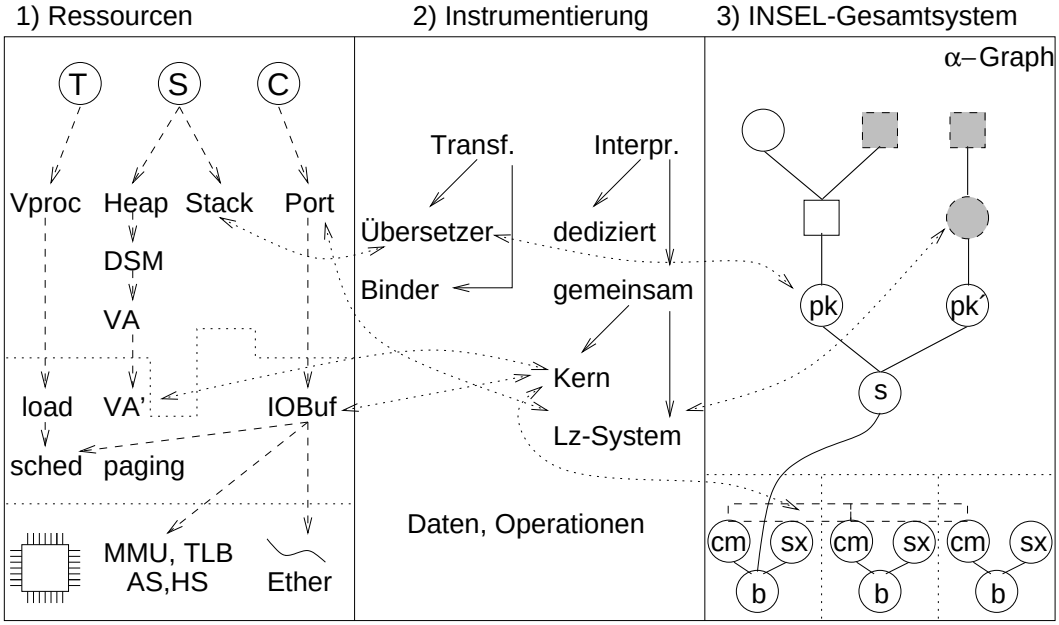


Abbildung 7.3: Sichten auf das INSEL-System

zwischen verteilungsopak und ortsabhängig und weiter zu den physischen Ressourcen angedeutet. Diese Sicht auf das System ähnelt stark gängigen Architekturmodellen.

3. In der Mitte sind mit den Instrumenten Übersetzer, Binder, Kern, etc. die Fertigungseinheiten bzw. administrativen Managementinstanzen des INSEL-Produktionsystems eingezeichnet.

Wird der sprachbasierte Gesamtsystem-Ansatz von MoDiS konsequent verfolgt, so muß es sich, mit Ausnahme der physischen Ressourcen, bei den drei Komponenten des INSEL-Produktionssystems ausschließlich um verschiedene Sichten auf das INSEL-Gesamtsystem, bzw. Abstraktionenräume mit unterschiedlichen Sprachen (S. 5.3.2 f.), handeln. Die Separation dieser Sichten und die Kenntnis ihrer Zusammenhänge ist eine wichtige Voraussetzung, um tatsächlich homogen, integriert, angepaßt und zielgerichtet konstruieren zu können und somit Bruchstellen mit Reibungsverlusten, angefangen von der Sprache, in der das System spezifiziert ist, bis hin zu Managementmaßnahmen, zu vermeiden. Es sei bemerkt, daß weitere Sichtweisen auf das INSEL-System, wie z. B. der $\theta$-Baum der AS-Manager, möglich und bei Bedarf nützlich sind.

Das Einnehmen verschiedener Sichtweisen entspricht der Nutzung unterschiedlicher Klassifikationskriterien zur Bildung der Ressourcenklassen und damit das Wechseln von Abstraktionenräumen. Es dient dem Zweck, von Details zu abstrahieren und diejenigen Eigenschaften von Bindungskomplexen abgestuft zu betrachten, die zur Lösung eines Managementproblems, bzw. für das Fällen von Entscheidungen, von Interesse sind. Das Modell des Produktionssystems verdeutlicht diese Zusammenhänge und liefert gleichzeitig mit der Produktionsspirale und der initialen Konfiguration $PS_{init}$ einen Ansatz für die systematische, inkrementelle Konstruktion eines gesamtheitlichen Systems, das homogen aus Elementen der Menge $\mathcal{V} = \mathcal{H} \cup \mathcal{N}$ mit wohldefinierten Eigenschaften komponiert wird. Es wird behauptet, daß genau diese Zu-

sammenhänge zwischen den separierten Sichten bisher kaum Berücksichtigung finden, wodurch Heterogenität statt Integration, Homogenität und Angepaßtheit, entsteht.

#### 7.1.2.2 Ziele

Die Ziele des V-PK-Management von $PS^I$ sind gemäß der Mehrzweckausrichtung von MoDiS vielfältig. Entsprechend der Analyse der Beweggründe für verteiltes Rechnen in Abschnitt 2.2 müssen divergierende Anforderungen bei der Entscheidungsfindung ggf. mit Prioritäten versehen werden. Unbefangen von der Gewichtung der Ziele bleibt die zentrale Forderung, daß abstrakte INSEL-Programme durch das $PS^I$ auf die physisch verteilte Hardware abgebildet werden. Daneben wird in der vorliegenden Arbeit lediglich quantitative Leistung betrachtet.

**Definition 7.4 (Vollständig realisiert)**

- *Sei $r \in \mathcal{R}_t$ eine Ressource aus dem STK-Systemmodell und $\mathfrak{W}_t(r)$ die Menge aller Konkretisierungspfade von $r$,d. h. Pfade, die in der Dimension $k$ in Richtung der Hardwareressourcen $\mathcal{H}$ verlaufen.*

  *$r$ ist zum Zeitpunkt $t$ vollständig realisiert* $\Leftrightarrow$
  $\mathfrak{W}_t(r) \neq \emptyset \wedge \forall w \in \mathfrak{W}_t(r) : \exists h \in \mathcal{H} : w$ *ist ein Präfix von* $\mathcal{W}_t(r, h)$

- *Sei $(r, s) \in \rho_t$ eine horizontale Bindung im Ausschnitt $STK(\cdot, t, k)$.*
  *$\rho(r, s)$ ist zum Zeitpunkt $t$ vollständig realisiert, wenn $r$ und $s$ vollständig realisiert sind und für die Elemente der Mengen $R \subset \mathcal{H}$ und $S \subset \mathcal{H}, R \cap S = \emptyset$, die an den Enden der Konkretisierungspfade von $r$ und $s$ deren physische Realisierungen repräsentieren, gilt: $\forall t \in R, t \in S : (t, u) \in \bar{\rho}_t^{*^{\mathcal{H}}}$, wobei $\bar{\rho}_t^{*^{\mathcal{H}}}$ die Menge der Bindungen im Ausschnitt $STK(\cdot, t, \mathcal{H})$ bezeichne.*

*Die semantische Korrektheit der Abbildung von abstrakt nach konkret wird in beiden Fällen postuliert.*

Definition 7.4 besagt, daß eine Ressource vollständig realisiert ist, wenn alle ihre Konkretisierungspfade an einer Hardwareressource enden. Abstrakte Bindungen zwischen Ressourcen eines Konkretisierungsniveaus sind realisiert, wenn zwischen den sie realisierenden Hardwareressourcen diese Bezüge ggf. transitiv ebenfalls bestehen. Auf dieser Basis lauten die Aufgaben des V-PK-Managements wie folgt:

**primär:** Betrachtet sei ein Schnappschuß des Systems zum Zeitpunkt $t$. Jede Ressource und jede Bindung aus $STK(\cdot, t, \mathcal{I})$ muß zum Zeitpunkt $t$, automatisiert durch $PS^I$, vollständig realisiert werden.

**sekundär:** Abstrakte $\Delta t$-Übergänge auf dem Konkretisierungsniveau $k \doteq \mathcal{I}$ sollen minimale $\Delta t$-Intervalle auf dem Niveau der physischen Ressourcen $k \doteq \mathcal{H}$ in Anspruch nehmen.

Das abstrakte INSEL-System besteht aus Komponenten und strukturellen Abhängigkeiten. Mit der zweiten Forderung der primären Ziele wird verlangt, daß mit den Beiträgen von $\alpha, \epsilon, \delta, \gamma$, etc. das INSEL-System strukturerhaltend auf die physische Ebene abgebildet wird. Die naheliegende Abbildung der Komponenten auf die Hardware würde für eine korrekte Realisierung des abstrakten Systems alleine nicht genügen. Zum Beispiel muß bereits der Wechsel der Ausführungskomponente $\varphi(x)$ eines Akteures $x$, der in $\sigma$ enthalten ist, durch entsprechende Zusammenhänge zwischen den Hardwareressourcen realisiert werden. Oftmals ist die Realisierung der abstrakten Bindungen sehr viel aufwendiger als die der Ressourcen. Während

der Zusammenhang zwischen den abstrakten Anweisungen $a_i \in L_a(x), x \in X$ durch eine Folge von Maschinenbefehlen im Speicher der Maschine verhältnismäßig einfach realisiert ist, müssen $\kappa$-Abhängigkeiten durch K-Order-Kooperation aufwendig durch Nachrichtenpuffer, Schedulingmaßnahmen, und vieles mehr realisiert werden. Die Kenntnis dieser Gesetzmäßigkeiten ist wiederum eine Voraussetzung, um Information über das in Ausführung befindliche System effizient gewinnen zu können und dabei unnötige Redundanzen zu vermeiden. Da die Abbildung der Resssourcen und Bindungen auf das physische Niveau allerdings oft nicht umkehrbar oder zumindest die Ermittlung des Urbildes ineffizient ist, werden Zusatzmaßnahmen durchgeführt, die es erlauben, die Rücktransformation effizient durchzuführen. Ein einfaches Beispiel hierfür ist die „Debug"-Information, die von gängingen Übersetzern bei Bedarf zusätzlich zur Unterstützung der Fehlersuche und Leistungsanalyse produziert werden kann. Da (redundate) Zusatzmaßnahmen dieser Art zusätzlichen Aufwand für das Management und somit Kosten verursachen, sollte ihre Notwendigkeit stets sorgfältig geprüft werden, was eine exakte Kenntnis der Realisierung der Bindungen voraussetzt. Genau hierin steckt jedoch ein weiteres Defizit der unmodifizierten Übernahme existierender Werkzeuge und Verfahren bei der Konstruktion eines neuen Systems.

**Freiräume des Managements**

Mit dem sekundären Ziel wird ein effizientes Fortschreiten der abstrakten Berechnungen in der Dimension $t$ gefordert, was weitreichende Konsequenzen nach sich zieht.

Anhand des primären Zieles würde es genügen, ein Instrument $A_0$ zu konstruieren, das bei jedem Berechnungsschritt eines Akteures des abstrakten Systems sämtliche Entscheidungen der Produktionsspirale, inklusive der Produktion der weiterer Instrumente, neu trifft. Im nächsten $\Delta t$ Schritt würden alle realisierungsbedingt erzeugten Ressourcen aufgelöst und angepaßt an die neue Situation neu konstruiert werden. Der Vorteil dieser Vorgehensweise wäre, daß sich das System stets genau aus den tatsächlich benötigten Ressourcen und Bindungen zusammensetzen würde.

Es ist davon auszugehen, daß mit $A_0$ die Berechnungen des abstrakten Systems sehr langsam fortschreiten würden, weil das Produzieren der Ressourcen, Bindungen inklusive der Instrumente selbst bereits physische Zeit in Anspruch nimmt. Unter dem sekundären Ziel Effizienz ist $A_0$ daher nicht tragfähig. Es müssen vielmehr Entscheidungen gefunden und Maßnahmen ergriffen werden, die dem inkrementellen Fortschreiten der Akteurberechnungen Rechnung tragen. Es ist anzustreben, daß jeder elementare Berechnungsschritt eines Akteures im Abstrakten nur minimale Maßnahmen zur vollständigen Realisierung nach sich zieht. In erster Linie werden somit Ressourcenklassen und Instrumente benötigt, die ungeachtet einer konkreten Situation von dauerhaftem Nutzen sind. Weiter ist die Minimierung der Generierung und Auflösung von Ressourcen und letztendlich auch deren Bindungen anzustreben. Verfahren, die nach dem Prinzip des *lazy evaluation* operieren oder die Aufbewahrung von Pools temporär nicht benötigter Arbeiterthreads in ALDY [GS96], sind vereinzelte Beispiele dafür, wie dies in der Praxis erfolgen kann. Aus der Systematisierung der inkrementellen Produktion und Bindung von Ressourcen anhand der Rahmenkonzepte der vorliegenden Arbeit sind vermutlich weitere, erhebliche Leistungssteigerungen erzielbar.

**Kosten**

Aus den weiter oben erklärten, verschiedenen Bindungsintensitäten und Bindungsschemen folgt, daß es für die Konstruktion von Pfaden und Zwischenressourcen und somit zur Rea-

lisierung einer abstrakten Ressource unendlich viele Alternativen gibt. Um Entscheidungen zwischen diesen Alternativen treffen zu können, mit denen die quantitativen Forderungen an das System erfüllt werden, muß zumindest eine rudimentäre Bewertung der Alternativen erfolgen. Dabei genügt es im ersten Schritt, auf eine exakte Quantifizierung zu verzichten und stattdessen die Kosten verschiedener Alternativen, bezogen auf Ausführungszeiten, grob abzuschätzen.

**Bemerkung 7.5 ($t$-Kosten)**
*Sei $m$ eine Maßnahme aus der Menge der möglichen Maßnahmen aus dem STK-Systemmodell: Erzeugung bzw. Initiierung oder Auflösung einer Ressource oder Bindung. $K^t(m)$ bezeichne die Anzahl der Einheiten einer abstrakten Zeit, die zur Durchführung der Maßnahme benötigt werden.*

$$K^t(m) = \begin{cases} 1000 : \textit{Bindung einer Netzwerkressource } n \in \mathcal{H} \\ 1 : \textit{sonst} \end{cases}$$

Wenngleich langfristig eine wesentlich differenzierte Beurteilung der Kosten benötigt wird, die z. B. das unterschiedliche Verhalten von Register und Arbeitsspeicher berücksichtigt, so hilft bereits diese rudimentäre Abschätzung bei der Beurteilung einiger, wichtiger Alternativen und soll für diesen Zweck genutzt werden. Neben den extrem hohen Kosten für die Nutzung von Netzwerkressourcen ist in Bemerkung 7.5 implizit auch enthalten, daß die Bindung vorhandener Ressourcen stets der Erzeugung neuer zu bevorzugen ist, da im letzteren Fall die Initiierung von Bindungen zur Nutzung der erzeugten Ressourcen zusätzlich erforderlich ist. Durch die rekursive Definition der Ressourcen in Def. 5.6 ist ferner gesichert, daß bei der Bewertung der Kosten für die Produktion einer Tupelressource die Anzahl ihrer Elemente Berücksichtigung findet.

Im Folgenden wird diese Vorstellung von den Kosten unterschiedlicher Varianten genutzt, um Alternativen qualitativ zu beurteilen, ohne daß $K^t$ für komplexe Verfahren präzise ermittelt wird, da hierfür noch einige offene Fragen zu klären wären, die im Rahmen der Erarbeitung eines Ansatzes zur Systematisierung der Managementkonstruktion in der vorliegenden Arbeit nicht bearbeitet werden.

### 7.1.3 Ein idealisiertes Realisierungsinstrumentarium

Die wesentliche Aufgabe des V-PK-Managements ist die Festlegung der Ressourcenklassen und die Fertigung der zugehörigen Ressourcengeneratoren, die als administrative Managementinstanzen das Management des Produktionssystems erweitern. Weiter oben wurde bereits das sehr einfache und ineffiziente Instrument $A_0$ beschrieben, das als eine erste Möglichkeit der Instrumentierung des Managements gewertet werden kann. In diesem Abschnitt wird die Instrumentierung des V-PK-Managements einen weiteren Schritt konkretisiert, indem der idealisierte, flexible Automat $A_1$ informell beschrieben wird, der INSEL-Systeme mit hoher Effizienz realisiert. Dabei erfolgt auch diese Betrachtung top-down orientiert und nach wie vor losgelöst von einem möglichen Realisierungsaufwand. Die im Abschnitt 7.2 beschriebene Instrumentierung des V-PK-Managements ist von der Vorstellung von $A_1$ geleitet und versucht sich diesem Automat unter den Randbedingungen einer konkreten Implementierung anzunähern.

#### 7.1.3.1 Motivation

Zur Motivation für die Arbeitsweise von $A_1$ dient das in Abbildung 7.4 abgedruckte INSEL-Programm $P_1$. Es beschreibt zwei $\delta^{\mathcal{G}}$-geschachtelte DA-Generatorfamilien, wovon die FS-

```
MACTOR test IS
                                                        – Deklarationsteil von test (1)
  a: INTEGER := 5;                                      – DE-Komponente von test (2)
  TYPE R_T IS RECORD X,Y : INTEGER; END RECORD;         – DE-Generatorfamilie (3)

  FUNCTION foo(x,y : IN INTEGER) RETURN R_T IS          – FS-Order Generatorfamilie (4)
    lokal: INTEGER;
    R : R_T;
  BEGIN                                                 – Anweisungsteil von dummy (5)
    lokal := x+y+a;
    R.X := lokal;  R.Y := lokal+1;
    RETURN R;
  END foo;

   Rec: R_T;                                            – DE-Komponente von test (6)
BEGIN                                                   – Anw.-Teil von test (7)
  rec := foo(1,2);                                      – Erzeugung einer δ-geschachtelten FS-Order (8)
  OUTPUT Rec.X;                                         – Ausgabe des Ergebnisses (9)
  OUTPUT Rec.Y;
END test;
```

Abbildung 7.4: Vollständig spezifizierte Generatorfamilien

Order-Generatorfamilie `foo` in die M-Akteur-Generatorfamilie `test` geschachtelt ist. Wird `test` in das INSEL-System eingebracht und mit einem Generator und anschließender Erzeugung einer Inkarnation zur Ausführung gebracht, so werden ein `test` M-Akteur, eine $\delta$-innere FS-Order, etc. erzeugt und deren Anweisungsteile ausgeführt. Vor der Auflösung des M-Akteures gibt dieser in seinen letzten beiden Anweisungen die Zahlen 8 und 9 als Ergebnis aus. Ansonsten interagiert das von `test` gebildete Teilsystem nicht mit seiner Umwelt. Gemäß der Konzepte und Definitionen der Unvollständigkeit in Kapitel 6, ist die Generatorfamilie `test` vollständig spezifiziert und besitzt bis auf den noch nicht festgelegten Zeitpunkt der Erzeugung eines Generators und einer Inkarnation keine weiteren Freiheitsgrade.

Wünschenswert wäre, daß ein leistungsfähiger Übersetzer bei der Analyse von $P_1$ die Eigenschaft der vollständigen Spezifikation erkennt und $P_1$ zur simplen Ausgabe der Konstanten 8 und 9 faltet. Trotz der vielfältigen Optimierungsstufen moderner Übersetzer (siehe Abb. 2.15) ist das Produkt eines Übersetzungsvorganges weit von dieser naheliegenden Vorstellung entfernt. Es wird ein aufwendiger Restinterpretierer $i_{gen}^{P_1}$ (siehe S. 60) generiert und zur Menge der Managementinstanzen hinzugefügt, der bei seiner Ausführung die nur geringfügig vereinfachten Anweisungen des Programmes inklusive der Erzeugung der Order, etc. mit großem Aufwand unmittelbar durchführt, obwohl in diesem Zuge ausschließlich Invarianten berechnet werden. Dafür gibt es zwei Gründe:

1. In herkömmlichen Sprachen ist (Un-)vollständigkeit, die über Parametrisierung hinausgeht, konzeptionell nicht verankert, weshalb sie bei der Analyse des Übersetzers nicht berücksichtigt werden kann. Dies gilt insbesondere auch für die Eingabe von Daten zum Zeitpunkt der Ausführung.

2. Die Orientierung der Aufgabe des Übersetzers an der mathematisch äquivalenten Funktion $Q \rightarrow Z$ anstelle $Q \rightarrow (\beta \rightarrow D)$ (s. Abschnitt 2.3.3.2) beeinflußt die Integration des Übersetzers in das Gesamtmanagement negativ.

Durch die Betrachtung der Aufgabe des Übersetzers als die Produktion eines effizienten Maschinenprogrammes (2) tritt die eigentliche Aufgabe, das Ergebnis der Programmberechnung zu ermitteln, in den Hintergrund und es erfolgt statt dessen eine lokale Optimierung des Übersetzungsvorganges. Zudem wird eine starre Grenze zwischen Transformation, als Umformung des Programmes innerhalb oder zwischen Sprachen und dem Interpretieren, das unmittelbare Ausführen, gezogen. Die Gesamtaufgabe lautet dahingegen, eine Interpretation des Programmes zu erarbeiten.

Die Trennlinie zwischen Interpretieren und Transformieren verläuft faktisch anders, als dies durch die Begriffe Übersetzer und Interpretierer suggeriert wird. Einige Schritte, die im Übersetzer als Optimierung des Zielcodes betrachtet werden, sind tatsächlich Interpretationen. Ein Beispiel ist das Falten arithmetischer Ausdrücke, wie $(5+3)/2 \mapsto 4$, das bereits sehr früh durchgeführt wird. Ein weiteres Beispiel ist das Einkopieren von Unterprogrammen in die Aufrufer zuzüglich Zuweisung der Parameter etc. (*function inlining*) unter bestimmten Voraussetzungen. In diesem Fall findet eine Vorabinterpretation von Unterprogrammaufrufen statt. Alle Einmal-Interpretationen dieser Art, die bereits früh durch den Übersetzer vorgenommen werden, bringen Effizienzgewinne, da sie später, bei der ggf. wiederholten Ausführung von $i_{gen}^{P_1}$ nicht mehr durchgeführt werden müssen. Diese Beobachtung motiviert folgende, zur Diskussion gestellte, Behauptung.

**Behauptung 7.6**
*Transformieren ist seltenes Interpretieren mit hoher Verbindlichkeit.*

Das Attribut der hohen Verbindlichkeit bringt eine gewisse Trägheit und Dauerhaftigkeit des Transformationsprozesses zum Ausdruck, die auf der Beobachtung beruht, daß Transformieren stets größeren Aufwand verursacht, weil das Transformatorprogramm selbst interpretiert werden muß. Dies lohnt sich nur dann, wenn das Produkt der Transformation in einem gewissen Intervall dauerhafte Gültigkeit besitzt und damit die Kosten für wiederholte, unmittelbare Interpretierungen eingespart werden können.

Insbesondere im verteilten Fall muß das wiederholte Interpretieren von Invarianten vermieden werden. Ein negatives Beispiel für die wiederholte, aufwendige Interpretation von Invarianten über Stellengrenzen hinweg, d. h. mit den hohen Kosten der Bindung an Netzwerkressourcen, waren die Pfadbeschreibungen im Experimentalsystem EVA. Bei jedem Zugriff auf eine DE-Komponente in der Ausführungsumgebung wird eine Pfadbeschreibung, die sich voraussichtlich nicht geändert hat, erneut interpretiert, was zu gravierenden Ineffizienzen im Hinblick auf die Ausführung der abstrakten Akteurberechnungen führt.

Die Auffassung über Interpretierer und Übersetzer, wie sie in der vorliegenden Arbeit vertreten wird, erlaubt es, folgende zentrale Anforderung an ein leistungsfähiges Managementinstrumentarium zu formulieren:

**Postulat 7.7 (Forderung nach flexibler und leistungsfähiger Transformation)**
*Unnötiges, wiederholtes Interpretieren muß durch den Einsatz flexibler und leistungsfähiger Transformatoren vermieden werden!*

Mit Bemerkung 7.6 und einem Instrumentarium, welches Postulat 7.7 erfüllt, würde ein geeigneter Übersetzer das Beispielprogramm $P_1$ einmalig analysieren, interpretieren und dabei

erkennen, daß $P_1$ ausschließlich Invarianten beschreibt und letztendlich den minimalen Restinterpretierer produzieren, der bei seiner Ausführung keine weiteren Berechnungen durchführt, als die Konstanten 8 und 9 auszugeben.

#### 7.1.3.2 Zielvorstellung

Die abschließende Vollständigkeit des zur Motivation verwendeten Programmes $P_1$ ist in dieser Art unrealistisch und nur von entarteten, bzw. trivialen Programmen zu erwarten. In aller Regel ist ein Programm nicht vollständig spezifiziert sondern bedient sich Komponenten aus $\mathcal{N}^u$ (siehe Kap. 6), die erst zu einem späteren Zeitpunkt, ggf. während der Ausführung, durch $r_i \in \mathcal{N}^v$ substituiert werden. Die Forderung nach der Minimierung unnötiger, wiederholter Interpretationen soll nun auch im allgemeinen Fall so weit wie möglich aufrecht erhalten werden. Hierfür muß eine Dynamisierung und Flexibilisierung der Zusammenarbeit zwischen Transformator und Interpretierer erfolgen. Der Modellautomat $A_1$, der das leistet, ist in Abbildung 7.5 dargestellt.

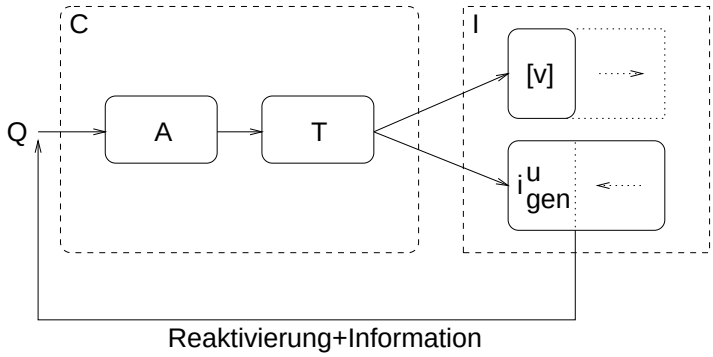


Abbildung 7.5: Modellautomat $A_1$ — inkrementelles Entscheiden

**Definition 7.8 (Modellautomat $A_1$)**
*Der Modellautomat $A_1$ ist definiert als $A_1 := (Q, C, I, \tau)$ mit den Bestandteilen:*

- *Eingabeprogramm $Q$*
- *Übersetzer $C$*
- *Interpretierer $I$; komponiert aus den interpretierten Invarianten $[v]$ und dem Restinterpretierer $i^u_{gen}$ für die Menge der unvollständig spezifizierten Komponenten $u, u \cup v = Q$*
- *Transitionsregel $\tau$*

Die durch $\tau$ festgelegte Funktionsweise des Automaten ist informell wie folgt beschrieben:

1. Der Übersetzer $C$ bearbeitet $Q$
   (a) Das Eingabeprogramm $Q$ wird in $A$ nach der Eigenschaft der vollständigen Spezifikation analysiert.

(b) Im Transformationsschritt $T$ wird, auf Basis der Ergebnisse von $A$, nach vollständig (v) und unvollständig spezifizierten Komponenten (u) differenziert. Für die Menge $u$ wird der Restinterpretierer $i^u_{gen}$ gefertigt, dessen Programm mit den üblichen Übersetzeroptimierungen effizient gestaltet wird. Die Komponenten aus $v$ werden von $C$ einmalig zu $[v]$ interpretiert. $[v]$ und $i^u_{gen}$ bilden gemeinsam den Interpretierer $I$ für das Restprogramm.

2. $C$ beendet seine Arbeit und $I$ setzt die Bearbeitung von $Q$ fort.

   (a) $i^u_{gen}$ interpretiert den noch unvollständigen Teil von $Q$. Wird dabei Information erarbeitet, die es ermöglicht, Elemente aus $u$ nach $v$ zu überführen, so wird diese Information an $C$ weitergereicht und es wird mit Schritt 1 fortgefahren.

   (b) Zusätzlich ist es möglich, daß während der Arbeit von $I$ das unvollständige Programm $Q$ durch GFV, GV oder IV des $Q$-Programmes partiell oder abschließend vervollständigt wird. In diesem Fall wird ebenfalls $I$ angehalten und mit Schritt 1 fortgefahren.

**Analysephase**
Die erste wichtige Fähigkeit von $A_1$ ist die Analyse von $Q$ durch $A$. Freiheitsgrade die von $A$ als unvollständige Spezifikation zu bewerten sind, wurden in Kapitel 6 erläutert. Die Analyse der Freiheitsgrade ist mit den Konzepten von INSEL unerwartet einfach und könnte in $O(|Q|)$ durch Berechnung der transitiven Hülle unvollständig spezifizierter Komponenten etwa wie folgt durchgeführt werden.

1. $u = \emptyset$

2. Suche alle $n_i \in \mathcal{N}^u$, die sich auf Komponenten aus der Umwelt von $Q$ beziehen (Benutzereingaben, Nutzung von Komponenten in der Ausführungsumgebung, `INCOMPLETE`) und füge sie in die Menge $u$ ein; $u = u \cup \{n_i\}$.

3. Folge sukzessive allen Deklarationen und Anweisungen in $L_0$ und $L_a$ und füge diese zu $u$ hinzu, falls in der Deklaration oder Anweisung ein Element aus $u$ benutzt wird.

Der Erfolg der Analyse hängt offensichtlich von der Kenntnis der Menge $\mathcal{N}^u$ ab. Die Kombination aus Sprachbasierung und Gesamtsystem-Ansatz in MoDiS ermöglicht es, $u$ präzise und mit geringem Aufwand zu berechnen. Erst die Verankerung der konzeptionellen Unvollständigkeit und insbesondere der Ein-/Ausgabe auf dem Niveau der Sprache INSEL erlaubt es, Benutzereingaben in $A$ zu identifizieren. Ohne diesem Aspekt der Sprachbasierung würde eine wichtige Grundlage fehlen, um mögliche Interaktionen mit der Umwelt erkennen zu können. Der Gesamtsystemansatz ermöglicht überdies eine vollständige Analyse des gesamten Systems. Bei Verzicht auf die gesamtheitliche Sicht würden sich stets zahlreiche Schnittstellen mit der Umwelt ergeben, die als potentiell unvollständig betrachtet werden müßten. Der Erfolg der Analyse $A$ wäre damit stark limitiert. Ein weiterer Aspekt ist die Kenntnis der Eigenschaftsabhängigkeiten zwischen Generatorfamilien, Generatoren und Inkarnationen gemäß Korollar 6.8. Erst dadurch ist es möglich, daß in $C$ bereits auf Grundlage der Generatorfamilien Entscheidungen getroffen und damit $i^u_{gen}$ schon früh in erheblichem Umfang reduziert werden kann.

**Idealisierte Integration**
Ein zweites wichtiges Merkmal von $A_1$ ist die idealisierte Integration der beiden Instrumente Übersetzer und Interpretierer, die nicht von einer starren Trennung zwischen *Statik* und *Dynamik* behindert wird und in zweierlei Hinsicht zu beobachten ist. Einerseits besteht ein **bidirektionaler Informationsfluß** zwischen $C$ und $I$, der für beide Instanzen von größter Bedeutung ist. Mit $[v]$ und $i^u_{gen}$ fließt Information von $C$ nach $I$. Von $I$ nach $C$ fließt genau die Information zurück, die $C$ zur Verbesserung seiner Entscheidungsgrundlage benötigt. Ferner erfolgen sämtliche **Maßnahmen** aufeinander abgestimmt. Durch das gezielte Produzieren von $i^u_{gen}$ und $[v]$ in jedem $T$-Schritt bereitet $C$ Entscheidungen, die von $I$ zum späteren Zeitpunkt mit minimalem Aufwand getroffen werden können, vor. In der umgekehrten Richtung reaktiviert $I$ den Übersetzer $C$ und veranlaßt diesen zu weiteren Transformationen, genau dann, wenn gesicherte Kenntnis darüber vorliegt, daß der Übersetzer für diesen Zweck besser geeignet ist. Bei der Reaktivierung handelt es sich ebenfalls um eine Maßnahme, die bereits von $C$ in $I$ vorbereitet wird.

**Realisierungsaspekte**
Der hypothetische Automat $A_1$ ist ein interessantes Modell für die Entwicklung eines leistungsfähigen Instrumentariums, daß genau die Maßnahmen vollzieht, die erforderlich sind, um das Ergebnis der Berechnung des durch $Q$ spezifizierten Systems zu ermitteln. Die Realisierung von $A_1$ ist von einigen zusätzlichen Aspekten beeinflußt, die jedoch nicht dazu führen sollten, von der grundsätzlichen Arbeitsweise von $A_1$ abzurücken.

Bei der Modellbildung wurde angenommen, daß die Produktion von $I$ durch $C$ keinen Aufwand verursachen würde und $I$ deshalb in jedem $T$ Schritt vollständig neu konstruiert werden könnte. Diese Vorstellung ist in der Realität nicht tragfähig. Vielmehr muß die Konstruktion von $I$ ebenso **inkrementell** stattfinden, wie die Gewinnung der Information über die Invarianten von $Q$. In Abbildung 7.5 ist aus diesem Grund bereits ein entgegengesetztes Wachsen und Schrumpfen der Anteile $[v]$ bzw. $i^u_{gen}$ von $I$ angedeutet. Nach dem ersten $T$-Schritt von $A_1$ ist zu erwarten, daß der $[v]$-Anteil an $I$ klein gegenüber $i^u_{gen}$ ist. Mit Fortschreiten der Berechnung wächst $[v]$ und der Interpretierer $i^u_{gen}$ wird sukzessive abgebaut. Wird zusätzlich das Hinzukommen sowie die Entnahme von Programmen aus dem System miteinbezogen, so ist ein entgegengerichtetes Pulsieren von $[v]$ und $i^u_{gen}$ statt der vollständigen Neukonstruktion in jedem $T$-Schritt anzustreben. Dieses kontinuierliche Inkrementieren und Dekrementieren der $I$-Beiträge hat weitreichende Konsequenzen. Offensichtlich werden inkrementelle **Bindeverfahren** benötigt, die es erlauben, Komponenten in die bereits bestehenden Anteile $[v]$ und $i^u_{gen}$ zu integrieren und wieder zu entfernen.

Ferner ist es nicht realistisch, daß $T$ auf Komponenten beliebig feiner **Granularität** operiert und deren Transformation nach $v$ inklusive dem inkrementellen Binden sich tatsächlich lohnen würde. Es muß vielmehr ein Repertoire an Granularitäten zwischen $C, I$ und dem inkrementellen Binder von $I$ vereinbart werden, das es erlaubt, in $I$ die aktuelle Situation hinreichend präzise und dennoch effizient zu bewerten und anschließend Entscheidungen über eine mögliche, erneute Transformation zu treffen. So ist zu erwarten, daß erneute Transformation besonders dann von Vorteil ist, wenn umfangreiche Bindungskomplexe behandelt werden können, die häufig genutzt werden und sich klar von anderen, noch unvollständigen Komponenten abgrenzen lassen.

Durch den Aufwand für die Transformation, die Granularitätsfrage und das Binden erhält der zu Beginn idealisiert dargestellte Übergang von $C$ nach $I$ und zurück eine gewisse Trägheit,

die nach einer weiteren zusätzlichen **Flexibilisierung** der Arbeitsweise von $A_1$ verlangt. In $A$ wurde Information bisher lediglich nach **Invarianten** klassifiziert. Im Rahmen der Flexibilisierung ist es von Nutzen, zusätzlich spekulatives Wissen über das zukünftige Verhalten, bzw. **Quasi-Invarianten**, auf Grundlage von Empirie oder partieller Information über die vergröberten und partiell unvollständigen Granule einzusetzen. In $T$ können dann auf Basis dieser Spekulationen Alternativen vorbereitet und als zusätzliche Menge $[v^s]$ dem Interpretierer zur Verfügung gestellt werden. Dieser ist später in der Lage, über die Gültigkeit der Quasi-Invarianten selbst zu urteilen und mit Hilfe seines inkrementellen Binders effizient Komponenten aus $i^u_{gen}$ durch Elemente aus $[v^s]$ zu ersetzen. Flexibilisierung ist somit bereits auf diesem abstrakten Niveau ein wichtiges Mittel zur Leistungssteigerung. Im nächsten Abschnitt werden Möglichkeiten der Flexibilisierung systematisch aus den Rahmenkonzepten für Ressourcen und Bindungen abgeleitet.

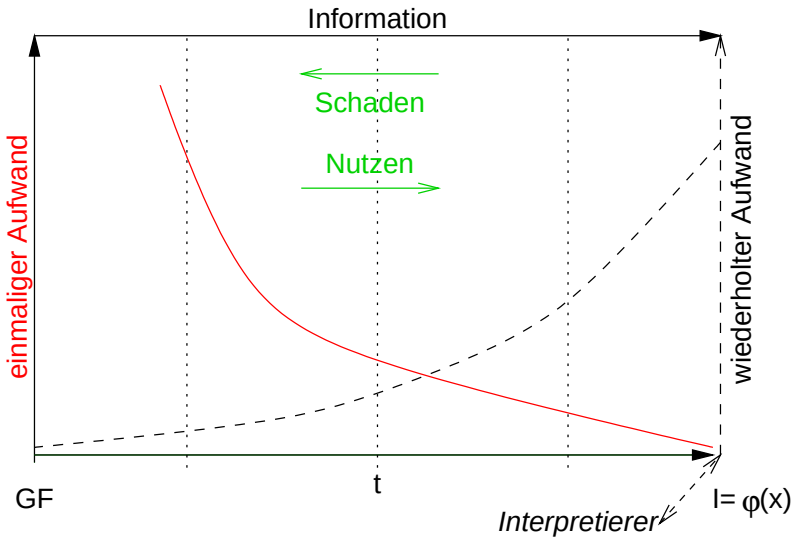


Abbildung 7.6: Einsatz abgestufter Transformatoren und Interpretierer

Durch die Hinzunahme des inkrementellen Binders und weitere flexibilisierende Maßnahmen wird die Anzahl der Instrumente erweitert und eine feinere Abstufung des Realisierungsspektrums erreicht. Abbildung 7.6 veranschaulicht einige der Zusammenhänge zwischen Information und der Wirkung von Entscheidungen mittels abgestuftem Transformieren und Interpretieren. Im Extremfall des reinen Interpretierens werden alle Entscheidungen für eine Inkarnation $x$ mit hohem Aufwand erst dann getroffen, wenn es im Zuge seiner Ausführung unbedingt erforderlich ist. Können Realisierungentscheidungen schon sehr früh auf Grundlage, evtl. auf Basis der Generatorfamilie, getroffen werden, so bedeutet dies einmalig hohen Aufwand, der bei wiederholter Nutzung gerechtfertigt ist. Die diesbezüglichen Möglichkeiten sind allerdings begrenzt, da die Information über die Begebenheiten mit dem Fortschreiten der Berechnungen zunimmt. Mit der Präzision der Information nimmt auch der Nutzen der Managementmaßnahmen zu, während in der anderen Richtung die Gefahr eines Schadens durch Fehlentscheidungen zunimmt. Statt zwei Instrumenten, sind die Handlungsbereiche von vier Instrumenten eingetragen, die größere Variation bezüglich dem Zeitpunkt und der Verbindlichkeit von Entscheidungen zulassen, so daß die Wahl für eine Vorgehensweise nicht auf die äußeren Extrembereiche mit stark unterschiedlichen Eigenschaften beschränkt ist.

Zum Abschluß der Betrachtungen des hypothetischen Instrumentariums $A_1$ sei bemerkt, daß die Arbeitsweise von $A_1$ Ähnlichkeiten zu dynamischer Programmierung im Allgemeinen sowie partielle Evaluation (PE) im Speziellen aufweist. Diese Assoziation ist berechtigt und erwünscht. Allerdings bestehen wichtige Unterschiede. Während in PE die inkrementelle Auswertung eines Programmes auf genau einem Abstraktionsniveau betrieben wird, zielt $A_1$ darauf ab, vertikal sämtliche Konkretisierungsniveaus von abstrakt nach konkret inklusive aller Zwischenressourcen und Bindungen konsequent nach diesem Schema inkrementell zu konstruieren, so daß von einer partiellen Konkretisierung (PK) gesprochen werden kann. Ein einfaches Beispiel wäre eine drastische Vereinfachung der Kommunikation zwischen zwei Akteuren $a$ und $b$ , sobald bekannt ist, daß sich $a$ und $b$ auf einer Stelle befinden. In diesem Fall kann auf aufwendige Kerneinsprünge zum Versenden von Nachrichten verzichtet und wesentlich effizienter über den gemeinsamen lokalen Speicher kommuniziert werden. Dies entspricht dem angestrebten Reduzieren von $i^u_{gen}$ je nach Kenntnisstand über Invarianten und Quasi-Invarianten!

### 7.1.4 Flexibilisierung

Mit $A_1$ konnte die Notwendigkeit für flexibilisierende Maßnahmen, losgelöst von Details, allgemein aufgezeigt werden. Inhärente Gründe sind die inkrementelle Präzisierung des Wissens über die Begebenheiten in Form von Invarianten und Quasi-Invarianten mit dem Fortschreiten der Berechnungen sowie die großen Differenzen zwischen den Kosten für unterschiedliche Realisierungsvarianten. Beiden Aspekten muß durch ein fein abgestuftes Realisierungsrepertoire Rechnung getragen werden. Ferner wurde bei der Diskussion der Entscheidungsfindung des Managements darauf hingewiesen, daß die Existenz und Kenntnis von Freiheitsgraden eine Grundvoraussetzung ist, um adäquate Entscheidungen finden zu können. In diesem Abschnitt wird auf Basis der Konzepte aus Kapitel 5 das Spektrum der grundsätzlichen Möglichkeiten für flexibilisierende Maßnahmen aufgespannt. Aus diesen Alternativen kann und muß das Management, angepaßt an die jeweilige Anforderung und anhand geeigneter Kriterien eine strategisch günstige Auswahl treffen. Auf triviale Flexibilität, wie sie z. B. in der hypothetischen und ineffizienten Maschine $A_0$ durch die vollständige Neuproduktion aller Ressourcen bei jedem $\Delta t$-Übergang betrieben wurde, wird hier nicht weiter eingegangen. Stattdessen werden einige wichtige Möglichkeiten diskutiert, die dabei helfen sollen, eine Balance zwischen den konträren Zielen, Langlebigkeit und Verbindlichkeit einerseits und Freiheitsgrade andererseits, zu erzielen. Mit ersterem wird angestrebt, Ressourcen und Bindungen mit hoher Verbindlichkeit zu produzieren, um die Kosten für deren Produktion und Auflösung zu minimieren. Konträr dazu muß dennoch ein adäquater Entscheidungsspielraum erhalten bleiben, damit das Management auf Veränderungen, die mit der Dynamik der Berechnungen einhergehen, effizient reagieren kann. Der wesentliche Faktor für die Balance zwischen effizienter Langlebigkeit und Flexibilität ist die Kompositionalität der Ressourcen.

**Bemerkung 7.9 (Kompositionalität)**
*Sei $r \subset \mathcal{R}$ eine Ressource oder ein Bindungskomplex. Die Kompositionalität von $r$ ist ein Maß für die Nutzbarkeit von $r$ und ergibt sich aus dem Verhältnis zwischen den Möglichkeiten $r$ zu binden und den dafür erforderlichen Veränderungen an $r$.*

#### 7.1.4.1 Qualifikationen der Referenzressourcen

Bereits die Festlegung der Qualifikationen der Referenzressourcen besitzt erhebliche Auswirkungen auf deren Kompositionalität. Einerseits ist eine möglichst eingeschränkte Qualifizierung anzustreben, um eine Strukturierung der (potentiellen) Abhängigkeiten erzielen und damit die Mächtigkeit der Menge an Entscheidungsalternativen für das Binden der Ressourcen gezielt einzuschränken. Andererseits führt eine stark limitierte Qualifikation zu starren Einschränkungen, die bei konkretem Bedarf nach mehr Flexibilität durch die Produktion zusätzlicher Ressourcen und Bindungen kurz- bis mittelfristig nicht überwunden werden können. Hierfür muß die Planungsinstanz $M$ erst kreativ neue Ressourcenklassen planen.

**Beispiel 7.10** *Im Experimentalsystem AdaM wurde entschieden, daß benannte DE-Komponenten $y_i \in Y_t$ ausschließlich in Kellern des virtuellen Speichers realisiert werden. D. h., die Qualifikation der entsprechenden Referenz-Zwischenressourcen sieht lediglich die Bindung von Elementen der Klasse N-DE-Komponente an Elementen der Klasse Kellerspeicher vor. Dadurch ist ausgeschlossen, daß eine N-DE-Komponente in einem Register realisiert wird, was in vielen Fällen wesentlich effizienter wäre. Die Hinzunahme der Klasse Maschinenregister zur Qualifikation einer Ressourcenklasse eines Zwischenniveaus wäre hier offensichtlich nützlich.*

**Beispiel 7.11** *Ein Negativbeispiel für flexible Qualifikation ist die Ressourcenklasse Datei in herkömmlichen Betriebssystemen. Dateien sind für viele Zwecke nutzbar und können an vielfältige unterschiedliche Dinge, wie ein Textdokument, ein ausführbares Programm oder ein Gerät gebunden sein. Diese Flexibilität ist so weitreichend, daß die wichtigen Eigenschaften einer Datei, Struktur, etc., von außen unbekannt sind und von potentiellen Nutzern bei jeder Nutzung erneut erarbeitet und selbsttätig gewährleistet werden muß.*

#### 7.1.4.2 Dimensionen $s$ und $k$

Die Schemata für Flexibilisierungen in den Dimensionen $s$ und $k$ ähneln einander und werden in diesem Abschnitt gemeinsam behandelt.

**Raum-Granularität**
Bei dem Automaten $A_1$ wurde darauf hingewiesen, daß die Granularität der behandelten Ressourcen einen entscheidenden Einfluß besitzt. Stets atomare Ressourcen zu betrachten ist offenkundig unzweckmäßig, da dies bedeuten würde, jede Bindungsentscheidung separat zu treffen. Können größere, dauerhaft (s. Zeit-Granularität, unten) kompositionsfähige Granule gefertigt werden, so wird die Zahl der zu bearbeitenden Ressourcen und Bindungen erheblich reduziert. Wird die Raum-Granularität zu groß gewählt, so wird es allerdings schwierig, Managemententscheidungen gezielt zu treffen. Im Extremfall würde dies wieder dazu führen, ganze Realisierungspfade in jedem Schritt vollständig neu zu konstruieren.

**Definition 7.12 (Raum-Granularität)**
*Sei $r \subset \mathcal{R}$ eine Ressource oder ein Bindungskomplex. Die Raum-Granularität $G^s(r)$ ist definiert als*

$$G^s(r) = \begin{cases} 1 : r \in \mathcal{H} \cup \mathcal{N} \\ 1 : r \text{ ist eine Referenzressource} \\ n : r \text{ ist eine } n\text{-Tupelressource} \\ \sum_{r_i \in r} G^s(r_i) : r \text{ ist ein Bindungskomplex} \end{cases}$$

**Beispiel 7.13** *Paradebeispiele für die Granularitätsproblematik finden sich im Bereich DSM. Seitenbasierte DSM (SVM) leiden unter der zu großen Raumgranularität der Einheiten; siehe* false sharing. *Objektbasierte Verfahren – DSOM – leiden andererseits oftmals unter der zu kleinen Granularität individueller Datenobjekte, deren separate Verwaltung erheblichen Mehraufwand verursacht. Die Frage nach einer geeigneten Raumgranularität ist bislang unbeantwortet, was in erster Linie darauf zurückzuführen ist, daß keine systematische Diskussion, losgelöst von Randbedingungen existierender Werkzeuge und Hardware, stattfindet.*

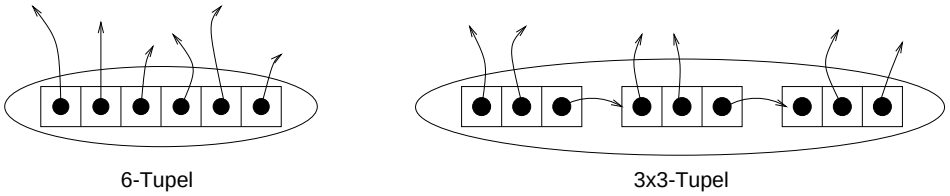


Abbildung 7.7: Abstufung der Raum-Granularität von Bindungskomplexen

Die Definition der Raumgranularität enthält selbst bereits einen Freiheitsgrad für die Wahl der Granule. Einerseits können Vergröberungen durch Vergrößerung der Tupel stattfinden, andererseits können stattdessen auch kleinere Tupel über dauerhafte Bindungen zu größeren Granulen gebunden werden. In Abbildung 7.7 sind die beiden Möglichkeiten als zwei Bindungskomplexe mit äquivalenten Nutzungseigenschaften gegenübergestellt; links ein Sechstupel, rechts drei gebundene Tripel. Die Wahl kleinerer Tupel flexibilisiert die Bildung der Bindungskomplexe. Der Nachteil, der dadurch entsteht, sind höhere Kosten für die größere Anzahl der elementaren Ressourcen und Bindungen. In der Abbildung wären die Kosten gemäß Bemerkung 7.5 links $K^t(6) = 12$, rechts $K^t(3 \times 3) = 18$. Neben den niedrigeren Kosten für die Tupel-Variante ist genauso wie bei der Qualifikation der Referenzen zu berücksichtigen, daß die Tupel-Größe eine Klasseneigenschaft ist, die kurz- bis mittelfristig nicht mehr anpassungsfähig ist.

**Beispiel 7.14** *Von der Hardware unterstützte, virtuelle Speicherseiten sind ein Beispiel für Vergröberung der Granule über die Tupelgröße. Über die Nutzung ganzer Seiten kann dadurch sehr effizient entschieden werden. Differenzierungen innerhalb einer Seite sind nicht möglich, bzw. müssen separat organisiert werden.*
*Verkettete Listen jeglicher Art, wie z. B. die Organisation von Z-Komponenten auf der Halde, bilden Bindungskomplexe aus kleineren Tupel, die individuell effizient handhabbar sind. Operationen, die die ganze Halde betreffen, wie z. B. Verlagerung auf den Hintergrundspeicher, sind demgegenüber aufwendig.*

**Schnittstellen**
Die Granularitätsfrage ist zudem abhängig von den Schnittstellen der Granule. In der Definition der Raum-Granularität wurden beliebige Bindungskomplexe betrachtet. Tatsächlich ist die Bestimmung geeigneter Granule wesentlich davon abhängig, inwiefern Ressourcen mit **abgrenzbaren Abhängigkeiten** zum einen gefunden und zum anderen gezielt produziert werden können. Aus dem Gesamtsystem-Ansatz von MoDiS ergibt sich, daß jede Komponente des abstrakten Systems Abhängigkeiten zu anderen Komponenten des Systems aufweist. Durch geeignete Festlegung von Zwischenressourcenklassen muß das Management aus diesem

Geflecht Substrukturen isolieren, die nach außen gut abgrenzbar sind und sich als Realisierungseinheiten lohnen. Im Beispiel des Automaten $A_1$ ist dies die entscheidende Frage für die Beurteilung der Zweckmäßigkeit eines erneuten Überganges von $I$ zu $C$. Dieser ist dann sinnvoll, wenn durch neue Information ein abgrenzbarer Bindungskomplex neu realisiert und durch inkrementelles Binden von $u$ nach $v$ überführt werden kann.

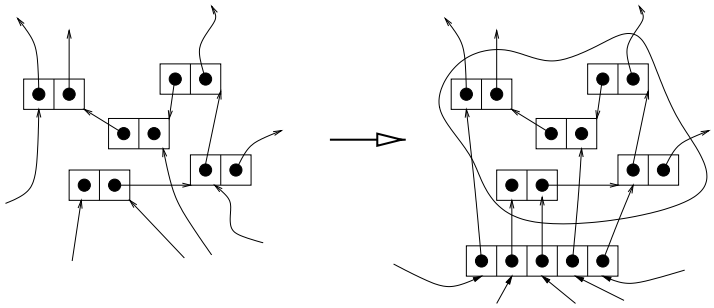

Abbildung 7.8: Äußere Abhängigkeiten von Bindungskomplexen

Mit den Bindungsintensitäten aus Abschnitt 5.2.2 wurde die Abstufung der inneren Abhängigkeiten erklärt. Für die Abgrenzbarkeit eines Bindungskomplexes sind nun seine äußeren Abhängigkeiten entscheidend.
In großen Komplexen können sehr effiziente, unmittelbare Bindungen festgelegt werden. Die Anpassung von Substrukturen wird durch die vielen aufzulösenden und zu substituierenden Bindungen allerdings sehr aufwendig. In Folge ist es oftmals zweckmäßig, die Abhängigkeiten über Vermittlerkomponenten zu strukturieren. Auf diese Weise werden besser abgrenzbare Komplexe gebildet, für deren Ersetzung lediglich eine Tupel-Ressource – der Vermittler – bearbeitet werden muß. Offensichtlich verursacht die Strukturierung durch Vermittler selbst Mehraufwand für die zusätzlichen Ressourcen und Bindungen, so daß auch hier nicht beliebig substrukturiert werden kann, sondern eine Balance zwischen der **Integration** und **Separation** in der Dimension Raum gefunden werden muß. In Abbildung 7.8 ist die Separation eines Komplexes durch Bündelung seiner äußeren Abhängigkeiten mittels einer Vermittlerkomponente dargestellt.

**Beispiel 7.15** *Beispiele für die Abstufung der Abhängigkeiten und die Schwierigkeiten, eine Balance zu finden, wurden insbesondere im Abschnitt 2.3.2.3 mit Monolithen, $\mu$-Kernen und Schichtenansätzen diskutiert.*
*Ein weiteres Beispiel ist die Bestimmung der Akteursphären als wesentliche Einheiten des Managements in MoDiS. Dabei wird davon ausgegangen, daß die Abhängigkeiten innerhalb der Akteur-Berechnung wesentlich intensiver als nach außen sind und somit diese Separation, die ebenfalls durch zahlreiche Vermittler realisiert werden muß, gerechtfertigt ist.*

**Bereitstellung von Alternativen**
Die Festlegungen der Raum-Granularität und der äußeren Schnittstellen von Bindungskomplexen bilden die Grundlage für die gezielte Bereitstellung von Realisierungsalternativen, das ein weiteres mächtiges Konzept zur Flexibiliserung der Produktion der Pfade zwischen $\mathcal{N}$ und $\mathcal{H}$ darstellt.

**Definition 7.16 (Realisierungsalternative)**
*Seien $r, r' \subset \mathcal{R}$ zwei Bindungskomplexe.*
*$r$ und $r'$ heißen Realisierungsalternativen, wenn sie die äquivalente Nutzungseigenschaften besitzen. Ihre Realisierungseigenschaften – Pfade nach $\mathcal{H}$ – unterscheiden sich.*

Realisierierungsalternativen erweitern das Konzept der Ressourcenklassen und Ressourcen auf komponierte Einheiten, deren innere - und Realisierungseigenschaften sehr unterschiedlich ausgeprägt sein können. Durch Bereitstellung dieser Alternativen wird die Ressourcenadministration erheblich flexibilisiert, da Teilstücke der Pfade von $\mathcal{N}$ nach $\mathcal{H}$ effizient, durch das Wechseln weniger Bindungen, neu festgelegt werden können. Der Grad dieser Flexibilität hängt stark davon ab, inwiefern abgrenzbare Komplexe festgelegt und die Instrumente zur Produktion der Alternativen vorbereitet wurden.

**Beispiel 7.17** *Kacheln des Arbeitsspeichers sind flexibel einsetzbare Alternativen zur Realisierung des virtuellen Speichers.*
*Ein komplexeres Beispiel im Kontext verteilter Systeme ist die Realisierung von Zugriffen auf entfernte Daten mittels dynamischer Replikation, Migration und RPC [Win96b].*

#### 7.1.4.3 Variabilität in der Dimension $t$

Die Kompositionalität jeder Ressource bzw. jedes Bindungskomplexes ist neben den Bezügen im Raum von der Dauer ihrer Existenz abhängig. Zur Abstufung des zeitlichen Verhaltens sind gemäß Kapitel 5 zwei Konzepte vorhanden. Erstens die Festlegung der Lebenszeiten $\Lambda$ der Ressourcen und zweitens die Wahl der Verbindlichkeitsintervalle (VI) $\Phi$. Aus der Abstufung dieser Intervalle ergibt sich nochmals großes Potential für die Flexibilisierung der Realisierung von Ressourcen und Bindungen auf abstraktem Niveau. Auf die Betrachtung der Auflösung und Neuproduktion einzelner Ressourcen, die selbstverständlich ein Mittel der Flexibilisierung ist (siehe Auflösung von Ressourcen in $u$ und Generierung neuer in $[v]$ im Automaten $A_1$), wird im Folgenden verzichtet und statt dessen zwei besonders wichtige Aspekte der VI betrachtet: Revisionen und die inkrementelle Konstruktion der Realisierungspfade.

**Definition 7.18 (Zeit-Granularität)**
*Sei $r \subset \mathcal{R}$ ein Bindungskomplex.*
*Die Zeit-Granularität $G^t(r)$ ist das minimale Zeitintervall, in dem $r$ unverändert Bestand hat.*

Die Zeit-Granularität subsumiert somit Lebenszeit- und Verbindlichkeitsintervalle und definiert eine Zeitspanne, in welcher in dem Komplex weder Ressourcen noch Bindungen hinzugefügt oder aufgelöst werden. Wieder sind einerseits Vergröberungen anzustreben, um die Notwendigkeit für Neuentscheidungen zu minimieren. Andererseits wird Flexibilität in der Zeit benötigt, um auf Veränderungen reagieren zu können, wofür die VI kurz gewählt werden sollten.

**Beispiel 7.19** *Durch die als dauerhaft – $[t, \infty]$ – festgelegte Verbindlichkeit der Produkte eines herkömmlichen, statischen Binders, ist es nur sehr schwer möglich, nachträglich Änderungen vorzunehmen, um z. B. ein Modul separat auszutauschen.*
*Ein wichtiges Beispiel für quantitative Verluste durch eine zu groß gewählte Zeit-Granularität im Umfeld paralleler Systeme, ist das unnötig lange Aufrechterhalten von Sperren, wodurch die Verfügbarkeit von Ressourcen reduziert wird.*

#### 7.1.4.4 Revisionen

Zeitgranularitäten lassen sich nach dem simplen Kriterium, ob die Grenzen des Intervalles fest vereinbart oder variabel gestaltet wurden, in die vier Klassen einteilen, die in Tabelle 7.1 für ein Intervall $G^t(x) = [a, b], x \subset \mathcal{R}$ angegeben sind. „Fix“ bedeutet, es wurde eine Vereinbarung getroffen, „variabel“ heißt, es wurde noch noch keine Vereinbarung getroffen.

| | a | b |
|---|---|---|
| a | fix | variabel |
| b | variabel | fix |

Tabelle 7.1: Charakteristische Verbindlichkeitsintervalle

Verläßliches Wissen über die Zeit-Granularität ist wichtig, um Entscheidungen treffen zu können. Ist nichts über die Dauer der Existenz von Ressourcen oder Teilstrecken von Pfaden zwischen $\mathcal{N}$ und $\mathcal{H}$ bekannt, so kann auch nur sehr unbefriedigend damit operiert werden. D. h. grundsätzlich sind Intervalle der Klasse $[fix, fix]$ anzustreben. Es gibt allerdings Situationen, die ein frühzeitiges Auflösen der Bindung vor ihrem vereinbarten Ende erforderlich machen. Dies ist insbesondere dann erforderlich, wenn die Planungsinstanz des Managements $M$ weitreichende Veränderungen an dem arbeitendem System vornehmen muß und ordnet sich in die Sicherung der Langlebigkeit des Systems ein. Das frühzeitige Auflösen eines Komplexes, einer Bindung oder Ressource heißt **Revision** und dient in erster Linie der Gewährleistung der **Evolutionsfähigkeit** des Systems, wie sie in MoDiS angestrebt wird.

Aus der oben diskutierten Raum-Granularität und den Schnittstellen des zu revidierenden Komplexes ergibt sich der Aufwand, der für die Revision aufzubringen ist und bei intensiven Abhängigkeiten erheblich sein kann.

#### 7.1.4.5 Inkrementelle Pfadkonstruktion

Eine zweite herausragende Möglichkeit der Nutzung von Freiheitsgraden in der Dimension Zeit, die in dieser Arbeit bereits verschiedentlich phänomenologisch beobachtet wurde, kann nun mit Hilfe der Rahmenkonzepte präzise klassifiziert werden. In Abbildung 7.9 ist die Kon-

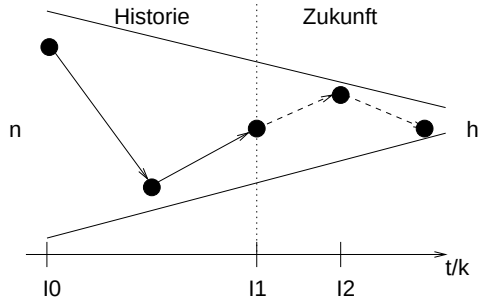


Abbildung 7.9: Inkrementelle Konstruktion der Pfade

struktion eines Realisierungspfades $\mathcal{W}(n, h), n \in \mathcal{N}, h \in \mathcal{H}$ durch die drei Instrumente $I_0, I_1$ und $I_2$ dargestellt. Der Pfad wird Schritt für Schritt durch die Instrumente von abstrakt nach konkret mit wachsender Präzisierung der Information über die Zeit $t$ in seinen Freiheitsgraden

eingeschränkt und fertiggestellt. Genau diese Art der Konstruktion des Realisierungspfades ist die Abstufung und **Angepaßtheit der Maßnahmen der Instrumente**, die in der vorliegenden Arbeit wiederholt gefordert wurde. Sie wird durch die gezielte Festlegung der Verbindlichkeitsintervalle und die Kompositionalität der Ressourcen und Komplexe auf dem Pfad erzielt.

Bei der in der Praxis in aller Regel mangelhaften Angepaßtheit werden von $I_1$ nicht nur neue Teilstücke des Pfades produziert, sondern bereits bestehende Elemente aufgelöst, redundant neu produziert, das Anreichern der Information mißachtet, et cetera.

**Beispiel 7.20** *Versuche Realisierungspfade inkrementell zu konstruieren wurden u. a. mit der Run-Time Code Generation und der Code Generation on the Fly in Abschnitt 2.3.3.2 diskutiert. Negative Gegenbeispiele für Angepaßtheit sind die aufwendige Analysen in einigen SVM-Systemen, die nachträglich versuchen, die Größe von Datenobjekten zu ermitteln. Im Grunde wird diese Information bereits im Rahmen der gewöhnlichen Übersetzeranalysen erarbeitet, dort anschließend aber wieder vernichtet. Gleiches gilt für die häufig doppelte Verwaltung von Seitentabellen in SVM-Systemen im Benutzer- und dem Kern-Modus.*

Verallgemeinert kann nun das idealisierte Spektrum an Möglichkeiten für die Zusammenarbeit der Instrumente eines Managements, das mit dem Automaten $A_1$ lediglich angedeutet wurde, präzise klassifiziert werden. Von jedem Instrument sollte die Entscheidung über den Verlauf eines Pfades **zurückgestellt**, **eingeschränkt**, **vorbereitet** oder **vollständig getroffen** werden können.

#### 7.1.4.6 Anmerkungen

In diesem Abschnitt wurde ein weites Feld an Möglichkeiten zur Erhaltung, Schaffung und Nutzung von Freiheitsgraden skizziert, aus dem die strategische Planung des Managements wählen kann und muß. Aus den elementaren Flexibilisierungsprinzipien, die hier erläutert wurden, werden in realen Systemen Spektren komplexer Verfahren mit unterschiedlichen Charakteristika konstruiert. Die Klasse der *lazy* und konträr *prefetching* Verfahren läßt sich gut in das zuletzt erläuterte Konzept der inkrementellen Pfadkonstruktion einordnen. Gleiches gilt für die prinzipielle Wahl zwischen *Fehlervermeidung* oder *-behebung*. Mit der Kenntnis, daß die Kosten für Revisionen von den Abhängigkeiten des Bindungskomplexes abhängen, kann die Wahl eines fehlerbehebenden Verfahrens fundiert beurteilt werden. Eine weitere interessante Aufgabe wäre die hierarchische Klassifikation weiterer Verfahren auf Basis der hier eingeführten Schemata, wie die Bündelung von Ressourcen in stellengebundenen Pools kontra deren stellenunabhängige Verwaltung.

Im nächsten Schritt der Konkretisierung des V-PK-Managements kommt der gewichtige Faktor Aufwand hinzu, der es verbietet, alle der hier skizzierten Aspekte im gebührenden Umfang zu berücksichtigen. Dennoch liefern diese Überlegungen Erkenntnisse, die in die Implementierung einfließen. Es wird in aller erster Linie angestrebt, die im Abstrakten vorhandenen Freiheitsgrade bei der Implementierung trotz des Aufwands weitestgehend zu erhalten. Mit den Argumenten pro und kontra feiner Raum-/Zeit-Granularität ist ebenso klar, daß es sich nicht um beliebige Flexibilität handeln darf. Vielmehr sind Raster charakteristischer Granularitäten von fein bis grob erfolgversprechend, die es erlauben, abgestufte Informationsgewinnung und ebensolche Maßnahmen der Instrumente zu realisieren. Dabei ist es zunächst ausreichend, einige wenige Stufen dieser Rasterung zu realisieren.

Zuletzt sei auf den Zusammenhang zwischen der Flexibilität hier und der inkrementellen Systemkonstruktion aus Kapitel 6 hingewiesen, der sich aus dem Gesamtsystemansatz und den Erläuterungen zur Produktionsspirale ergibt. Konzeptionelle Unvollständigkeit liefert erst die Gelegenheit, die Instrumente des Managementsystems über die Zeit flexibel anzupassen.

## 7.2 Realisierung der AS-Manager

Mit den methodischen Grundlagen, die angefangen von Kapitel 5 bis zu dieser Stelle erklärt wurden, konnten aufschlußreiche Erkenntnisse über die Konstruktion von Managementsystemen gesammelt werden. Die erarbeiteten Konzepte und das Wissen über die vielfältigen Möglichkeiten und Abhängigkeiten machen es jetzt möglich, das V-PK-Management so zu realisieren, daß substantielle Gewinne gegenüber vergleichbaren Ansätzen erzielbar sind. Dabei ist in Erinnerung zu rufen, daß die wesentlichen Schwachpunkte anderer Ansätze entweder das unbefriedigende Abstraktionsniveau auf Nutzungsebene oder schwache Leistung insbesondere aufgrund eines hohen lokalen Mehraufwandes waren. Mit der Sprache INSEL und der Realisierung des V-PK-Managements gemäß der Leitlinien dieses Kapitels, besteht nun die Chance, sowohl qualitativ (Sprache INSEL) als auch quantitativ (V-PK-Management) hochwertiges paralleles Rechnen, auf Basis physisch verteilter NOW-Konfigurationen zu verwirklichen.

Im Rahmen der vorliegenden Arbeit wurden entsprechende Implementierungsarbeiten an einem geeigneten Instrumentarium im erheblichen Umfang bereits durchgeführt. Gegenstand dieses Abschnittes ist die Skizzierung dieser konkreten Realisierungsarbeiten. Bei der Synthese der Konzepte im Teil II dieser Arbeit wurde bislang der kreative Spielraum des top-down orientierten, gesamtheitlichen MoDiS-Ansatzes genutzt. Möglich war das, indem der bereits auf Seite 9 diskutierte dritte fundamentale Einflußfaktor *Aufwand* (neben Methodik und Kreativität) bis hierhin ausgeklammert wurde. Bei der Verwirklichung der erarbeiteten Konzepte muß nun eine Antwort oder zumindest ein Lösungsansatz für die Frage gefunden werden, wie die idealisiert konzipierte Architektur mit vertretbarem Aufwand in die Realität umgesetzt werden kann. Offensichtlich zwingt Aufwand zu Kompromissen, so daß in einem ersten Schritt die vollständige Realisierung der Maschine $A_1$ nicht ernsthaft erwartet werden kann. Allerdings soll und kann eine Annäherung an diese erfolgen.

Vor der Realisierung des V-PK-Managements ist noch eine weitere wichtige Frage zu klären. In Kapitel 3 wurde das Konzept der AS-Manager eingeführt, welche die Aufgabe besitzen, sämtliche Ressourcen des Systems angepaßt an die Anforderungen der durch $man^{-1}(x)$ (Def. 3.12) assoziierten Akteure zu befriedigen. Zu klären ist, wie sich dieser Ansatz in das oben eingeführte Modell des Produktionssystems, dessen Management und Administration einordnet. Im Folgenden wird zunächst diese architekturelle Frage beantwortet, bevor ab Abschnitt 7.2.2 die konkreten Realisierungsarbeiten an dem V-PK-Management erläutert werden.

### 7.2.1 Einordnung der Manager in das Modell des Produktionssystems

Die Antwort auf die zuletzt aufgeworfene Frage wurde allgemein in Abschnitt 5.3.2 auf Seite 140 bei der Erläuterung der Abstraktionenräume gegeben. Die Aufteilung des Managements in die multiplen, autonom agierenden AS-Manager bedient sich anderer Kriterien zur Bildung der Ressourcenklassen des Systems als die Betrachtung der Fertigungseinrichtungen des Produktionssystems. Die gebildeten Klassen sind weder disjunkt noch identisch.

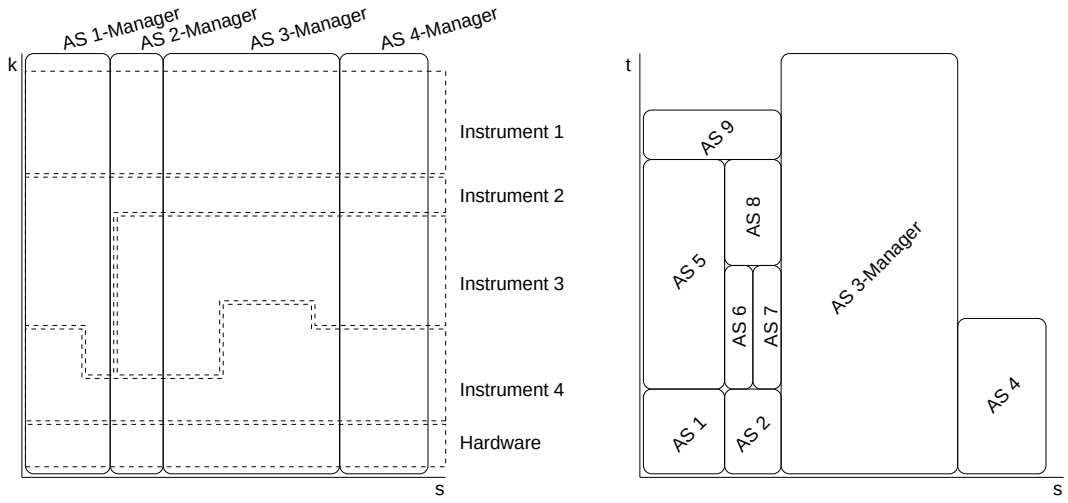


Abbildung 7.10: Einordnung der Instrumentierung in den Manageransatz

Die $\theta$-Struktur der Manager orientiert sich an den Akteursphären und strukturiert die Managementaufgabe in die Dimensionen $s$ und $t$. In der Dimension $k$ hat jeder Manager alle Anforderungen zu erfüllen und betreibt damit vertikale Integration[1]. Mit der Instrumentierung erfolgt eine Strukturierung des Managements in die Dimensionen $k$ und $t$, wohingegen in diesem Fall über $s$ (horizontal) integriert wird. Beide Sichten sind damit nicht widersprüchlich sondern ergänzen einander und sind in der Realisierung zu verschmelzen. Aus der Sicht der Instrumentierung ist die Menge der Ressourcen eines $STK(\cdot, t, k)$-Ausschnittes gemäß $\theta$ zu organisieren. Umgekehrt ist der Zuständigkeitsbereich $STK(m, t, \cdot)$ eines AS-Manager $m$ gemäß den Fähigkeiten der Instrumente weiter zu ordnen. Der gemeinsame Anker für alle Maßnahmen ist die Struktur über der Dimension Zeit, die auf abstraktem Niveau von den konzeptionellen Lebenszeiten $\Lambda(a_i)$ der INSEL-Akteure $a_i \in \mathcal{A}_t$ grob vorgegeben ist und gemäß $\epsilon$ weitere Substrukturierung erfährt. Auf dieses Ordnungsprinzip ist die abgestufte Entscheidungsfindung der Instrumente abzustimmen. In Abbildung 7.10 ist die Einordnung der Manager in die Instrumentierung (bzw. umgekehrt) und seine Auswirkung auf die drei Dimensionen des STK-Systemmodells dargestellt. Auf der linken Seite ist eine Situation zu einem Zeitpunkt $t_x$ abgebildet, deren Weiterentwicklung bezüglich $t > t_x$ in dem rechten Diagramm skizziert ist.

#### 7.2.1.1 Instrumentierung der Manager

Die abstrakten Manager zeichnen sich dadurch aus, daß sie jeweils an genau eine Akteursphäre für die Dauer ihrer Existenz gebunden sind. Dies impliziert keineswegs, daß Manager als aktive (s. EVA) oder passive (s. AdaM) Laufzeitobjekte zu realisieren sind, wenngleich diese Vorstellung bei der Betrachtung von Managern als Objekte zusammen mit der aus objektorientierten Sprachen vorgeprägten Auffassung über die Realisierung von Objekten durchaus naheliegend erscheint. Es sei bemerkt, daß es sich bei diesem Design-Fehler um ein Sprachproblem handelt, wie es in 5.3.2 und 5.4 diskutiert wurde. Durch die Assoziation mit dem vorbelegten Begriff „Objekt“ inklusive deren Realisierung werden Entscheidungen getroffen, deren Trag-

[1] Bem.: Der Begriff vertikal beruht auf der üblichen Betrachtung der „Abstraktionsebenen“ in 2-dimensionalen ($s/k$) Architekturmodellen.

weite erst spät zu Tage tritt. Sie ist vermutlich deshalb so naheliegend, da sich für die vielen anderen Möglichkeiten, Manager zu realisieren, keine anderen, ähnlich prägnanten Begriffe anbieten.[2] Die Konsequenz dieser Assoziation in den Experimentalsystemen war hoher lokaler Mehraufwand. Demnach mußte der Manageransatz generell als „zu ineffizient“ verworfen oder seine Realisierung von der Planungsinstanz $M$ grundlegend neu und wesentlich differenzierter geplant werden. Letzteres erfolgte in der vorliegenden Arbeit, da das konzeptionell Wesentliche des Manageransatzes, Dezentralisierung und Orientierung an den Anforderungen der Berechnungen für verteiltes Rechnen inhärent sinnvoll erscheint und wohl in jedem Fall bewerkstelligt werden muß.

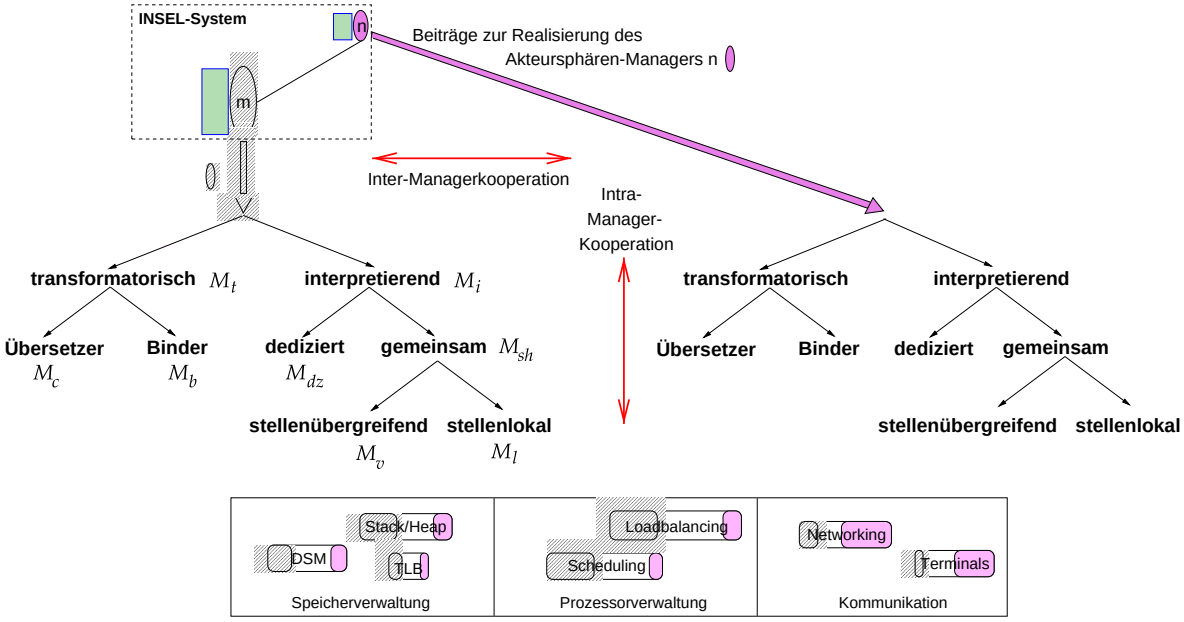


Abbildung 7.11: Instrumentierung der abstrakten Akteursphärenmanager

In dem Ansatz der vorliegenden Arbeit ist jeder Manager für sich ein Produktionssystem, wobei alle Manager eine gemeinsame, übergeordnete Planungsinstanz $M$ besitzen. Die Wahl der konkreten Instrumente ist einerseits von den Überlegungen im Zusammenhang mit der idealisierten Maschine $A_1$ und andererseits von der Bemühung geprägt, den Realisierungsaufwand im Rahmen zu halten. Abbildung 7.11 klassifiziert die Instrumente, die zur Realisierung der V-PK-Manager gewählt wurden grob gemäß der Unterscheidung zwischen Transformatoren $M_t$ und den Bestandteilen des repetitiv operierenden Restinterpretierers $M_i$. Jedes der dargestellten Instrumente kann einen Beitrag zur Realisierung eines Managers besitzen, wobei der exakte Umfang der einzelnen Beiträge von den Aufgaben des jeweiligen Managers, bzw. den Anforderungen seiner assoziierten Akteursphäre abhängt. U. u. werden die Aufgaben eines Managers, z. B. bereits vollständig durch Beiträge von $M_c$ geleistet. Durch die Variabilität der Beiträge ist die Realisierung der Manager keineswegs uniform, sondern so flexibel, wie es den Managern als separate Produktionssysteme mit den $A_1$-Fähigkeiten entspricht.

[2] Hinzu kommen weitere Faktoren, wie die Verfügbarkeit von Quelltexten zur Reduktion des Realisierungsaufwandes, etc. (siehe 4.3).

**Definition 7.21 (Manager-Realisierung)**
*Ein Manager*[3] $m \in M_t$ *ist ein Bindungskomplex,* $m \subset \mathcal{R}$*, der für einen Ausschnitt aus der Dimension* $s$ *die gesamte Dimension* $k$ *für die Lebenszeit* $\Lambda(a)$ *des Akteures* $a \in A$ : $man^{-1}(m) = a$ *überspannt. Er wird gebildet durch individuelle Beiträge der Instrumente des V-PK-Managementinstrumentariums.*

$$m = (M_c^m, M_b^m, M_{dz}^m, M_l^m, M_v^m)$$

Bei den Beiträgen $M_c$ und $M_b$ handelt es sich um die in das V-PK-Management integrierten Werkzeuge Übersetzer und Binder. $M_{dz}$ bezeichnet den von $M_c$ dediziert für eine definierte Menge von Komponenten generierten Anteil $i_{gen}^q$ des Restinterpretieres und entspricht somit dem vom Übersetzer generierten Zielcode für eine Problemlösungsspezifikation $q$ in INSEL. $M_{sh}$ ersetzt die in klassischen Systemen gewohnten Laufzeitbibliotheken zuzüglich Betriebssystemkern und umfaßt die Komponenten des INSEL Gesamtsystems, die den gemeinsamen Standardteil des Restinterpretierers $i_{std}$ bilden, der bei der Ausführung jeder Problemlösung eingesetzt wird. In $M_{sh}$ befindet sich der Großteil der Funktionalität zur verteilten Administration – $M_v$ – von Ressourcen, wie die Kommunikation zwischen den Stellen, aber auch solche Anteile, die sich mit stellengebundenen Ressourcen befassen – $M_l$. Neben den Beiträgen von $M_v$ und $M_l$ wird hier $M_{sh}$ nicht weiter differenziert, da Arbeiten am Kern oder an der Hardware in der vorliegenden Arbeit nicht durchgeführt wurden. Die charakteristischen Eigenschaften dieser Beiträge eignen sich, um die Abstufung der Konstruktion der Realisierungspfade, wie sie mit $A_1$ und im Zuge der Flexibilisierungsschemata erklärt und gefordert wurde, durchzusetzen. Anhand dieser Eigenschaften und elementarer Kriterien wird für eine konkret zu treffende Realisierungsentscheidung ein geeignetes Instrument gewählt.

Mit diesem Spektrum ist der grundsätzliche Rahmen der Managerrealisierung aufgespannt, der nach wie vor besonders in $M_{sh}$ große Freiheitsgrade zuläßt, die hier nicht vertieft werden. So wäre weiter zu unterscheiden, ob eine Managerfunktionalität überwiegend durch Daten oder Rechenvorschriften (s. 5.2.3) realisiert ist, in einer nebenläufigen Berechnung (aktiv) oder sequentiell eingebettet (passiv) erbracht wird und vieles mehr.

Ergebnis dieses Realisierungsansatzes ist dessen ungeachtet, daß alle beteiligten Instrumente an die spezifischen Anforderung in V-PK-Systemen, insbesondere der Strukturierungsprinzipien gemäß 2.3.2.3 zuzüglich $\theta$, angepaßt werden müssen. Die in diesen Werkzeugen üblicherweise gezogene Trennlinie zwischen Statik und Dynamik muß unbedingt überwunden und der bidirektionale Informationsfluß etabliert werden.

#### 7.2.1.2 Intra- und Inter-Managerkooperation

Das V-PK-Management ist somit durch die AS-Manager einerseits und die Instrumente andererseits in zweierlei Hinsicht separiert. Die Separation ist nützlich zur Strukturierung des Managements und erlaubt effizientes, autonomes und dezentrales Handeln. Auf der anderen Seite muß die Separation in vielen Fällen überwunden werden, um gesamtheitliche Entscheidungen zu treffen. Dies gilt für Entscheidungen, die mehrere Manager betreffen, wie z. B. das Prägen des gemeinsamen virtuellen Adreßraumes durch die AS-Manager, als auch Maßnahmen, die von mehreren Instrumenten gemeinsam vollzogen werden müssen, u. a. zur inkrementellen Konstruktion der Realisierungspfade, wie es oben erklärt wurde. Um Aufgaben dieser Art zu erfüllen, ist ein abgestuftes Repertoire an Kooperationskonzepten erforderlich, das sich in die grundlegenden Klassen Inter- und Intra-Managerkooperation gliedert.

[3] zu $M_t$ und $A_t$ siehe Def. 3.12

**Bemerkung 7.22 (Inter-Managerkooperation)**
*Eine Veränderung der Mengen $\mathcal{R}$ oder $\rho$, die durch zwei Manager $m, n \in M_t, m \neq n$ auf Basis eines gemeinsamen Zieles geplant und durchgeführt wird, heißt Inter-Managerkooperation (IEK).*

Inter-Managerkooperation (IEK) dient dem Austausch von Information oder der Lösung von Konflikten bzw. generell schwieriger Probleme, die autonom nicht oder nur mühsam bewältigt werden können. Ein wichtiges Beispiel für einen Konflikt, der durch IEK gelöst wird, ist die Zuteilung der physischen Prozessoren. Abstrakt betrachtet bedeutet IEK, den Übergang von Bindungen und Ressourcen zwischen den Bindungskomplexen der Manager, bzw. die temporäre Schaffung von Bindungen zwischen diesen.

**Bemerkung 7.23 (Intra-Managerkooperation)**
*Sei $m \in M_t$ ein Manager. Eine Veränderung der Ressourcen und Bindungen in $m$, die gemeinsam von unterschiedlichen Instrumenten aus der Menge $\{M_c^m, M_b^m, M_{dz}^m, M_l^m, M_v^m\}$ herbeigeführt wird, heißt Intra-Managerkooperation (IAK).*

Die Gliederung in IEK und IAK ist in doppelter Weise hilfreich. Einerseits erlaubt sie, die Maßnahmen des Managements präzise zu trennen und zu beschreiben. Andererseits verdeutlicht dies die Anforderungen, die an die einzelnen Managementinstanzen gestellt werden. IEK und IAK sind per Definition orthogonal, so daß grundsätzlich sämtliche Elemente des kartesischen Produkts IAK $\times$ IEK als Kooperationsszenarios zwischen den Instanzen des V-PK-Managements zugelassen sind und bei der Realisierung Berücksichtigung finden müssen.

Der Ansatz zur Realisierung dieser Kooperationsfähigkeiten unterscheidet zunächst zwischen dem Austausch von Information und der Durchführung bzw. Veranlassung von Maßnahmen. Zur Realisierung des Informationsaustausches wurde die bereits vorhandene **Attributierung** des Übersetzers als Grundlage gewählt und verallgemeinert.

**Definition 7.24 (Attribut)**
*Sei $a \in \mathcal{R}$ eine Ressource mit den Eigenschaften eines Datums. $a$ ist ein Attribut, wenn es an einen Bindungskomplex $B \subset \mathcal{R}$ gebunden ist und Auskunft über dessen Eigenschaften gibt.*

In herkömmlichen Systemen besitzen Attribute einen *statischen* Charakter, da sie meist lediglich vom Übersetzer erarbeitet werden und überdies Invarianten beschreiben. Ferner werden sie meist bei Beendigung des Übersetzungsvorganges aufgelöst. Gelegentlich werden zusätzlich *dynamische* Attribute eingesetzt, deren Wert während der Ausführung ermittelt und Lebenszeit auf die Dauer der Berechnung beschränkt ist. Der Gesamtsystemansatz und das V-PK-Management überwinden die realisierungsbedingte Unterscheidung zwischen Statik und Dynamik. Ferner wurde im Kontext von $A_1$ auf den inkrementellen Zugewinn an Information hingewiesen. Die von den Instrumenten des V-PK-Managements durchgesetzte Attributierung trägt dem Rechnung. Eine nützliche Klassifikation der Attribute unterscheidet in diesem Zusammenhang von welchen Instrumenten ein Attribut erarbeitet bzw. genutzt wird.

**T-Attribut:** Das Attribut wird ausschließlich in $M_t$ erarbeitet und genutzt.

**I-Attribut:** Das Attribut wird ausschließlich in $M_i$ erarbeitet und genutzt.

**hybrides Attribut:** Das Attribut wird von genau einem $m \in \{M_t, M_i\}$ erarbeitet und von der verbleibenden Instanz genutzt.

**transientes Attribut:** Das Attribut wird von $M_t$ und $M_i$ erarbeitet.

Hybride Attribute etablieren den Informationsfluß von Übersetzer und Binder an das eng an die Ausführung gebundene Management und umgekehrt. Transiente Attribute sind ein besonders wirksames Mittel zur Realisierung der inkrementellen Anreicherung der Präzision einer Information, bis hin zum Übergang einer Quasi-Invarianten in eine Invariante.

| | |
|---|---|
| $M_t \hookrightarrow M_{dz}$ | generativ |
| $M_{dz} \hookrightarrow M_t$ | prozedurorientiert |
| $M_{sh} \hookrightarrow M_t$ | prozedurorientiert |
| $M_{sh} \hookrightarrow M_{sh}$ | prozedur-/nachrichtenorientiert (breit-/schmalbandig) |
| Kern $\hookrightarrow M_{sh}$ | schmalbandige Nachrichten (Signale, Unterbrechungen), etc. |

Tabelle 7.2: Realisierungstechniken für Managementkooperation

Neben den Attributen, die dem Informationsaustausch zwischen allen Instanzen dienen und technisch durch gemeinsamen Speicher realisiert werden, existieren zahlreiche weitere Techniken, die zur Durchführung bzw. Veranlassung gemeinsamer Maßnahmen herangezogen werden. Zur Illustration sind einige weniger naheliegende Techniken aus diesem Kontext in Tabelle 7.2 angeführt ($m \hookrightarrow n$ bezeichnet $m$ kooperiert mit $n$). Ähnlich wie bei der Realisierung der Manager als Objekte, wäre es auch im Falle der Kooperation naheliegend, ein Konzept, z. B. Nachrichtenboxen, zu planen, realisieren und uniform einzusetzen. Dies würde die unterschiedlichen Anforderungen nach schmal- und – breitbandiger, bzw. synchroner und asynchroner Kommunikation ignorieren und durch das abermals generalisierte und wenig angepaßte Verfahren – zu gravierenden Ineffizienzen führen, die auch durch den Einsatz großer Mengen physisch verteilter Ressourcen nicht nivelliert werden könnten. Gleiches gilt für die Orientierung der IEK an der $\theta$-Relation, womit ein ernstzunehmendes und sinnvolles Strukturierungsprinzip vorgegeben ist, das aber auf der anderen Seite nicht besagt, daß in der technischen Realisierung starr Nachrichten entlang $\theta$ versendet werden müssen. Wie im Zusammenhang mit der Speicherverwaltung gezeigt wird, ist es in vielen Fällen erheblich effizienter, Ressourcenpools einzusetzen, auf die über den gemeinsamen Speicher prozedurorientiert zugegriffen wird. Die $\theta$-Struktur schlägt sich in diesen Fällen in der Strukturierung der Pools nieder.

Zum Abschluß des Abschnittes über die Einordnung der abstrakten AS-Manager in die Instrumentierung, ist in Abbildung 7.12 die Struktur des V-PK-Betriebssystems dargestellt, welches sich in die Manager und die Instrumente, die sie implementieren, aufteilt. Das Gesamtmangement und die Funktionalität des V-PK-Betriebssystems ergibt sich aus den einzelnen Beiträgen der Instanzen und der Kooperation zwischen diesen.

**Bemerkung 7.25 (V-PK-Betriebssystem)**
*Das V-PK-Betriebssystem ist die Realisierung des administrativen Teils des V-PK-Managements. Es umfaßt alle Instanzen, die die Ausführung der Akteurberechnungen, die Verteilung und Nutzung der abstrakten und konkreten Ressourcen steuern und überwachen.*

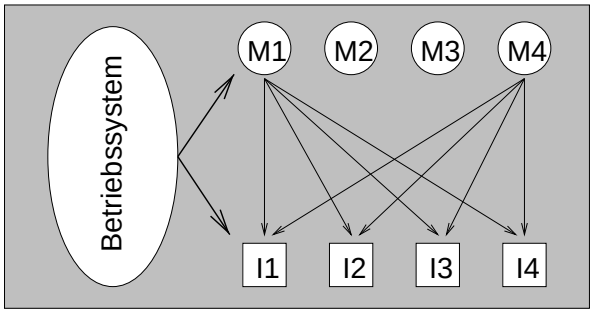


Abbildung 7.12: Struktur des V-PK-Betriebssystems

### 7.2.2 Stellenbasis

Die Stellenbasis umfaßt die Ressourcen der physischen Hardware sowie den Betriebssystemkern, der im privilegierten Modus der Maschine ausgeführt wird und damit die Ausgangsbasis für die Durchsetzung von Zugriffskontrollen liefert. Aufgrund der Konzentration auf leistungsfähige transformatorische Maßnahmen wurden im Rahmen der vorliegenden Arbeit bislang keine Realisierungsarbeiten an der Stellenbasis vorgenommen.

#### 7.2.2.1 Auswahlkriterien

Dennoch wurden vor dem Hintergrund der Anforderungen des integrierten V-PK-Managements zahlreiche Mikrokerne (u. a. Mach 3.0 [ABG+86], Chorus [The92]) und in Entwicklung befindliche Betriebssysteme (u. a. Spring[HK93]) im Hinblick auf ihre Eignung als Basis für die Realisierung des V-PK-Managements untersucht. Da Fähigkeiten bezüglich der physischen Verteilung nicht erwartet werden konnten, wurde die funktionale Eignung der Kandidaten nach folgenden, einfachen Kriterien beurteilt:

- Leichtgewichtige Aktivitätsträger zur Realisierung der Rechenfähigkeit.
- Leistungsfähige lokale und entfernte Kommunikationskonzepte, die möglichst an den Aktivitätsträgern orientiert sein sollten.
- Freiheitsgrade für die Realisierung der Speicherverwaltung.
- Freiheitsgrade zur Realisierung der inkrementellen Erweiterbarkeit.

Neben diesen funktionalen Gesichtspunkten wurde bei der Wahl der Stellenbasis auch berücksichtigt, inwiefern sie sich mittelfristig für die Weiterentwicklung des V-PK-Managements eignen würde. Eine solche Anforderung, an der Kandidaten, wie der Mach 3.0 Kern, scheiterten, war die Verfügbarkeit einer Implementierung auf einer **modernen Hardware-Architektur** mit $>$ 32Bit virtuellem Adreßraum. Das zweite wichtige Kriterium aus diesem Kontext, an dem Kandidaten wie HP mit dem Betriebssystem HP-UX scheiterten, war die Verfügbarkeit der **Quellen des Systems**, um notwendige Anpassungen vornehmen zu können. Dabei sei bemerkt, daß der PA-RISC Prozessor der Firma HP in Kombination mit dem Betriebssystem-Kern HP-UX 9.x auch die Realisierung der dynamischen Erweiterbarkeit verhindert hätte, da in diesem System der Prozessor dediziert auf die Ausführung von UNIX-Programmen mit *shared libraries* abgestimmt ist. Das nachträgliche Einbringen eines Maschinenprogrammes in

den UNIX-Prozeßadreßraum und seine anschließende Ausführung ist ohne Modifikation des Kerns nicht möglich. Ein ebenso gewichtiges Entscheidungskriterium war die Verfügbarkeit einer **Infrastruktur** für die Entwicklung von Systemprogrammen auf der gewählten Plattform. Die Abwesenheit einer fundierten Unterstützung zum Editieren, zur Fehlersuche oder Performance-Analyse hätte den ohnehin hohen Aufwand für die Realisierung der weiteren und ebenfalls wichtigen Instanzen des integrierten Managements nochmals deutlich in die Höhe getrieben.

Keines der untersuchten Systeme konnte sämtliche Anforderungen zufriedenstellend erfüllen. Als akzeptable Kompromisse verblieben die Alternativen:

1. Spring Nucleus auf der SUN Sparc Architektur

2. Mach 4.0 auf HP PA-RISC

3. SUN Solaris auf SUN V9 UltraSparc

Die Wahl des Spring Nucleus mußte aus den Gründen, die auf Seite 120 erläutert wurden, verworfen werden. Bei dem Mach 4.0 System handelt es sich um eine Weiterentwicklung des Mach 3.0 Kerns, die von einer Forschergruppe der University of Utah betrieben wurde. In Zusammenarbeit mit dieser Gruppe konnte der Mach 4.0 auf einer HP PA-RISC Maschine zur Ausführung gebracht werden, wobei sich allerdings gravierende Mängel zeigten, die sowohl die bereits durchgeführten Modifikationen an dem Kern als auch den Zustand der Infrastruktur betrafen. Die Überführung des Systems in einen nutzbaren Zustand hätte im Hinblick auf die Gesamtaufgabe inakzeptablen Aufwand verursacht und mußte deshalb ebenfalls verworfen werden.

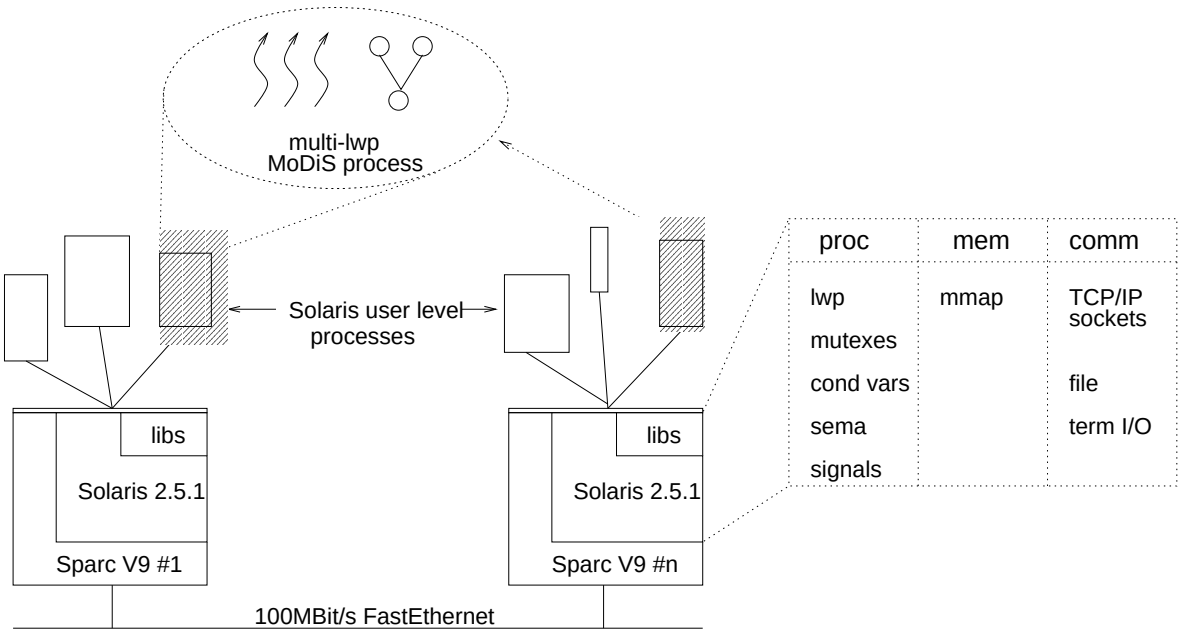


Abbildung 7.13: Aufbau der übernommenen Stellenbasis

#### 7.2.2.2 Nutzung der Kombination SUN V9 und Solaris 2.5.1

Die gewählte SUN Solaris Plattform mit der V9 Architektur erwies sich in den Realisierungsarbeiten als brauchbare Alternative. Der V9 Prozessor unterstützt 64Bit Adreßräume und ist damit zur Realisierung des verteilten Ein-Adreßraumes grundsätzlich einsetzbar, wenngleich Solaris in der Version 2.5.1 noch auf 32Bit beschränkt ist. Mit dem Wechsel auf Solaris Version 7 oder dem Betriebssystem Linux für UltraSparc kann diese Beschränkung aufgehoben werden. Auf die vorliegende Arbeit hatte diese Einschränkung keine bedeutenden Auswirkungen.

In Abbildung 7.13 ist der grundsätzliche Aufbau der Hardwarekonfiguration und die Nutzung der Stellenbasis zur Realisierung von INSEL-Systemen skizziert. Die Stellen der Konfiguration sind durch ein 100MBit/s FastEthernet verbunden. Auf jeder Stelle wird ein MoDiS-Prozeß ausgeführt, der als Hülle für die Realisierung von INSEL-Komponenten auf dieser Stelle dient. Die MoDiS-Prozesse aller Stellen realisieren gemeinsam das INSEL-Gesamtsystem. Dienste von Solaris werden dabei nur im geringem Umfang genutzt. Auf diese Weise sind die unerwünschten Einflüsse, unmodifiziert übernommener Verfahren auf ein Minimum reduziert. Gleichzeitig sind die Freiheitsgrade für die Entwicklung angepaßter Verfahren im Benutzermodus gewahrt.

Zur Realisierung der **Rechenfähigkeit** der Akteure werden *light weight processes* [Sun95b] (LWP) eingesetzt, die vom Solaris Kern verwaltet werden. Von Solaris ebenfalls angebotene *User-Level Threads* (ULT) erwiesen sich trotz ihrer geringeren Erzeugungs- und Kontextwechselzeiten als unbrauchbar. Bei einem ULT-Kontextwechsel werden globale Register nicht gesichert, was inkonsistent mit der Registerzuteilung einiger Übersetzer ist. Signale werden nicht zuverlässig dem ULT zugestellt, der als Empfänger spezifiziert wurde. Weiter findet für ULT keine preemptive Prozessorzuteilung statt. ULT sind insgesamt ein ausgezeichnetes Beispiel für ein mangelhaft integriertes Konzept.
Zur Synchronisation und Kommunikation zwischen LWP, was insbesondere zur Realisierung der IEK zwischen den $M_{sh}$-Anteilen der Manager benötigt wird, werden Solaris Semaphore, Bedingungsvariablen, Mutexe und Signale eingesetzt.

Im Zusammenhang mit der **Speicherfähigkeit** bedienen sich die MoDiS-Prozesse lediglich dem rudimentären Dienst `mmap`, mit dem Seiten im virtuellen Adreßraum (VA) der MoDiS-Prozesse dynamisch alloziert und freigegeben werden können. Alle weiteren Speicherverwaltungsaufgaben zur Realisierung der Keller und Halden der Akteure werden durch gezielt angepaßte bzw. neu entwickelte Managementinstanzen des V-PK-Managements innerhalb der MoDiS-Prozesse durchgeführt. Solaris unterstützt diese Vorgehensweise im erfreulichen Maß, indem mit `mmap` der VA, mit wenigen Ausnahmen, von Benutzer-Prozessen und damit auch dem MoDiS nahezu beliebig geprägt werden kann. Als Erweiterung der Stellenbasis wurden je nach Bedarf die verschiedenen DSM-Systeme MaX [Rei96], Shadow Stacks [GPR97] und das SVM-System Murks[4] zur Realisierung verteilter Speicherzugriffe eingesetzt. Ein grundauf neuer Ansatz zur adäquaten Realisierung des verteilten Speichers wird in aktuell laufenden Arbeiten verfolgt [RP99]. Das wesentliche Merkmal dieses Ansatzes, über den hier nicht berichtet wird, ist die dynamische Festlegung der Raum-Granularität der zu transportierenden Einheiten, als Mengen abstrakter Anwendungsobjekte, die auf Grundlage hybrider Attribute gebildet werden. Trotz dieser Flexibilität werden bei der Realisierung dieses Verfahrens die Fähigkeiten der V9 Hardware-Architektur genutzt, um Effizienz zu erzielen.

Für die **Kommunikation** zwischen den Stellen werden derzeit TCP/IP-Sockets genutzt, die in künftigen Arbeiten von leistungsfähigen und zuverlässigen Broadcast und Multicast-

[4] Praktikumsprojekt; Verteilte Systeme, TUM WS 1997/98, Prof. Dr. P. P. Spies

Protokollen zu ersetzen sind. Weiter wird die Dateischnittstelle genutzt, um Daten mit der Umwelt des INSEL-Systems auszutauschen und mit Benutzern zu kommunizieren.

### 7.2.3 GIC – INSEL Quelltext Transformation

Die Fähigkeiten des Übersetzers, der die Beschreibungen von Problemlösungen in der Sprache INSEL analysiert und den dedizierten Anteil $M_{dz}$ produziert, sind entscheidend für die Nutzung des Potentials des gesamten sprachbasierten Ansatzes und haben dominanten Einfluß auf die Leistungsfähigkeit seiner Realisierung. Die Wahl einer Zwischenhochsprache ist bereits anhand der Erfahrungen aus den Experimentalsystemen EVA und AdaM abzulehnen. Weitere Bestärkung findet diese Ablehnung durch die konzeptionellen Überlegungen im Rahmen des STK-Systemmodells. Der Übersetzer prägt die Realisierungspfade von $\mathcal{I}$ nach $\mathcal{H}$ erheblich. Eine Zwischenhochsprache bedeutet, wesentliche Abschnitte aller Pfade zu ignorieren und partiell, limitiert auf Teilstücke, zu optimieren. Das Resultat dieser Vorgehensweise ist, daß bereits lokal Mehraufwand entsteht und überdies eine adäquate Unterstützung der abstrakten und räumlichen Verteilung durch transformatorische Maßnahmen ausgeschlossen ist. Wird dahingegen der Übersetzungsvorgang von INSEL-Quellniveau bis auf das Maschinenniveau vollständig in das Ressourcenmanagement miteinbezogen, so wird der Handlungsspielraum des V-PK-Managements mit den zusätzlichen Instanzen $M_c$ und $M_{dz}$ erheblich erweitert, so daß tatsächliche Leistungssteigerungen möglich werden.

Aus diesen Gründen hatte die vollständige Kontrolle über die Transformationsschritte des Übersetzers in der vorliegenden Arbeit höchste Priorität. Um dies zu erreichen standen folgende Alternativen zur Wahl:

1. Vollständige Neuentwicklung eines Übersetzers — „*from scratch*“
2. Neuentwicklung mit generativen Techniken
3. Anpassung eines bestehenden Übersetzers

Der Aufwand für die Implementierung eines leistungsstarken, hoch optimierenden Übersetzers ist enorm, so daß eine vollständige Neuentwicklung nicht in Frage kam. Zudem wäre dies unnötig, da der überwiegende Aufwand für die redundante Neuentwicklung bereits realisierter Verfahren ohne jeglichen Erkenntniszugewinn oder Verbesserungen aufzubringen wäre. Trotz der verschiedenen Werkzeuge, die zur Realisierung von Übersetzern existieren, mußte auch Variante 2 als zu aufwendig verworfen werden, da allein die Implementierung gewöhnlicher Optimierungsstufen (siehe Abb. 2.15) bereits den gesamten zur Verfügung stehenden Zeitraum in Anspruch genommen hätte und der Verzicht auf diese die Leistungsfähigkeit des V-PK-Managements a priori rapide herabgesetzt hätte.

Die Implementierungsstrategie, die in der vorliegenden Arbeit generell verfolgt und insbesondere für den Übersetzer gewählt wurde, ist die Modifikation hochwertiger und frei verfügbarer Software, um arbeitsfähige und dennoch angepaßte Verfahren mit akzeptablem Aufwand realisieren zu können. Dabei sei bemerkt, daß sich gerade diese Arbeiten demonstrieren, wie schwierig es mit den existierenden Methoden ist, auf abstraktem Niveau festgelegte Bindungen aufzulösen und durch neue Ressourcen und Bindungen zu ersetzen. Der Grund dafür ist, daß die Evolutionsfähigkeit, die in Kapitel 6 beschrieben wurde, in gängigen Systemen bislang nicht systematisch verankert ist und somit ohne methodische Unterstützung bewerkstelligt werden muß.

Die Anpassung eines bestehenden Übersetzers an die Anforderungen in V-PK-Systemen bedeutet, INSEL als akzeptierte Sprache zu etablieren und zusätzliche Veränderungen an der semantischen Analyse sowie Zielcode-Synthese vorzunehmen. Als Ausgangsbasis für diese Arbeiten bot sich der im Quelltext verfügbare, wettbewerbsfähige GNU C Übersetzer – *gcc* – an [Sta95, Ken95]. Auf dieser Grundlage konnte in der vorliegenden Arbeit die wichtige Managementinstanz $M_c$ durch den hoch optimierenden INSEL Übersetzer *gic*[5] realisiert und vollständige Kontrolle über den Transformationsprozess von INSEL bis auf das Niveau der Maschineninstruktionen gewonnen werden. Die Architektur, spezielle Anpassungen sowie Details der Implementierung des Übersetzers *gic* wurden bereits ausführlich in gesonderten Berichten dokumentiert [Str96, Piz97, PEG97]. Im Folgenden beschränkt sich die Darstellung daher auf grundlegende Aspekte der Architektur, einen Überblick über die Informationsgewinnung und Erläuterungen zu einigen ausgewählten Anpassungen der Zielcode-Synthese.

#### 7.2.3.1 Architektur

Abbildung 7.14 stellt die Phasen des INSEL Übersetzers und die in diesen Schritten verarbeiteten Ressourcenklassen dar. Die Struktur des Übersetzers gliedert sich grob in das neu entwickelte und INSEL-spezifische **Front-End** und das von dem GNU Übersetzer übernommene, angepaßte und sprachunabhängige **Back-End**. Als Eingabe akzeptiert der Übersetzer Beschreibungen vollständiger und unvollständiger **DA-Komponenten**, wobei es sich um Generatorfamilien, Generatoren oder Inkarnationen handeln kann. Der Übersetzer ist derzeit noch nicht in der Lage, das INSEL-System inkrementell zu übersetzen, in dem Sinne, daß er seine Arbeit anhält und bei Aufnahme neuer Komponenten $K$ in das System fortsetzt, ohne bereits transformierte Komponenten nochmals neu bearbeiten zu müssen. Um dem Übersetzer die benötigte Kenntnis über das System zur Verfügung zu stellen, die er zur Übersetzung der $k_i \in K$ benötigt wird, werden bei $\delta$-geschachtelten Komponenten zusätzlich die Beschreibungen der $\delta$-äußeren Komponenten eingelesen. Um diesen Schritt zu beschleunigen, wurde die Möglichkeit geschaffen, neben INSEL auch vorverarbeitete AAST (s. u.) ohne erneute Analyse einzulesen.

Die syntaktische Analyse des Übersetzers im ersten Teil des Front-Ends wurde mit gewöhnlichen, generativen Verfahren entwickelt. Zwischen der Syntaxanalyse und der Synthese wurde eine abstrakte Syntaxbaum-Repräsentation (**AST**) eingeführt, deren Realisierung mit dem Werkzeug MAX [PH] ebenfalls generativ erfolgte. Auf der Ressourcenklasse AST erfolgt die gesamte semantische Analyse der Programme inklusive der Attributauswertung. Die Separation der konkreten Syntax von der abstrakten Syntax ermöglicht es, Attributberechnungen, die sowohl von synthetisierten als auch ererbten Attributen abhängig sind, losgelöst von der konkreten INSEL-Syntax, auf einem hohen Abstraktionsniveau in der funktionalen Sprache MAX zu spezifizieren. Effiziente Auswertungsalgorithmen inklusive der Bestimmung ihrer Reihenfolge werden auf dieser Basis von MAX automatisiert generiert. Die Fähigkeiten von MAX wurden im Zuge der Entwicklung des INSEL-Übersetzers erweitert, so daß Ressourcen der Klasse attributierte, abstrakte Syntaxbäume (**AAST**) durch das Front-End des Übersetzers ausgegeben und zu einem späteren Zeitpunkt für eine dynamische Neuübersetzung mit angereicherter Information in hybriden und transienten Attributen, ohne erneute Syntaxanalyse, wieder eingelesen werden können. Diese Fähigkeit ist die Verankerung des Rückflusses von Information sowie der Reaktivierung des Übersetzers, wie sie allgemein mit der hypothetischen Maschine $A_1$ erklärt wurde.

[5] GNU INSEL Compiler

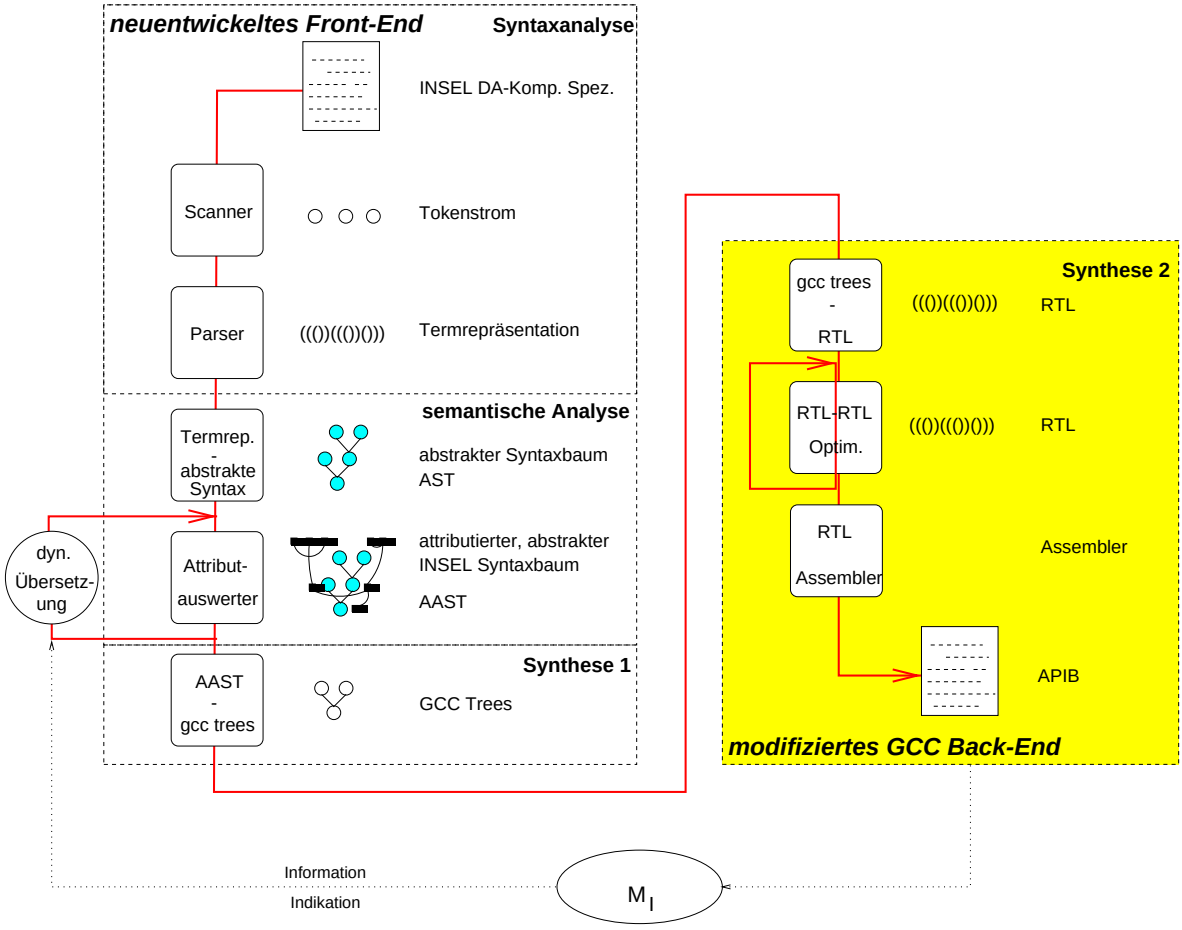


Abbildung 7.14: Architektur des INSEL Übersetzers und seine Koppelung an $M_i$

An der Schnittstelle zwischen dem **gic** Front-End und dem GNU C Back-End werden aus den AAST Ressourcen der Klasse **Tree** gefertigt, bei der es sich um abstrakte Syntaxbäume des GNU C Übersetzers handelt. Trees werden anschließend in die funktionale **RTL**[6]-Repräsentation überführt, auf deren Grundlage Übersetzer-Optimierungen durchgeführt werden und anschließend ein SPARC V9 Assembler Zielprogramm erstellt wird. Nicht dargstellt ist die anschließende Transformation des Assembler-Programmes in Ressourcen der Klasse **APIB**[7], durch den Assemblierer. APIB Ressourcen sind das primäre Produktionsziel des Übersetzers. Sie enthalten die auf den physischen Prozessoren ausführbaren Maschinenprogramme und sind die Eingaberessourcen der Managementinstanz $M_b$ – dem inkrementellen Binder.

Die Nutzung des GNU C Übersetzers als Ausgangsbasis hatte einige weitere positive Nebeneffekte. Einerseits sind für die Sprache INSEL durch den auf diese Weise implementierten Übersetzer zusätzliche Entwicklungswerkzeuge, u. a. zur Leistungsanalyse und Fehlersuche nutzbar. Weiter betreffen die Änderungen, die an dem Back-End des Übersetzers vorgenom-

[6] Register Transfer Language
[7] attributierter, prozessor-interpretierbarer Baustein

men wurden, alle sprachspezifischen Front-Ends, die der GNU Übersetzer unterstützt[8]. Dies ist eine wichtige Voraussetzung, um Programme oder Fragemente davon, die in anderen Sprachen als INSEL verfaßt sind, in das INSEL-System integrieren zu können, ohne dessen konzeptionellen Eigenschaften zu korrumpieren. Zuletzt sei darauf hingewiesen, daß der GNU INSEL Übersetzer *gic* durch das GNU Bach-End bei Bedarf mit minimalen Aufwand auf derzeit etwa 250 Hardwarearchitekturen und Betriebssysteme übertragen werden kann.

**Integration in das INSEL Gesamtsystem**
Der INSEL Übersetzer ist nach wie vor in der Sprache C und damit außerhalb des INSEL-Systems implementiert. Ebenso wird er als separater Solaris-Prozeß ausgeführt. In dem INSEL-Gesamtsystem ist der Übersetzer als M-Akteur-Generator maskiert, der nahe der Wurzel des stellenunabhängigen INSEL-Gesamtsystems plaziert ist. Bei Erzeugung einer Inkarnation wird ein Auftrag zur Ausführung des Übersetzers von dem MoDiS-Prozeß nach außen gegeben. INSEL-Programme, AAST und APIB Ressourcen werden zwischen dem MoDiS-Prozeß und dem Übersetzungs-Prozeß über die Solaris Datei-Schnittstelle transferiert.

Mit Ausnahme der Schnittstellen-Ressourcen INSEL-Programm, AAST und APIB ist die Zeit-Granularität aller Ressourcen des Übersetzers auf die Dauer eines Übersetzungsvorganges beschränkt. AAST, APIB und INSEL-Programme sind im INSEL-Gesamtsystem durch ebenfalls wurzelnahe Z-Depot-Inkarnationen repräsentiert.

#### 7.2.3.2 Informationsgewinnung

Von der separaten Attributierung der AAST Repräsentation wird intensiv Gebrauch gemacht, um Realisierungseigenschaften der INSEL Komponenten zu ermitteln. Abbildung 7.15 zeigt eine Selektion wichtiger Attribute und wechselseitigen Abhängigkeiten. Im unteren Teil der Darstellung befinden sich zusätzlich einige Realisierungsmaßnahmen, deren Durchführung von den oberhalb dargestellten Attributen abhängt.

Bei allen diesen Attributen und Realisierungsentscheidungen handelt es sich um Maßnahmen, die zusätzlich zu den umfangreichen Optimierungen des Back-Ends durchgeführt werden. Aus der dargestellten Menge werden im Folgenden einige interessante Attribute überwiegend informell erläutert, um den Einsatz der Attributierung zu illustrieren.

Grundsätzlich wird zunächst eine iterative Kontrollflußanalyse mit Hilfe von Grundblöcken vorgenommen. Das Ergebnis dieser Analyse, der Kontrollflußgraph, liegt anschließend in den *ControlFlow* Attributen der Elemente des AAST vor und wird im nächsten Schritt auf den Aufrufgraphen zwischen den DA-Komponenten vergröbert und in dem Attribut *CallGraph* gespeichert.

**Definition 7.26 (CallGraph)**
*Sei $X^P$ die Menge der Beschreibungen von DA-Komponenten im Programm P, $L_a^P(x)$ für ein $x \in X^P$ die Beschreibung des Anweisungsteils von x und $L_0^P(x)$ die Beschreibung des Deklarationsteils. Der $CallGraph(X^P, C), C \subseteq X^P \times X^P$ ist ein gerichteter Graph. Seine Kantenmenge beschreibt potentielle Aufrufabhängigkeiten und ist wie folgt definiert:*

$$(x, y) \in C \Leftrightarrow \exists a \in L_a^P(x) \cup L_0^P(x) : a \text{ ist ein an } y \text{ gebundener Erzeugungsauftrag}$$

Die Aufhebung von Statik und Dynamik in dem inkrementell konstruierten INSEL-Gesamtsystem verbietet die übliche Bezeichnung des *CallGraph* als „statischer Aufrufgraph“,

[8] Derzeit u. a. C, C++, Ada, Pascal und Fortran

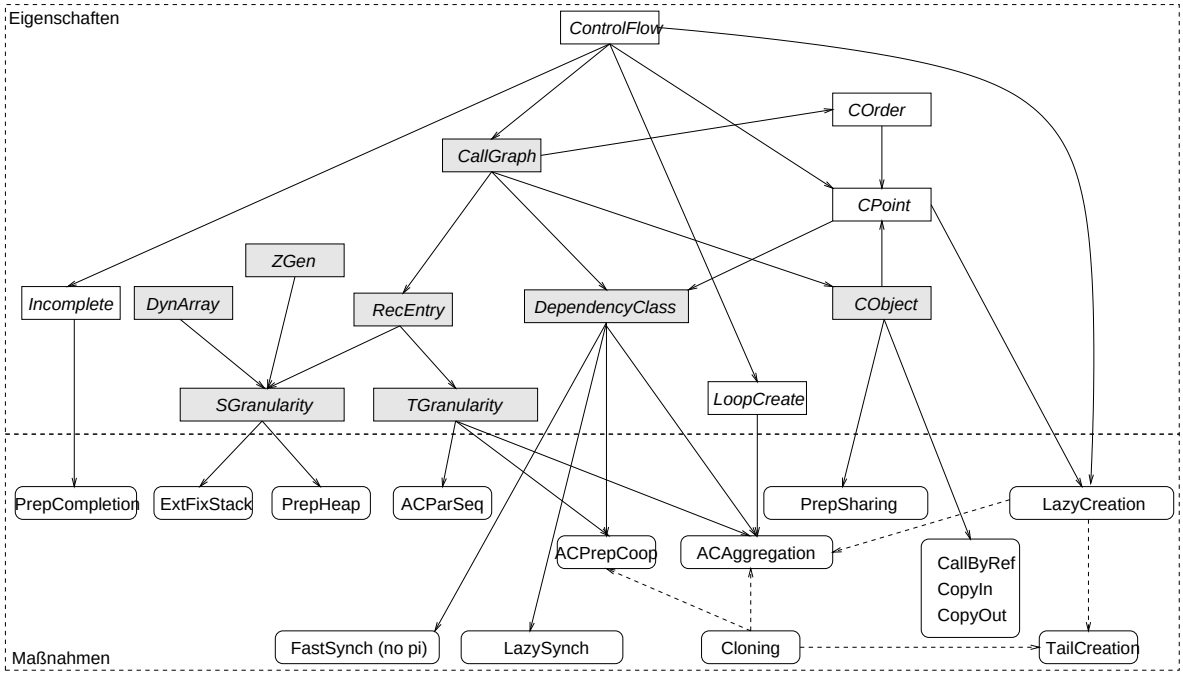


Abbildung 7.15: Attributierung des Übersetzers

da er selbst einer dynamischen Fortentwicklung unterliegt. Je nachdem, ob Generatorfamilien- oder Generatorbeschreibungen verarbeitet werden, sind zwei Aufrufgraphen zu unterscheiden, die durch Vererbung der Eigenschaften zwischen den Komponentenkategorien gemäß Korollar 6.8 voneinander abhängig sind und sich letztendlich in der $\alpha$-Struktur der DA-Inkarnationen widerspiegeln.

*CallGraph* wird unter anderem genutzt, um DA-Komponenten zu finden, die Zyklen des Aufrufgraphen dominieren und somit Eintrittspunkte für Rekursionen darstellen. DA-Komponenten auf die das zutrifft erhalten das Attribut *RecEntry*, das seinerseits benötigt wird, um Abschätzung über die Raum- und Zeit-Granularität – *SGranularity* und *TGranularity* – der DA-Komponenten vorzunehmen.

Der Wert des Attributes *SGanularity* beschreibt die Speicheranforderungen der assoziierten Komponente und hängt von den Attributen *RecEntry*, *ZGen* und *DynArray* ab.

**ZGen:** Gemäß der $\gamma$-Struktur werden die Z-Komponenten in der Lebenszeitrelation $\epsilon$ der DA-Komponente $x$ zugeordnet, für die der zugehörige Zeigergenerator ein Element aus $L_0(x)$ ist. Zur effizienten Realisierung dieses Konzeptes, das die Z-Komponenten-Halden in die Keller der DA-Komponenten einordnet, wird das boolesche Attribut *ZGen* erarbeitet. Es zeigt für jede DA-Komponente an, ob ein Zeigergenerator enthalten ist und deshalb Vorbereitungen für die Verwaltung von Z-Komponenten getroffen werden müssen. Ist dies nicht der Fall, so werden die entsprechenden Managementfähigkeiten in $M_{dz}$ ausgelassen und die Realisierung des betreffenden AS-Managers dadurch stark vereinfacht.

**DynArray:** Das Attribut *DynArray* gibt an, ob eine DA-Komponente Felder mit variablen Feldgrenzen besitzt. Ist dies der Fall, so ist die *SGranularity* einer DA-Komponente

keine Invariante, sondern kann vom Übersetzer lediglich spekulativ abgeschätzt und als hybrides Attribut zur weiteren Verfeinerung an $M_i$ übergeben werden.
Ähnliches gilt für Rekursionsdominatoren, die mit *RecEntry* gekennzeichnet sind. Die *SGranularity* einer DA-Komponenten $x$ mit dieser Eigenschaft wird auf die Summe der *SGranularity* Attribute aller DA-Komponenten gesetzt, die sich im Zyklus auf dem Pfad von $x$ bis zu dem nächsten Rekursions-Dominator befinden. Dem dedizierten Management $M_{dz}$ wird es auf diese Weise ermöglicht, vergröberte Speicheranforderungen, angepaßt an absehbare Anforderungen in der Zukunft, an $M_{sh}$ heranzutragen.

**SGranularity:** Trifft weder *RecEntry* noch *DynArray* zu, so wird der Bedarf an Kellerspeicher exakt errechnet und in *SGranularity* vermerkt. Dies wird unmittelbar genutzt, um die Größe der Kellerrahmen bereits durch den Übersetzer zu bestimmen, sofern dies möglich ist. Durch Analyse der *SGranularity* Attribute entlang des *CallGraph* werden ferner die initialen Größen der Keller für die Akteursphären-Berechnungen ermittelt. Das hybride Atribut *DynArray* erlaubt somit die flexible, angepaßte Realisierung der Akteurkeller, die je nach Bedarf von dem Minimum 8kByte (1 Seite) bis hin zu beträchtlicher Größe durch $M_i$ auf Basis dieser Information von $M_t$ eingerichtet werden. Das Attribut *SGranulatiy* ist damit ebenso wie *ZGen* ein plakatives Beispiel für das nichtuniforme, sondern flexible und inkrementelle Gestalten der Realisierungspfade zwischen den INSEL-Komponenten in $\mathcal{I}$ und den Hardware-Angeboten aus $\mathcal{H}$, wie es im Zuge der Flexibilisierung im Abschnitt 7.1.4 allgemein dargestellt wurde.

```
!#PROLOGUE# 0
save %sp,-136,%sp
call MANAGER_START_SYNC_PI,0
sethi %hi(.LLC5),%l4
call CONCUR2$SYSTEM,0
mov %l1,%g2
mov 1,%l1
sethi %hi(.LLC0),%l2
add %l0,-24,%g2
.
.
or %o0,%lo(.LLC7),%o0
call MANAGER_FINISH_SYNC_PI,0
ret
```

Abbildung 7.16: Abgestufte Realisierung der Abschlußsynchronisation

Ein weiteres hybrides Attribut ist „*DependencyClass*“, mit dem die Abhängigkeit einer DA-Komponente zu anderen Komponenten des Systems in ein einfaches Klassifikationsschema eingeordnet wird. Ergibt der *CallGraph*, daß eine DA-Komponente keine Akteure erzeugt, so können, ähnlich wie bei *ZGen*, gesonderte Einrichtungen zur Durchführung der konzeptionellen Abschlußsynchronisation mit $\pi$-inneren Akteuren eingespart werden. In Abbildung 7.16 ist ein Beispiel für ein generiertes Zielprogramm abgedruckt. Die Durchführung der Abschlußsynchronisation durch $M_{sh}$ wird seitens $M_{dz}$ durch die hervorgehobenen Aufrufe im Prolog und Epilog der Operation initiiert. Ergibt die Analyse des *DependencyClass* Attributes, daß keine $\pi$-inneren Akteure erzeugt werden, so werden diese Aufrufe von $M_c$ nicht generiert, wodurch

pro realisierten Anweisungsteil zwei Unterprogrammaufrufe eingespart werden können. Ohne diese Detailmaßnahme würde bereits hier beträchtlicher lokaler Mehraufwand entstehen, der den Erfolg der Nutzung verteilter Ressourcen zur Steigerung der quantitativen Leistung in Frage stellen würde. Bei der Realisierung von Akteuren wird die *DependencyClass* zusätzlich herangezogen, um zu beurteilen, ob die Berechnung eines Akteures ggf. auch verzögert bzw. allgemein durch Aggregation mehrerer feingranularer Akteurberechnungen mit einem gemeinsamen LWP realisiert werden kann, um deren kostspielige Erzeugung zu minimieren und eine geeignete Realisierungs-Granularität dynamisch und flexibel festzulegen.

Durch Analyse des Attributs „*TGranularity*“ wird versucht, die Berechnungsdauer von Akteuren abzuschätzen, um Entscheidungen bezüglich der Lastverteilung sowie der Erzeugung von LWPs zu stützen. *TGranularity* ist ein Beispiel für ein transientes Attribut[9]. Vom Übersetzer wird der Wert grob anhand einer Heuristik abgeschätzt. Während der Ausführung der Komponente wird in $M_{dz}$ die tatsächliche Berechnungsdauer durch Monitoring ermittelt und dem Übersetzer für folgende Transformationsvorgänge über die AAST Schnittstelle zurückgeführt.

Das letzte Attribut, das in dieser Kurzdarstellung wegen seiner großen Bedeutung erklärt wird, ist der boolesche Wert „*CObject*“, mit dem sämtliche DE-Komponenten versehen werden, die gemeinsame Komponenten in interferierenden Kontrollflüssen von Akteuren sind. Diese Information ist von grundlegender Bedeutung für die gezielte und automatisierte Realisierung verteilter Zugriffe auf wert-orientierte DE-Komponenten. Neben dieser allgemeinen Bedeutung wird die *CObject* Eigenschaft insbesondere genutzt, um die abstrakte Semantik der Parametermodi „Copy-In“ und „Copy-Out“ (s. Tab. 3.1 auf Seite 81) effizient zu realisieren. Das tatsächliche Umkopieren der Parameter im Speicher, wie es in EVA[10] und AdaM[11] durchgeführt wurde, erzeugt unakzeptablen Mehraufwand. Faktisch bleibt die abstrakte Semantik in vielen Fällen auch bei Übergabe in Registern oder durch ebenfalls effizientes *call-by-reference* gewährleistet. Ausschlaggebend dafür ist die Kenntnis, ob der formale Parameter als *CObject* genutzt wird. Der lokale Mehraufwand bezüglich der Parameterübergabe konnte mit dem neuen INSEL Übersetzer so nahezu eliminiert werden. Dabei sei darauf hingewiesen, daß die Nutzung des gesamten Spektrums von der Übergabe in Registern bis hin zu einer Kellerkopie nur dadurch möglich wurde, daß der gesamte Transformationsprozeß von INSEL in V9 Maschinensprache in das V-PK-Management miteinbezogen wurde.

#### 7.2.3.3 Realisierungsentscheidungen

Zusätzlich zu den bereits im Rahmen der Informationsgewinnung erklärten Maßnahmen werden vom Übersetzer weitere wichtige Realisierungsentscheidungen vorbereitet oder vollständig getroffen, die im unmittelbarem Zusammenhang mit den besonderen Anforderungen in parallelen und physisch verteilten Systemen stehen. In den folgenden Absätzen werden einige dieser Maßnahmen skizziert, um die besondere Eignung des Übersetzers zu belegen.

**Managerregister %g3**

Es ist offenkundig sinnvoll, sämtliche Information, die an eine Akteursphäre gebunden ist, wie z. B. das Start- und Ende seines Kellerspeichers, die Einordnung in die $\pi$-Struktur oder

[9] Realisierungsaspekte der effizienten inkrementellen Informationsgewinnung werden am Beispiel von *TGranularity* in aktuell laufenden Arbeiten untersucht.

[10] Parameter werden über die Halde kopiert $\Rightarrow$ lokaler Mehraufwand $l \approx 7$.

[11] Kellerkopie, $l \approx 2.5$

die Nummer des Solaris LWPs, der die Rechenfähigkeit der Akteursphäre realisiert, als Daten in dem dedizierten Anteil $M_{dz}$ zu speichern, bzw. zumindest dort zu verankern. Der INSEL-Übersetzer produziert hierfür Code in $M_{dz}$, der dafür sogt, daß bei der Erzeugung eines neuen Akteures ein Speicherbereich, mit einem Vielfachen der Seitengröße, von $M_{sh}$ angefordert und in $M_{dz}$-Daten sowie Akteur-Keller geprägt wird. Die $M_{dz}$-Daten dienen während der Lebenszeit der Akteursphäre als zentraler Ankerpunkt für sämtliche Maßnahmen, die den assoziierten Manager betreffen und werden als solche von sämtliche Instanzen, insbesondere aus $M_i$, intensiv genutzt.

Die Effizienz des Zugriffs auf diese Daten ist somit entscheidend. Die Frage lautet daher, wie der Daten-Anteil von $M_{dz}$ schnellstmöglich aufgefunden werden kann. Naheliegende Alternativen sind die Verwendung eines Tabelleneintrages in $M_{sh}$, ein globales Datum, das von dem Kern im Zuge der dynamischen Prozessorzuteilung gesetzt wird oder die Übergabe eines zusätzlichen Parameters entlang der $\alpha$-Struktur. Mit der vollständigen Kontrolle über die Code-Erzeugung war es möglich, diese Alternativen abzulehnen und stattdessen eine Hardware-Ressource unmittelbar und verbindlich an die $M_{dz}$-Daten zu binden. Das Back-End des Übersetzers wurde in diesem Sinne modifiziert, so daß das Register `%g3` der SUN V9 für die Identifikation der $M_{dz}$-Daten abgestellt ist[12]. Operationen von $M_{dz}$ und $M_{sh}$ ist es auf diese Weise ermöglicht, mit minimaler Verzögerung auf die benötigten Daten zuzugreifen.

Dies ist ein deutliches Beispiel für die Anpassung des Übersetzers an die $\theta$-Struktur des V-PK-Managements. Die Wahl eines Vielfachen der Seitengröße für die Daten reflektiert die besonderen Anforderungen der Ausführung auf einer NOW-Konfiguration und dient der Unterstützung möglicher Migrationen von Berechnung sowie deren entfernte Erzeugung.

**Erweiterte Alpha-Displays**

Für die Realisierung von Zugriffen auf N-DE-Komponenten, die in einer $\delta$-äußere Komponente eingeordnet sind, bietet das GNU C Back-End eine Verweisketten-Technik an. Ein Zugriff aus Schachtelungstiefe $n$ auf eine Komponente auf Ebene $m, n \geq m$ erfordert das Interpretieren von $n - m$ Zugriffsverweisen, d. h. das Lesen eines Speicherwortes und Dereferenzierung. Für sequentielle und zentrale Rechensysteme ist dieses Verfahren bei zu erwartenden geringen Schachtelungstiefen sehr effizient. In einem verteilten System kann im ungünstigsten Fall das Verfolgen jedes einzelnen Zugriffsverweises Kommunikation zwischen den Stellen erfordern, d. h. $(n-m)*x$ Nachrichten. Die Leistung wäre dementsprechend inakzeptabel. Eine bessere Alternative stellen *Displays* dar, mit deren Hilfe bei jedem Zugriff auf eine $\delta$-äußere Komponente nur genau ein Tabellenzugriff erforderlich ist. In dem Back-End des Übersetzers wurde deshalb die Verweiskettentechnik durch Displays ersetzt.

Dabei werden die Displays in umgekehrter Reihenfolge organisiert, d. h. die unmittelbar $\delta$-äußere Komponente ist mit Ausnahme einer Referenz auf sich selbst, der letzte Eintrag im Display. Diese spezielle Variante wurde entwickelt und implementiert, um es zu ermöglichen, daß das Display stets vom dem dynamischen $\alpha$-Vorgänger bezogen werden kann und nicht, wie üblich, von dem $\delta$-Vorgänger. Der Grund dafür ist ähnlich der Problematik der Verweisketten. Würden in einer Berechnung auf Tiefe $n$, Order auf Schachtelungtiefe $m$ mit $m \leq n$ erzeugt werden, so müßte jede dieser Order sich an ihre $\delta$-äußere Komponente wenden, die sich auf Tiefe $o, o < m$ befände, um ihr Display zu erhalten. Dabei muß davon ausgegangen werden, daß die Komponente der Tiefe $o$ nicht zu der Akteursphäre gehört, in der die Order erzeugt werden und damit die Wahrscheinlichkeit groß ist, daß sie auf einer anderen Stelle realisiert ist.

[12] „Managerregister“

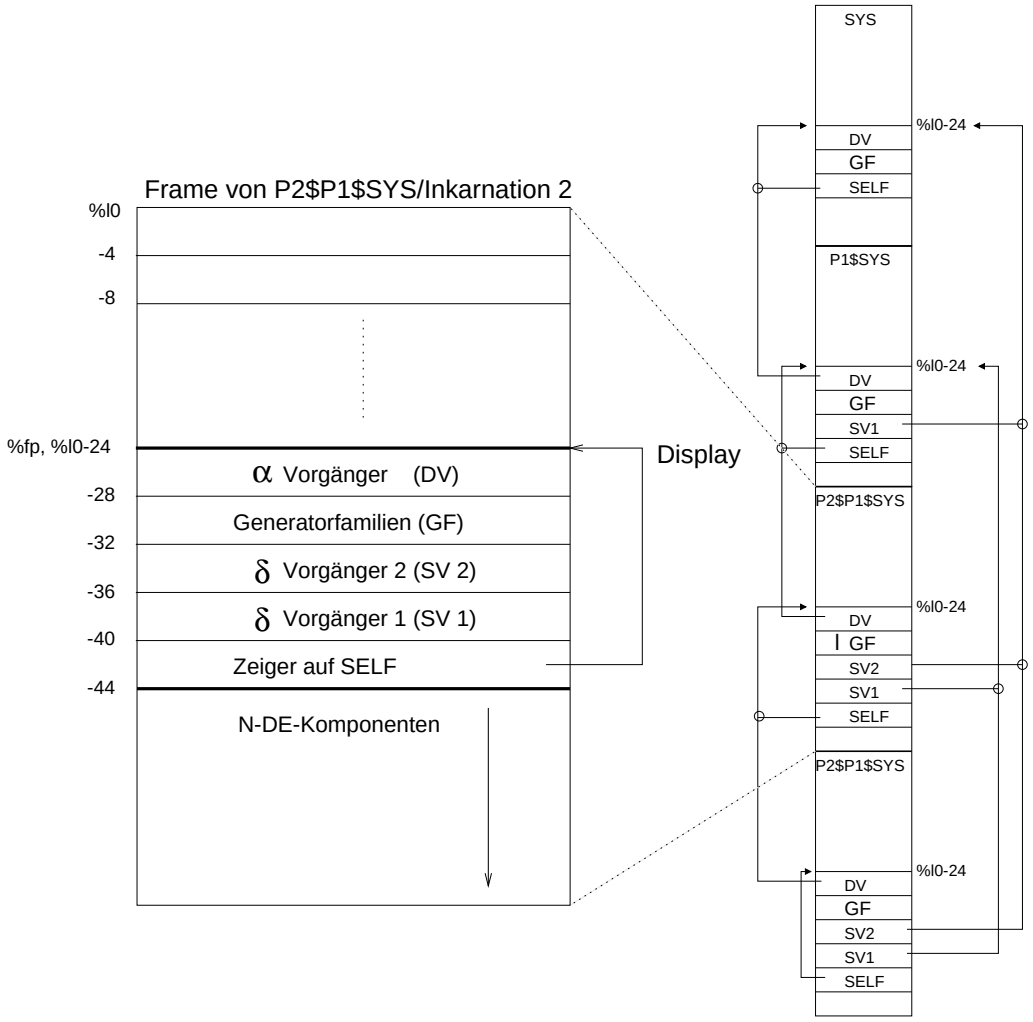


Abbildung 7.17: Erweiterte Alpha-Displays

Dies würde pro Order-Erzeugung abermals das Versenden von Nachrichten nach sich ziehen. Mit den $\alpha$-Displays, die in dem INSEL-Übersetzer realisiert wurden, ist dieses Problem gelöst. Das Display wird stets von dem unmittelbaren $\alpha$-Vorgänger beschafft, welcher in aller Regel der selben Akteursphäre angehört und damit keine Kommunikation über das Verbindungsnetz erfordert. In Abbildung 7.17 ist der Keller einer Akteursphäre und ein Rahmen einer $\delta$-inneren DA-Komponente inklusive erweitertem $\alpha$-Display dargestellt.

Im Zuge der Modifikation des Kellerrahmen-Layouts wurde noch drei weitere Änderungen zur Anpassung an die speziellen Anforderungen in V-PK-Systemen vorgenommen.

**Realisierung der Views:** Es wurde ein Feld für den $\alpha$-Vorgänger in das Display und damit $M_{dz}$ aufgenommen, mit dem alle Managementinstanzen aus $M_t$ und $M_i$ den $\alpha$-Graphen des Systems durchwandern können. Dies wird insbeondere von dem inkrementellen **Binder** genutzt, um die Views durchzusetzen, die im Abschnitt 6.4 konzeptionell eingeführt wurden.

**Verankerung der Generatorfamilien:** In dem gleichen Kapitel 6 wurde die Kategorie der Generatorfamilien eingeführt, die unvollständig oder vollständig in den Kellerbaum der DA-Komponenten plaziert werden. Das zweite Erweiterungsfeld des Displays im Kellerrahmen einer DA-Komponente ist der Ankerpunkt zur Einordnung von Generatorfamilien und ihrer Beschreibungen in die Dynamik des Systems.

**Diskontinuierliche Keller:** Es wurde das Frame-Pointer Registers `%fp` durch das Register `%l0` ersetzt und `%fp` als Zeiger auf die erhaltenen Argumente festgelegt.
Diese Modifikation erlaubt es, Kellerrahmen vor dem Beginn der Argumente zu spalten und an einem anderem Ort im virtuellen Adreßraum fortzusetzen. Auf dieser Grundlage ist das V-PK-Management in der Lage, Keller-Überläufe, wie sie auf Seite 93 diskutiert wurden, auszuschließen.

**Produktion kompositionsfähiger Realisierungsalternativen**
Eine weitere hervorzuhebende Fähigkeit des Übersetzers ist die Produktion kompositionsfähiger Realisierungsalternativen im Sinne der allgemeinen Erläuterungen zur Flexibilisierung des Ressourcenmanagements (s. o.).

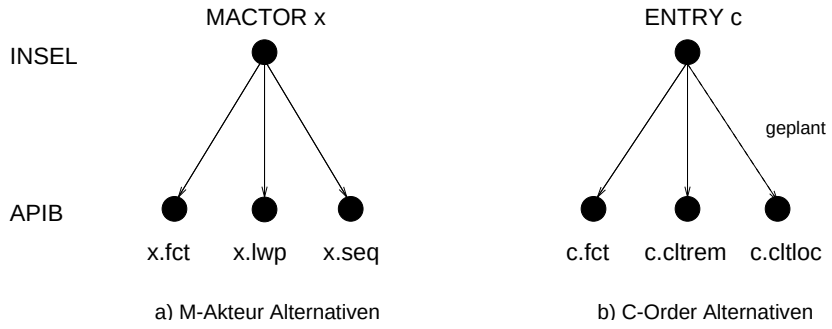


Abbildung 7.18: Produktion von Realisierungsalternativen

In Abbildung 7.18 sind die Alternativen dargestellt, die von $M_c$ als Beitrag zur inkrementellen und flexiblen Gestaltung der Realisierungspfade für M-Akteur- und C-Order-Beschreibungen produziert werden.

**M-Akteure:** Für eine Beschreibung von M-Akteuren `x` produziert der Übersetzer drei alternative APIBs. Erstens eine Standardvariante (`.fct`), die den Körper des M-Akteures implementiert. Zweitens ein Hilfs-Unterprogramm, das bei seinem Aufruf Vorbereitungen trifft, um gemeinsam mit $M_{dz}$ die Ausführung der Standardvariante durch einen neuen LWP zu veranlassen. Drittens eine spezialisierte Version `.seq`, die eine effiziente sequentielle Ausführung der M-Akteur-Berechnung ermöglicht. Die Zulässigkeit ihres Einsatzes hängt von dem Attribut *DependencyClass* ab.
Alle drei Varianten sind Bindungskomplexe mit äquivalenten abstrakten Eigenschaften und unterschiedlichen Realisierungseigenschaften, so wie es im Abschnitt 7.1.4 gefordert wurde. Der Nutzer einer dieser Varianten kann flexibel wählen ohne seine Schnittstellen anzupaßen. Diese Eigenschaft der Codeinvarianz, mit dem Ziel hoher Kompositionalität, wurde bei der Realisierung des INSEL-Übersetzers generell durch systematische Anpassung und Generierung geeigneter Unterprogramm-Prologe und Epiloge anstelle von Modifikationen in den Aufrufern (den Nutzern), erreicht.

**K-Order:** Für K-Order (Terminal `ENTRY`) werden drei APIB Bausteine produziert, mit deren Hilfe die Realisierung lokaler und entfernter Aufrufe differenziert wird. Die `.fct` Variante realisiert den Berechnungkörper der K-Order und wird von dem Auftragnehmer K-Akteur eingesetzt. Der Auftraggeber nutzt je nachdem, ob er sich auf der gleichen Stelle wie der Auftragnehmer befindet, die effiziente `.cltloc` oder im verteilten Fall, die aufwendigere `.cltrem`-Alternative. Auch hier erfordert die Wahl keine Änderung des Aufrufers.

### 7.2.4 FLink – Flexibles, inkrementelles Binden

Die Aufgabe eines Binders (siehe S. 64 ff.) ist das Festlegen von Bindungen zwischen den vom Übersetzer produzierten Bindungskomplexen. Im wesentlichen werden dabei horizontale Bindungen, die zwischen den Komponenten auf abstraktem Niveau bestehen, durch Bindungen auf Niveau der Hardware realisiert (s. Def. 7.4), so daß die Zusammenhänge von den Prozessoren der Hardware effizient interpretiert werden können. Die Konzentration auf das Festlegen von Bindungen zwischen Ressourcen einer Klasse, macht den Binder zu einem Paradebeispiel für die generelle Problematik der Bindung von Ressourcen.

Prinzipiell könnte gemutmaßt werden, daß das V-PK-Management keinen Binder benötigt. Die ohnehin einfachen Fähigkeiten eines Binders werden durch den Ein-Adreßraumansatz zunächst nochmals deutlich vereinfacht. Das übliche Binden von *shared libraries* an unterschiedlichen Positionen in verschiedenen Adreßräumen entfällt. Somit wäre es denkbar, daß der Übersetzer als Managementinstanz $M_c$ die verbleibende Funktionalität übernimmt, d. h. seine Produkte mit entsprechenden Bindungen auf dem Niveau $STK(\cdot,\cdot,\mathcal{H})$ generiert und selbst in dem virtuellen Ein-Adreßraum plaziert. Gegen diese Vorgehensweise sprechen drei Gründe, die gleichzeitig die Ziele des Binders formulieren und seine Rolle als Instanz $M_b$ für das V-PK-Management festlegen:

1. Flexibilisierung des Managements

   (a) In erster Linie werden durch den separierten Bindevorgang Bindungskomplexe mit wohldefinierten Abhängigkeiten isoliert, wodurch die **Raum-Granularität** abgestuft und Management-Flexibilität durch kompositionsfähige Bindungskomplexe im Sinne von Abschnitt 7.1.4 geschaffen wird. Mit dem Übersetzer wurden hierfür die Schnittstelleneigenschaften und Raum-Granularitäten der zu bindenden Ressourcen der Klasse APIB verbindlich vereinbart.

   (b) Die Instanz $M_c$ des V-PK-Managements besitzt die Fähigkeit, pro abstrakte INSEL-Komponente $n \geq 1$ Realisierungsalternativen zu produzieren. Die Möglichkeiten, diese Alternativen dynamisch und angepaßt an die Anforderungen einsetzen zu können, hängt von dem Binder ab. $M_b$ muß in der Lage sein, Bindungen zwischen den Übersetzerprodukten mit variabler **Zeit-Granularität** zu gestalten, so daß sie dynamisch initiiert und wieder aufgelöst werden können.

2. Die inkrementellen Konstruktion des INSEL-Gesamtsystems verlangt offenkundig nach inkrementellen Bindeverfahren.

Im Rahmen der vorliegenden Arbeit wurde der Binder FLink[13] konzipiert, implementiert und als Managementinstanz $M_b$ in das V-PK-Management integriert. FLink besitzt die geforderte

[13] Fast and flexible Linker

Fähigkeit, die APIB Zielressourcen des Übersetzers dynamisch inkrementell an das System in Ausführung zu binden und die Verbindlichkeit der Bindung variabel zu gestalten, so daß mit geringem Aufwand zwischen Alternativen gewechselt werden kann. Im Gegensatz zu dem Übersetzer mußte der Binder FLink vollständig neu entwickelt werden, da hierfür weder generative Techniken verfügbar sind noch ein anderer Binder aufgefunden werden konnte, der als geeignete Ausgangsbasis für eine Weiterentwicklung hätte dienen können. Die Ergebnisse dieser Arbeiten bis hin zu Details der Implementierung sind ausführlich in [Reh98b, Jek, Reh98a] beschrieben. Die folgende Darstellung des Binders beschränkt sich daher auf eine Zusammenfassung der wesentlichen Eigenschaften. Hierfür wird der Binder zunächst auf hohem Abstraktionsniveau als Bindeautomat beschrieben, woran sich kurze Erläuterungen zu den simultan eingesetzten Bindungsvarianten anschließen.

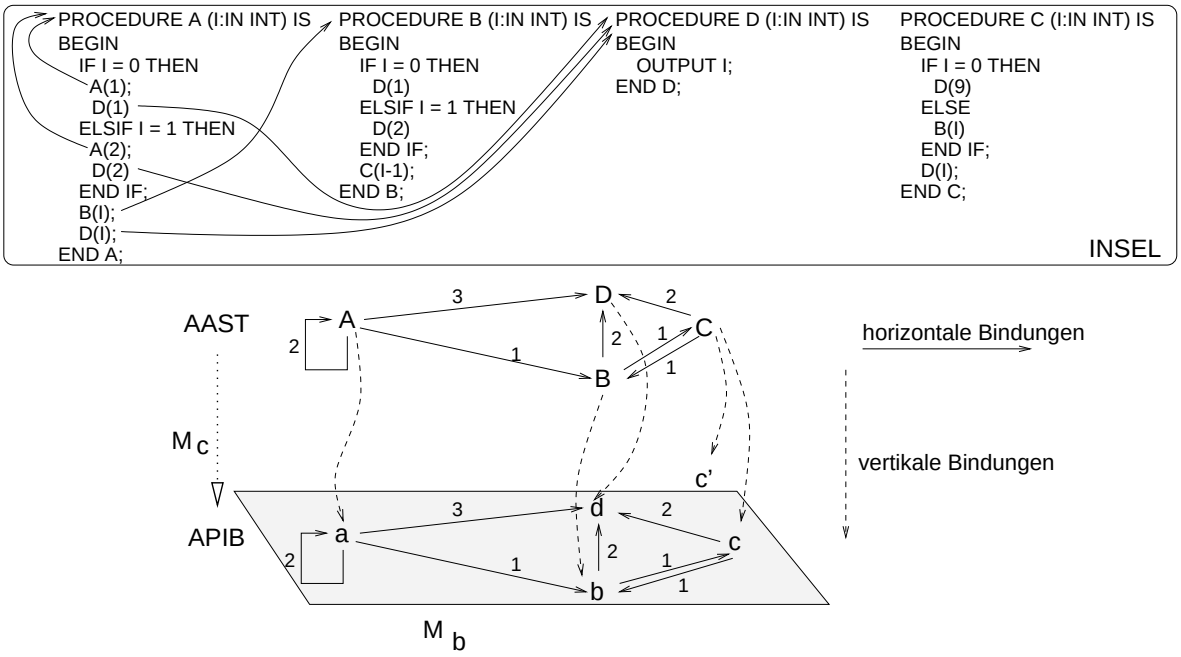


Abbildung 7.19: Bindungsgraphen

#### 7.2.4.1 Der Binder als Automat

In Abbildung 7.19 ist ein Beispielszenario dargestellt, das die Ausführungen über die Instanz $M_b$ des V-PK-Managements begleiten wird. Im oberen Teil ist ein synthetisches INSEL-Programm abgedruckt, das vier PS-Order-Generatorfamilien beschreibt. In den Anweisungsteilen der Komponenten sind Aufrufe der anderen Komponenten spezifiziert, die invariante horizontale Bindungen zwischen den Ressourcen auf diesem Niveau – $STK(\cdot,\cdot,\mathcal{I})$ – festlegen (im Bild sind diese lediglich aus der Sicht von `A` eingezeichnet). Unmittelbar unterhalb des Programmtextes ist das Programm vergröbert auf die DA-Komponentenfamilien und den Abhängigkeiten zwischen diesen dargestellt. Diese Sicht entspricht in etwa dem *CallGraph* der AAST Repräsentation des Übersetzers. Gemäß Definition 7.4 müssen zur Realisierung dieser Repräsentation für alle abstrakten Ressourcen Pfade $\bar{\mathcal{W}}$ von $\mathcal{I}$ nach $\mathcal{H}$ festgelegt und die abstrakten Bindungen zwischen diesen, strukturerhaltend durch Bindungen von Ressourcen

auf dem Niveau der Hardware – $STK(\cdot,\cdot,\mathcal{H})$ – realisiert werden. Ersteres, die Festlegung der **vertikalen Pfade**, wird durch $M_c$ mit der Produktion der APIBs erbracht. Die Schnittstellen der APIBs, die als Realisierungen der Körper von DA-Komponenten festgelegt wurden, entsprechen den qualifizierten Aufrufabhängigkeiten des abstrakten Niveaus. Die **horizontalen Bindungen** zwischen den APIBs werden von $M_c$ offen gelassen und es ist die Aufgabe von $M_b$ diese an APIBs zu binden. Ebenfalls in der Abbildung dargestellt sind zwei Realisierungsalternativen `c'` und `c` für die abstrakte Komponente `C`, wobei aktuell letztere eingesetzt wird.

**Bindungsgraphen**
Aus der Sicht von $M_b$ ist damit der Graph relevant, der von den APIB-Ressourcen und den (potentiellen) Bindungen zwischen diesen gebildet wird. In dem Beispielszenario ist ein solcher Graph, der *Standardbindungsgraph* eingezeichnet, der sämtliche, vom Übersetzer produzierten, Standardalternativen enthält. Neben diesem Graphen verwaltet $M_c$ mit *Akteursphärenbindungsgraphen* weitere APIB-Graphen, in welchen diejenigen Realisierungsalternativen gebunden sind, die von einer Akteursphäre gewünscht werden. So kann der Manager einer Akteursphäre zur Realisierung eines K-Order Auftrages an einen lokalen K-Akteur eine andere Realisierung der K-Order einsetzen als im entfernten Fall, wodurch das aufwendige Verpacken der Argumente in einer Nachricht eingespart wird. Dies entspricht der geforderten, flexiblen und gezielten Konstruktion der Realisierungspfade.

**Definition 7.27 (Bindungsgraphen)**
*Sei $K_t \subset \mathcal{G} \cup G \cup I$ die Menge der INSEL-DA-Komponenten*[14] *des Systems zum Zeitpunkt $t$, $A_{std}$ die Menge der APIB-Standardrealisierungen der Komponenten aus $K$, $A_{spez}$ die Menge der APIBs für $k_i \in K_t$ mit speziellen Realisierungseigenschaften und $A = A_{std} \cup A_{spez}$:*

- *Der* Standardbindungsgraph $\Gamma_{std}$ *ist definiert als*

$$\Gamma_{std} = (A_{std}, E_{std}), E_{std} \subseteq A_{std} \times A_{std}.$$

- *Der* Akteursphärenbindungsgraph *eines Akteures $x \in X \subset I$, $\Gamma_{AS}(x)$ ist definiert als*

$$\Gamma_{AS}(x) = (A, E), E \subseteq A \times A.$$

  *$\Gamma_{AS}$ bezeichne die Vereinigung aller Akteursphärenbindungsgraphen.*

*Die Kanten der Graphen werden ebenso wie ihre Knoten dynamisch erzeugt und aufgelöst. Jeder Graph ist damit von der Zeit $t$ abhängig.*

**Bemerkung 7.28**
*Offensichtlich sind die Knoten der Bindungsgraphen $\Gamma$ Elemente aus $\mathcal{R}$ und die Kanten Elemente aus $\rho$. Bindungsgraphen repräsentieren allerdings nicht zwangsläufig Bindungskomplexe, da sie nicht zusammenhängend sein müssen. Die aktuell nicht eingesetzte Alternative* `c'` *aus dem Beispielszenario macht dies deutlich.*

Gemäß dieser Definition ist es durchaus möglich, daß eine APIB $a$ in mehreren Bindungsgraphen enthalten ist. Dies ist erwünscht und entspricht der flexiblen Nutzung kompositionsfähiger Ressourcen. Die zusätzliche Dynamik der Kanten, erfordert in der Realisierung

[14] zu $\mathcal{G}$, etc. siehe Def. 6.2.

abgestufte Techniken, die Bindungsvarianten mit unterschiedlicher Intensität (s. 5.2.2) realisieren. Die gewählten Verfahren *unmittelbare Bindung*, *mittelbare Bindung*, *schwache Bindung* und *fail-safe Bindung* werden im nächsten Abschnitt erklärt. In das Modell des Binders geht die Variante der Bindung als Bewertung der Kanten des Bindungsgraphen mit Elementen aus $\{u, m, s, f\}$ ein.

Die klassische Sichtweise des Binders, der vor Ausführung des Programmes seine Arbeitet verrichtet, ist durch die Überwindung der Trennung von Statik und Dynamik inadäquat. Tatsächlich arbeitet der Binder kontinuierlich, solange das System existiert. Diese Veränderung wird in folgender Definition der Managementinstanz $M_b$ als Bindeautomat zum Ausdruck gebracht.

**Definition 7.29 (Bindeautomat)**
*Seien $K_t$ und $A$ wie in 7.27 vereinbart. Der Bindeautomat $M_b$ ist definiert als:*

$$M_b = (K_t, R, \Gamma, V, \tau)$$

- *$R \subseteq K_t \times A$ sei die Realisierungsrelation, für die gilt*
  *$R \subset \bar{\rho}_t^*, (k, a) \in R \Leftrightarrow a$ ist eine Realisierung von $k$*
  *Durch $R$ besitzt $M_b$ Kenntnis über die abstrakten, horizontalen Bindungen.*
- *$V = \{u, m, s, f\}$ seien die Bindungsvarianten.*
  *Jede Kante der Bindungsgraphen wird mit genau einem $v \in V$ bewertet.*
- *$\tau$ ist die Zustandsübergangsfunktion des Automaten.*
- $\Gamma = \Gamma_{std} \cup \Gamma_{AS}$

Die Zustandsübergangsfunktion $\tau$ ist im einzelnen umfangreich. Erläuterungen zu den verschiedenen Fähigkeiten finden sich in der angegebenen Literatur. Einige wichtige Übergänge sind:

- Konsistente Erweiterung und Verringerung der Menge $K$ und ihrer Realisierungen
- Erzeugung und Auflösung von Bindungsgraphen $g_i \in \Gamma_{AS}$
- Festlegung einzelner Kanten in individuellen Bindungsgraphen
- Wechsel der Kantenbewertungen individuell oder in Mengen abhängiger Graphen

Gesteuert wird $M_b$ aus zwei Richtungen. Erstens werden von $M_c$ neue INSEL-Komponenten und APIB-Realisierungen eingebracht. Zweitens veranlassen Instanzen aus $M_i$ die Auflösung von Komponenten und legen den Verlauf der Kanten innerhalb der Graphen fest.

#### 7.2.4.2 Bindungsvarianten

Damit diese Fähigkeiten keinen grundlegenden Mehraufwand während der Ausführung des Systems und damit Effizienzverluste verursachen, wurde ein abgestuftes Repertoire an Bindeverfahren entwickelt und realisiert. Mit den verschiedenen Varianten werden die Abhängigkeiten zwischen den APIBs, welche auf den *CallGraph*-Abhängigkeiten zwischen den abstrakten INSEL-Komponenten beruhen, unterschiedlich streng strukturiert bzw. gebündelt. Ohne zusätzliche Strukturierungsmaßnahmen werden APIBs zu eng verwobenen und damit sehr

| Aufrufer / R.-Ressourcen | Aufgerufener | | | | |
|---|---|---|---|---|---|
| | *a* | *b* | *c* | *d* | Anmerkung |
| a/6 | 2 | 1 | 0 | 3 | rekursiv |
| b/3 | 0 | 0 | 1 | 2 | verschränkt rekursiv |
| c/3 | 0 | 1 | 0 | 2 | |
| d/0 | 0 | 0 | 0 | 0 | |

Tabelle 7.3: Komponenten und Aufrufabhängigkeiten

effizienten Bindungskomplexen gebunden. Die Entnahme einer APIB ist in diesem Falle aufwendig. Ist z. B. die Nutzung von Realisierungsalternativen beabsichtigt, so empfiehlt sich die Bündelung von Abhängigkeiten und der Einsatz von Vermittlerkomponenten, wie sie in Abb. 7.7 erklärt wurden. Der Gewinn an Separation verursacht auf der anderen Seite höheren Kosten. FLink unterstützt insgesamt vier Varianten, die simultan eingesetzt werden können und das Spektrum von maximaler Effizienz bis maximaler Flexibilität abdecken.

**Unmittelbare Bindung**

In Tabelle 7.3 sind die Aufrufabhängigkeiten zwischen den APIB Bausteinen aus dem Beispielszenario angegeben. Zusätzlich zur Identifikation der APIB ist angegeben, wieviele nach außen gerichtete Referenzressourcen die APIB aufgrund von Aufrufabhängigkeiten besitzt. Bei diesen Verweisen handelt es sich um Referenzressourcen, die mit der Klasse APIB qualifiziert sind. Die Identifikation einer APIB ist in der Realisierung eine virtuelle Adresse aus dem Ein-Adreßraum, die den Start der APIB bezeichnet.

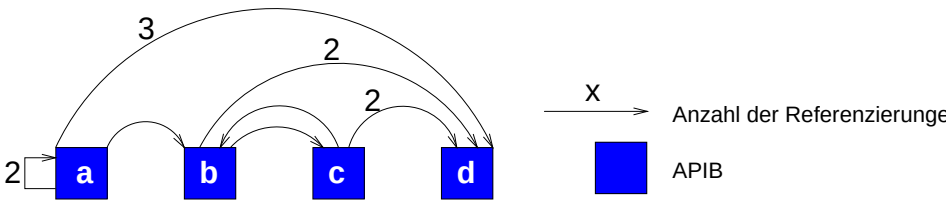


Abbildung 7.20: Unmittelbare Bindung

Bei der Variante der *unmittelbaren Bindung* werden die Referenzen der APIBs unmittelbar an die virtuellen Adressen der aufzurufenden APIB gebunden. In Abbildung 7.20 ist dargestellt, welche Abhängigkeiten sich in diesem Falle auf dem Niveau der Hardware ergeben, wenn alle Komponenten des Szenarios unmittelbar aneinander gebunden werden. Der Vorteil dieser Variante ist ihre hohe Effizienz. Zum einen werden keine zusätzlichen Vermittlerressourcen benötigt und zum anderen ist die Anzahl der Bindungen minimal. Daher sind die Pfade, die von den Prozessoren zu beschreiten sind, kurz. Der Nachteil dieser Variante ist, daß auf eine unmittelbar gebundene APIBs unter Umständen sehr viele Referenzen verweisen, was ihren Austausch sehr aufwendig macht. Soll im Beispiel APIB `d` durch eine andere Realisierungsalternative `d'` ersetzt werden, so müssen in Summe sieben Bindungen aufgelöst und neu festgelegt werden[15].

[15]Dabei wird vorausgesetzt, daß `d'` nicht an der selben virtuellen Adresse wie `d` plaziert werden kann.

Wegen ihrer hohen Effizienz wird die unmittelbare Bindung in gängigen Bindern meist als alleinige Variante eingesetzt.

**Mittelbare Bindung**

Die zweite von FLink angebotene und simultan zu den anderen nutzbare Bindungsvariante ist die *mittelbare Bindung* bzw. code-mittelbare Bindung, bei der *Trampolines* als Vermittler hinzugezogen werden. Bei einem Trampoline handelt es sich um drei Maschinen-Instruktionen, die ausschließlich zum Sprung auf eine feste APIB führen. In Abb. 7.21 ist ein solches Tram-

```
set  <APIB>, %g1
jmpl %g1, %g0
nop
```

Abbildung 7.21: SPARC V9 Trampoline Code

poline dargestellt (`<APIB>` wird durch die Adresse des assoziierten APIB ersetzt). Der Binder generiert für jeden APIB $x$, der unmittelbar gebunden werden soll, ein dediziertes Trampoline $tr(x)$. Alle Referenzen, die sich auf $x$ beziehen würden, werden anschließend an die virtuelle Adresse des $tr(x)$ gebunden.

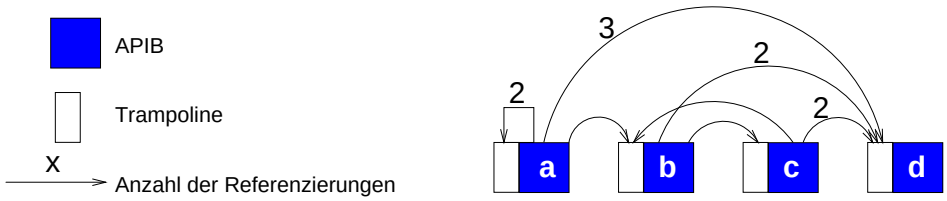


Abbildung 7.22: Mittelbare Bindung

In der Darstellung 7.22 sind alle APIBs mittelbar gebunden. Der Vorteil dieser Variante ist die Bündelung der Abhängigkeiten. APIB `d` kann in dieser Situation mit geringem Aufwand ausgetauscht werden. Hierfür ist lediglich der Verweis auf `d` in $tr(\texttt{d})$ anzupassen. Die Nachteile dieses Konzeptes sind geringe Leistungseinbußen, welche durch die zusätzlichen Trampolines und ihrer Bindung an APIBs entstehen.

Hierbei ist deutlich der fließende Übergang von transformatorischen zu interpretierenden Verfahren zu beobachten. Bei jedem Aufruf eines mittelbar gebundenem APIB wird das Trampoline erneut interpretiert um die Lokalisation des APIB in Erfahrung zu bringen. Bei der unmittelbaren Bindung erfolgt diese Interpretation einmalig bzw. nur bei Veränderungen am Bindungsgraphen. Eine weitere interessante Eigenschaft der mittelbaren Bindung ist der Übergang der Generierung von Fragmenten des Maschinenprogrammes von dem Übersetzer an den Binder. Indem der Binder Fragmente des Programms selbst generiert, bleibt der Übersetzer davon unberührt und die Kompositionalität seiner Produkte gewahrt. Revisionen von Entscheidungen über die Nutzung alternativer APIBs beschränken sich auf den Binder und die von ihm generierten Trampolines und werden damit äußerst effizient vollzogen.

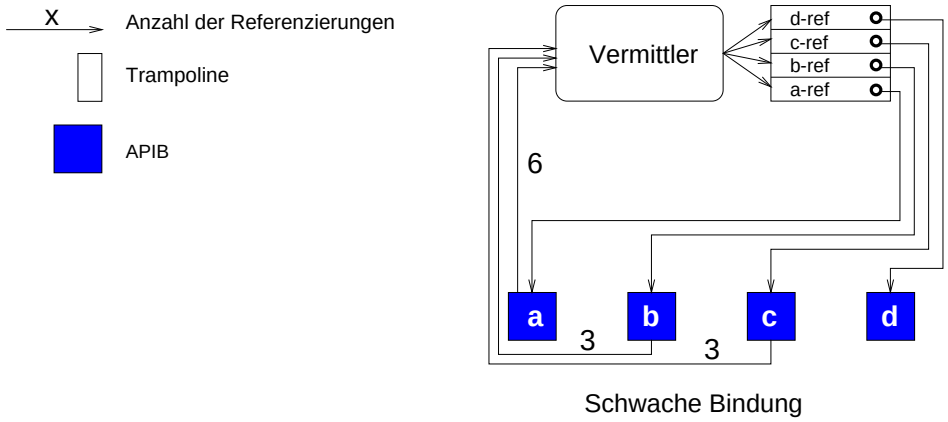


Abbildung 7.23: Schwache Bindung

### Schwache Bindung

Die ebenfalls realisierte, *schwache Bindung*, besitzt die größte Flexibilität. Der zur Strukturierung der Abhängigkeiten als Vermittler eingesetzte Bindungskomplex besteht aus einer Tabelle und einem Suchverfahren innerhalb dieser Tabelle. Offensichtlich ist das Ersetzen von Realisierungsalternativen in diesem Fall besonders einfach. Ebenso deutlich sind die höheren Kosten für dieses Verfahren, das wesentlich stärker interpretierenden Charakter besitzt. Die Zweckbestimmung dieser Variante gegenüber mittelbarer Bindung ist die Durchführung von Zugriffskontrollen durch die Vermittlerkomponente. So kann im Vermittler Buch geführt werden, von welchen Akteursphären eine APIB zu einem Zeitpunkt $t$ genutzt wird bzw. es können Sperren gesetzt werden, um aufwendige Rücksetzlinien im Falle von Revisionen zu vermeiden.

Die Implementierung der schwachen Bindung nutzt in dem Suchverfahren des Vermittlers die erweiterten $\alpha$-Displays, die von $M_c$ in den APIBs und damit $M_{dz}$ erzeugt werden, um sowohl den Aufrufer als auch die Aufzurufenden APIBs zu identifizieren. Diese Technik Kompositionalität zu gewährleisten und die schwache Bindung ebenfalls orthogonal zur Codeerzeugung des Übersetzers einsetzen zu können.

### Fail-safe Bindung

Bei dieser letzten Variante handelt es sich um eine Sonderform der schwachen Bindung, die der Detektion von Fehlern und der Behandlung unvollständiger Komponenten dient. In der Vermittlertabelle wird für eine fail-safe gebundene APIB ein gesonderter Vermerk notiert. Findet ein Zugriff auf eine solche APIB statt, so startet ein Suchverfahren, das mit Hilfe der erweiterten $\alpha$-Displays nach einer Vervollständigung sucht und bei Mißerfolg $M_{sh}$ auffordert, geeignete Schritte vorzunehmen – ggf. Aktivierung des INSEL-System-Editors zur Vervollständigung der Komponentenbeschreibung und anschließende Reaktivierung des Übersetzers.

### Quantifizierung der Kosten

Die Wahl einer dieser Varianten für eine zu bindende APIB ist von der gewünschten Flexibilität und Effizient abhängig. Um die unterschiedlichen Kosten präzise beurteilen zu können, wurden Messungen durchgeführt, deren Ergebnis in Tabelle 7.4 dargestellt sind. Im Versuch

```
FUNCTION foo(I: IN INTEGER) RETURN INTEGER IS
BEGIN
   RETURN (I+1);
END;
```

Abbildung 7.24: Testprogramm – Bindungsvarianten

wurde der einfache FS-Order-Generator `foo` aus Abbildung 7.24 genutzt, um eine Zahl von Inkarnationen in einer Schleife zu erzeugen. Die Meßdaten machen den Einfluß der Bindungsvariante auf die Leistung deutlich. Mittelbare Bindung verursacht einen Mehraufwand von $\approx 50\%$. Für sinnvolle und etwas längere Unterprogramme als `foo`, ist die mittelbare Bindung eine effiziente Alternative zur unmittelbaren Bindung, falls ein höheres Maß an Flexibilität benötigt wird. Bei den beiden anderen Verfahren liegt der Mehraufwand relativ zu unmittelbarer Bindung bei Faktor 100 bis 183, wobei zu beachten ist, daß dies lediglich für den Aufruf einer leeren Order wie `foo` gilt. Dennoch ist der Einsatz der beiden schwachen Bindungen sorgfältig abzuwägen. Von dem exzessiven Gebrauch einer tabellengesteuerter Bindung, wie sie in vielen erweiterbaren Systemen und Kernen betrieben wird, ist abzuraten.

| | Zeit für $n$ Aufrufe bzgl. `foo` in $\mu s$ | | | |
|---|---|---|---|---|
| $n$ | unmittelbar | mittelbar | schwach | fail-safe |
| 100 | 25 | 38 | 2380 | 4400 |
| 1000 | 230 | 360 | 24000 | 44100 |
| 10000 | 2400 | 3500 | 242000 | 440000 |

Tabelle 7.4: Meßdaten für unterschiedliche Bindungsvarianten

### 7.2.5 ISE – INSEL Laufzeitmanagement in INSEL

Den Schwerpunkt der Implementierungsarbeiten, die im Kontext der vorliegenden Arbeit durchgeführt wurden, wurde auf die transformatorischen Fähigkeiten des Übersetzers und des Binders gelegt, da hier bei vergleichbaren Ansätzen und den INSEL-Experimentalsystemen besonders große Defizite beobachtet wurden. Im Kontext der gemeinsamen interpretierenden Managementmaßnahmen $M_{sh}$ wurden zusätzlich Arbeiten im Bereich des Speichermanagements und zur Bewerkstelligung des Überganges von dem stellenunabhängigen auf das stellengebundene INSEL-System durchgeführt. Die Ergebnisse dieser Arbeiten werden weiter untenstehend zusammengefaßt.

Im Hinblick auf die **Verteilung der Rechenlast** wurde davon ausgegangen, daß die Verfahren, die im Kontext des Experimentalsystems EVA [Rad95] intensiv untersucht und implementiert wurden, in einem späteren Entwicklungsstand in das hier beschriebene V-PK-Management integriert werden. Derzeit werden Solaris LWPs, welche die Rechenfähigkeit der Akteure realisieren, von der Lastkomponente in $M_{sh}$ zyklisch den einzelnen Stellen der Konfiguration zugewiesen, sofern das Attribut der Zeit-Granularität die Berechtigung der physisch verteilten Realisierung indiziert. Ist dies nicht der Fall, so wird der LWP lokal erzeugt und ausgeführt.

In Bezug auf die räumlich **verteilte Speichverwaltung** fanden im Vorfeld zur vorliegenden Arbeit und parallel zu dieser, die Arbeiten an den Systemen AdaM [Win96b] und Shadow [Gro98] statt. Ebenso wie die Lastverteilungsverfahren von EVA ist geplant, die Speicherverwaltungsverfahren aus AdaM und Shadow in den Managementbeitrag $M_{sh}$ aufzunehmen. Ein flexibles, leistungsfähiges Verfahren zur effizienten, anwendungsangepaßten Realisierung hoch dynamischer Zugriffe auf lokale und verteilte Datenobjekte, das die Erfahrungen von AdaM und Shadow mit zusätzlichen Erkenntnissen aus dem Bereich *Distributed Caching and Replication*[TP99] verbindet, wird in derzeit laufenden Arbeiten entwickelt [RP99].

#### 7.2.5.1 INSEL Sprachfamilie

In Abschnitt 7.1.2 wurde erklärt, daß abstrakte INSEL Problemlösungen, durch Komponenten realisiert sind, die ihrerseits dem sprachbasierten Gesamtsystem angehören und daher selbst in INSEL beschrieben sind. Dies wirft ein Rekursionsproblem auf, für dessen Lösung bislang keine Antwort gegeben wurde.

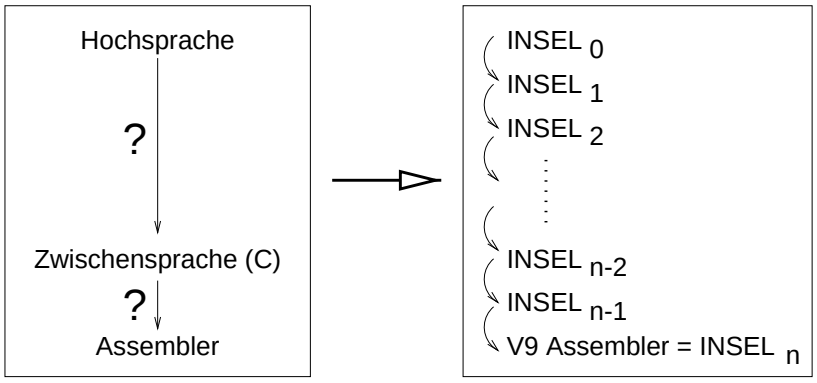


Abbildung 7.25: INSEL Sprachfamilie

In Abbildung 7.25 ist auf der linken Seite das Schema skizziert, nach dem dieses Problem in aller Regel gelöst wird. Auf hohem Abstraktionsniveau existiert eine ausdrucksstarke Sprache, die es erlaubt, Systeme mit wohldefinierten strukturellen Abhängigkeiten zu komponieren. Auf niedrigem Abstraktionsniveau ist mit der Sprache der Maschine eine weitere ausdrucksfähige Sprache vorhanden, mit der die Nutzung der realen Angebote flexibel spezifiziert werden kann. Beim Versuch der Realisierung der Konzepte der Hochsprache wird festgestellt, daß die von dieser Sprache angebotenen Konzepte, insbesondere bezüglich der Strukturierungsprinzipien durch strenge Typisierung, etc., nicht geeignet sind, um den Übergang auf das konkrete Niveau zu formulieren. Infolge wird eine Zwischensprache hinzugezogen, die sich selbst bereits auf einem sehr niedrigem Abstraktionsniveau befindet. Mit ihrer Hilfe werden die Konzepte der Hochsprache implementiert und mechanistisch, automatisiert auf das Maschinenniveau abgebildet. Problematisch an dieser Methode ist, daß mit der Zwischensprache in aller Regel die begründeten Restriktionen der Hochsprache ausgehebelt werden, ohne daß der Zusammenhang bzw. der Übergang systematisch verankert wäre. Die mangelnde Kenntnis der Übergänge macht es allerdings unmöglich, sie zu beherrschen oder auch nur zu kontrollieren, was sich in Fehlverhalten der realisierten Systeme widerspiegelt.

Bei der Implementierung der Instanz $M_{sh}$ wurde ein anderer Weg gewählt, um den Übergang zwischen den ausdrucksstarken Sprachen, von INSEL auf SUN V9 Maschinensprache bzw. die zugehörige Assemblersprache, zu erreichen. Durch die graduelle Abstufung der Sprache INSEL und ihrer Grammatik sind die Abhängigkeiten zwischen den verwendeten Konzepten auf den verschiedenen Abstraktionsniveaus konzeptionell erfaßt, womit ein Ankerpunkt für die systematische und evtl. automatisierte Kontrolle der Übergänge gesetzt ist. Die ursprüngliche Sprache INSEL wurde hierfür zu einer INSEL Sprachfamilie weiterentwickelt. INSEL auf hohem Abstraktionsniveau ist durch eine Grammatik $IG_0(N, T, P, S)$ definiert. Bei dem Übergang von einem Abstraktionsniveau $i$ auf das nächste darunter liegende, $i + 1$, werden konsistent Terminale, Nichtterminale und Produktionen aus der Grammatik $IG_i$ entfernt und neue hinzugefügt, so daß auf Basis der abstrakten Grammatik $IG_i$ die konkretere Grammatik $IG_{i+1}$ gebildet wird. Konkret wurden im Zuge der Entwicklung von $M_{sh}$ unter anderem folgende Schritte dieser Art vorgenommen:

- Aufnahme des Terminals `RTMANAGER`, mit dem der dedizierte Anteil $M_{dz}$, dessen Lokalisation in dem Register `%g3` gespeichert ist (siehe Übersetzer), von $M_{sh}$ im Zuge der Intra-Managerkooperation identifiziert werden kann.
- Aufnahme von Terminalen, die Datentypen mit einer fest definierter Anzahl Bits zur Spezifikation der verschiedenen Maschinenformate 64, 32, 16 und 8 Bit zur Verfügung stellen.
- Aufnahme von Terminalen für bitweise logische Operation zu der Alternativenproduktion für arithmetische Ausdrücke.
- Hinzunahme von Adreßarithmetik und Typkonvertierung.
- Auf einem niedrigen Niveau: Entfernung der Symbole und zugehöriger Regeln zur Spezifikation von Akteuren als Beitrag zur Lösung des Rekursionsproblems..

**Behauptung 7.30**
*Jedes System ist ein sprachbasiertes System.*

Im Grunde ist die Aussage, es handle sich um ein sprachbasiertes System, wenig aussagekräftig, da die Übergänge zwischen den Sprachniveaus von abstrakt nach konkret stets vorhanden sein müssen. Üblicherweise sind sie jedoch nicht systematisch erfaßt und bleiben unklar. Der Unterschied der gewählten Vorgehensweise ist, daß die Sprachbasierung in MoDiS, neben den bisher bereits erläuterten Eigenschaften, durch die graduelle Abstufung der INSEL-Grammatiken die zusätzliche Eigenschaft erhält, die Zusammenhänge zwischen den Repräsentationen auf unterschiedlichem Niveau konzeptionell zu verankern. Dies ist die Voraussetzung für die Beherrschung der komplexen Bindungen, die im Zuge der Konkretisierung in der Dimension $k$ hergestellt werden müssen und unterstützt ferner den homogenen Umgang mit den Ressourcen des Systems, indem Bruchstellen und Reibungsverluste vermieden werden.

#### 7.2.5.2 Übergang auf die Stellen in INSEL

Diese Vorgehensweise ermöglichte es, den Übergang von dem stellenunabhängigen Anteil des Managements $M_v$ auf den stellengebundenen Anteil $M_l$ ebenfalls in INSEL zu realisieren und homogen in das Gesamtsystem aufzunehmen. Die Vorteile dieser Integration sind vielfältig.

Unter anderem profitiert die Realisierung des Managements selbst von den qualitativen Vorteilen der INSEL-Konzepte, wobei Parallelismus auf hohem Abstraktionsniveau, die Kooperationsformen operationen-orientiertes Rendezvous und gemeinsame benutzbare passive Komponenten, adäquate Parametermodi sowie inkrementelle Erweiterbarkeit hervorzuheben sind. Für den größten Teil von $M_l$ stehen diese Fähigkeiten nach wie vor zur Verfügung.

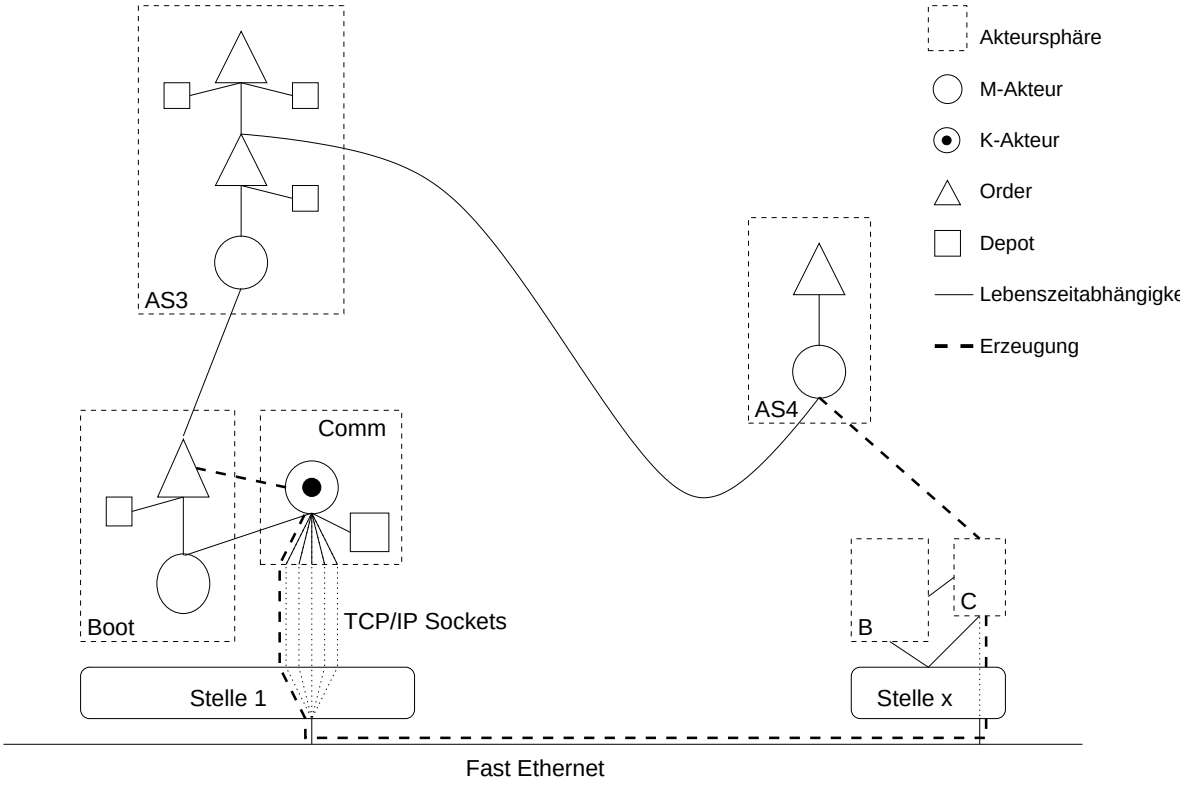


Abbildung 7.26: Übergang auf die Stellen

In Abbildung 7.26 ist zur Illustration die Struktur eines INSEL-Systems, das auf einer NOW-Konfiguration ausgeführt wird, inklusive dem Übergang auf die Stellen, abgebildet. Auf jeder Stelle, die an der Realisierung des INSEL-Systems teilnimmt, wird mit den Solaris Primitiven ein MoDiS-Prozeß gestartet. In diesem wird zunächst ein `Boot` M-Akteur inklusive seinem Manager[16] erzeugt, welcher den virtuellen Prozeßadreßraum grob prägt und anschließend den K-Akteur `Communicator` startet. Der `Communicator` initiiert und administriert sämtliche Netzwerk-Verbindungen zu den anderen Stellen über TCP/IP-Sockets. Anschließend zeichnen die `Boot` M-Akteure einen Repräsentanten aus ihrer Reihe eine als Wurzel des INSEL-Systems aus. Dieser (in der Abbildung auf Stelle 1) startet anschließend den ersten M-Akteur des stellenunabhängigen INSEL-Systems, aus dem sich das weitere verteilte System dynamisch und inkrementell entfaltet. Die `Boot` Akteure der Stellen beenden daraufhin ihre Hauptphase und warten auf die Durchführung der Abschlußsynchronisation mit den von ihnen erzeugten Akteuren. Der `Communicator` jeder Stelle wickelt sämtliche Aufträge ab, die Kommunikation mit anderen Stellen erfordern, z. B. die dargestellte, entfernte Erzeugung

[16] Manager sind in der Abbildung nicht eingezeichnet

eines Akteures. Als INSEL-K-Akteur ist der `Communicator` in der Lage, bei Bedarf $\pi$-innere M-Akteure zu seiner eigenen Hilfe abzuspalten, die dann ebenfalls Teil des stellenabhängigen INSEL-Systems sind. Für die Komponenten stellenunabhängigen Systems ist der Übergang auf die Stellen transparent. Unabhängig von dem Ort ihrer Ausführung nutzen sie mit den INSEL-Konzepten Komponenten ihrer Ausführungsumgebung, die dem stellenunabhängigen oder -abhängigen Teil des Systems angehören können, ohne diese Unterscheidung explizit zu spezifizieren.

#### 7.2.5.3 Prägung des virtuellen Ein-Adreßraumes

Im Bereich der Verwaltung multipler Berechnungen in dem gemeinsamen Ein-Adreßraum wurden Verfahren entworfen und implementiert, die das Problem von Überläufen und Kollisionen, das bei der Diskussion von Threads im Abschnitt 4.1.1.2 erklärt wurde, lösen. Umfassende Ausführungen zu diesen Verfahren und deren Realisierung durch das Instrumentarium des integrierten V-PK-Managements, finden sich in [Piz99a, Piz99b]. In den folgenden Absätzen wird das Verfahren in seinen Grundzügen skizziert.

**Partitionen**
Bei Start des verteilten INSEL-Systems wird zunächst, wie in Abbildung 7.27 dargestellt, der virtuelle Adreßraum (VA) des MoDiS-Prozesses grob in VA Partitionen eingeteilt. Am oberen und unteren Ende des VA befinden sich zwei Solaris Partitionen, die von dem Solaris-Prozeß selbst genutzt werden, wobei die Grenzen dieser Partitionen von dem `Boot`-Akteur determiniert werden. Der verbleibende und wesentliche Anteil des VA wird in die *Stellenpartition* und *verteilte Partition* unterteilt. Komponenten, die lediglich von dem stellengebundenen System genutzt werden, um z. B. Zustände der Maschine zu halten, werden in der Stellenpartition realisiert. Komponenten, die in der Stellenpartition einer Stelle $n$ realisiert sind, können ausschließlich auf der Stelle $n$ identifiziert und genutzt werden. Diese Separation hat einige Gründe und Vorteile, u. a.:

- Der Menge der VA Identifikatoren des gesamten Systems wird vergrößert.
- Zusätzliche Strukturierung des VA, auf deren Grundlage Zugriffskontrollen durchführbar sind.
- Migration, Replikation, etc. muß für die Stellenpartition nicht vorgesehen werden
- Ein große Menge von Komponenten, wie z. B. lokale I/O-Puffer, benötigt schlichtweg keine systemweit eindeutige Identifikation.

Den wesentlich größeren Anteil[17] an dem VA besitzt die verteilte Partition, in der jede virtuelle Adresse ein Bezeichner mit systemweit eindeutiger Bedeutung ist. Komponenten, die in diesem Bereich realisiert werden, sind auf jeder Stelle identifizierbar und darüberhinaus in der Lage zu migrieren oder repliziert zu werden, wobei durch die Ein-Eindeutigkeit der Identifikatoren keine Adreßtransformationen, z. B. aufwendiges *pointer swizzling*, nötig sind.

Die interne Organisation ist dadurch, daß sowohl stellenunabhängige als auch -abhängige Komponenten mit den INSEL-Konzepten konstruiert werden, bei beiden Partitionen gleich. Beide Partitionen beinhalten entgegengesetzt wachsende Halden und Keller, die durch die

[17] derzeit 3GByte gegenüber 256MByte der Stellenpartition

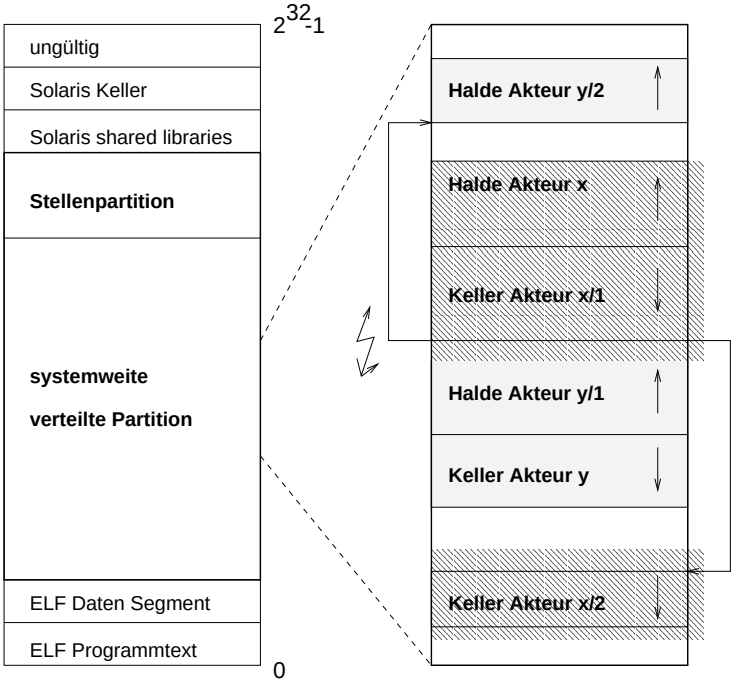


Abbildung 7.27: Prägung des virtuellen Adreßraumes

unten beschriebene Segmentierung in der Lage sind, bei Bedarf die gesamte zur Verfügung stehende VA Partition ausschöpfend zu nutzen.

**Regionenverwaltung**
Auf jeder Stelle der Konfiguration steht somit die Stellenpartition und ein adaptiv festgelegter Ausschnitt aus der verteilten Partition zur Verfügung. Diese Ausschnitte aus dem VA werden pro Stelle in einem Regionen-Pool verwaltet.

**Definition 7.31 (VA Region)**
*Eine VA Region ist ein Intervall aus dem virtuellen Adreßraum, daß an Grenzen virtueller Seiten startet und endet. Das Intervall muß dabei vollständig innerhalb einer Partition liegen.*

Ursprünglich wurde geplant, die Aufteilung der Partitionen in Regionen streng entlang der $\theta$-Struktur vorzunehmen, d. h. Regionen ausschließlich entlang dieser Struktur zuzuteilen. Es hat sich allerdings gezeigt, daß dies in vielen Fällen zu hohem Kommunikationsaufkommen führt. Insbesondere die rekursive Erzeugung von Akteuren oder die Erzeugung einer großen Zahl von Akteuren in einer Schleife stellen Problemfälle dieser Methode dar, die im *worst case* globale Kommunikation erforderlich machen würden.

Da es sich bei der Ressource VA um ein nahezu ubiquitäres Gut handelt, ist es demgegenüber sinnvoller, Regionen in Stellen-Pools zu sammeln und bei der Entnahme aus dem Pool das Wissen über die $\theta$-Einordnung gewinnbringend einzusetzen (s. u.). Deshalb wurde entschieden, pro Stelle einen Regionen-Allokator (RA) einzusetzen, der den dynamischen Pool der Regionen dieser Stelle verwaltet. Bei dem Start des Systems erhalten die RAs von dem als Wurzel ausgezeichneten `Boot`-Akteur Regionen der verteilten Partition zu ihrer autonomen Nutzung. In dem voraussichtlich seltenen Fall, daß auf einer Stelle der Regionen-Pool

erschöpft ist, tauschen die RAs der Stellen mittelbar über die `Communicator`-Akteure Regionen mit grober Raum-Granularität aus. Zur Administration der Regionen-Pools besitzen ein RA die Operationen *get, put, merge* und *split*. Mit den ersten beiden werden Regionen von dem RA angefordert bzw. an diesen übergeben. Mit den letzteren produziert der RA selbständig Regionen mit geeigneter Granularität.

Diese Systematik der RAs ähnelt den Wurzelverwaltern in AdaM [Win96b], unterscheidet sich von diesen jedoch signifikant in zwei Punkten. Erstens erfolgt die Zuordnung von Regionen zu den RAs der Stellen adaptiv. In AdaM wurde die Zuordnung von VA Bereichen an die Wurzelverwalter mit dauerhafter Verbindlichkeit festgelegt, indem einige Bits der virtuelle Adresse zur Identifikation der Stelle des zuständigen Wurzelverwalter eingesetzt wurden. Zweitens nutzen die RAs als $M_{sh}$-Instanzen bei der Kooperation mit dem dedizierten Anteil $M_{dz}$ die Kenntnis über die $\theta$-Struktur über den AS-Managern des Systems, um möglichst geeignete Regionen bezüglich der Granularität und Lokation im VA zur Verfügung stellen zu können.

**Segment-Keller**

In herkömmlichen Systemen bedecken sowohl die Keller der parallelen Berechnungen als auch die Halde kontinuierliche Adreßbereiche. In parallelen und insbesondere verteilten Systemen ist dies nicht adäquat, da auf diese Weise der potentiell große VA nicht ausschöpfend genutzt werden kann. Die Kollision zweier Keller oder eines Kellers mit der Halde stellt eine nicht behebbare Fehlersituation dar, obwohl der VA faktisch meist nur sehr dünn besetzt ist. Das integrierte V-PK-Management löst dieses Problem durch Abschwächung der Forderung nach Linearität und dem Einsatz von Segment-Kellern für die Keller und Halden der Akteure, mit deren Hilfe VA Regionen angepaßt an die Anforderungen bis zur vollständigen Ausschöpfung des VA nutzbar sind.

**Definition 7.32 (VA Segment und VA Segment-Keller)**

- *Ein VA Segment besteht aus einer oder mehreren VA Regionen, die gemeinsam ein Intervall des virtuellen Adreßraumes lückenlos überspannen.*

- *Ein VA Segment-Keller besteht aus individuellen VA Segmenten, die dynamisch mit* push *und* pop *nach dem LIFO-Prinzip dem Segment-Keller hinzugefügt oder diesem entnommen werden.*

Aus den Regionen, die der dedizierte Anteil $M_{dz}$ eines Managers von dem Stellen-RA anfordert und erhält, fertigt dieser zwei Segment-Keller. Einen für die Halde des Akteures (Halden-Segment-Keller – **HSK**), falls der Akteursphäre Z-Komponenten zugeordnet sind, und einen weiteren für den Keller der Akteursphäre (Keller-Segment-Keller – **KSK**), in dem die N-Komponenten der Akteursphäre realisiert werden (siehe Abb. 7.28). Die Verwendung von Segment-Kellern ist in beiden Fällen nützlich, da sowohl Keller als auch Halden, nur an einem Ende dynamisch erweiterbar sein müssen. Bei Halden ist dies lediglich seltener erforderlich als bei Keller, da zusätzlich innerhalb der Halde Freispeicher anfällt und genutzt wird, bevor die Halde als Ganzes wieder vergrößert – respektive verkleinert – werden kann.

Jedes Segment besitzt einen *Kopf* (hdr), in dem die Größe des Segmentes und ein Verweis auf das nächste Segment im Segment-Keller vermerkt ist. Bei HSK-Segmenten befindet sich der Kopf am unteren Ende und bei KSK-Segmenten am oberen Ende des Segmentes, um eine mögliche Erweiterung nicht zu behindern. Das aktuelle Segment eines Segment-Keller

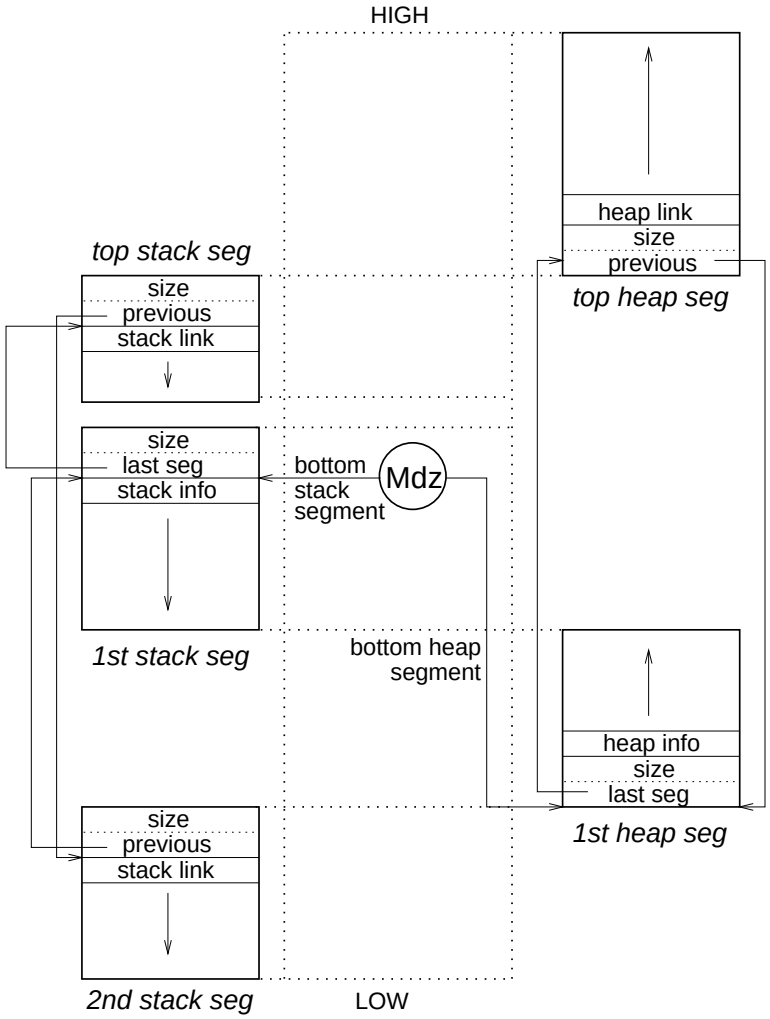


Abbildung 7.28: Segmentierte Keller und Halden

ist das *top*-Segment. Ist ein top-Segment voll und es erfolgt eine weitere Allokation in diesem Segment, dann würde das top-Segment überlaufen. Diese Situation wird von $M_{dz}$ (KSK) bzw. $M_{sh}$ (HSK) abgefangen. Diese wenden sich an den zuständigen RA, um eine neue Region anzufordern. Dabei nutzt der RA der Stelle das Wissen über die Manager-Struktur des V-PK-Managements und über die $\theta$-Struktur, um bei Möglichkeit eine Region zuzuteilen die sich unmittelbar an das obere (HSK) oder untere (KSK) Ende des übergelaufenen *top*-Segmentes anfügt. Je nachdem, ob dies gelingt, sind zwei verschiedene *top*-Segment-Extensionen zu unterscheiden:

1. **Lineare Extension:** es wurde eine passende Region gefunden und das top-Segment wurde um dieses in seiner Größe erweitert.

2. **Nichtlineare Extension:** es konnte keine passende Region gefunden werden. Deshalb wurde eine neue Region als neues *top*-Segment mit der Operation *push* dem betreffenden Segment-Keller hinzugefügt.

Unterläufe bewirken analog ein *pop* des aktuellen *top*-Segmentes.

**Extension der Halden**

Bei einer linearen Extension genügt es, die neue Größe des Segmentes im Segment-Kopf einzutragen. Anschließend kann die Berechnung des Akteures fortgesetzt werden. Diffiziler ist die effiziente Behandlung nichtlinearer Extensionen. In Abbildung 7.29 sind die Maßnahmen eingezeichnet, die erforderlich waren und vorgenommen wurden, damit Keller und Halden infolge nichtlinearer Extensionen an einer anderer Position im VA fortgesetzt werden können.

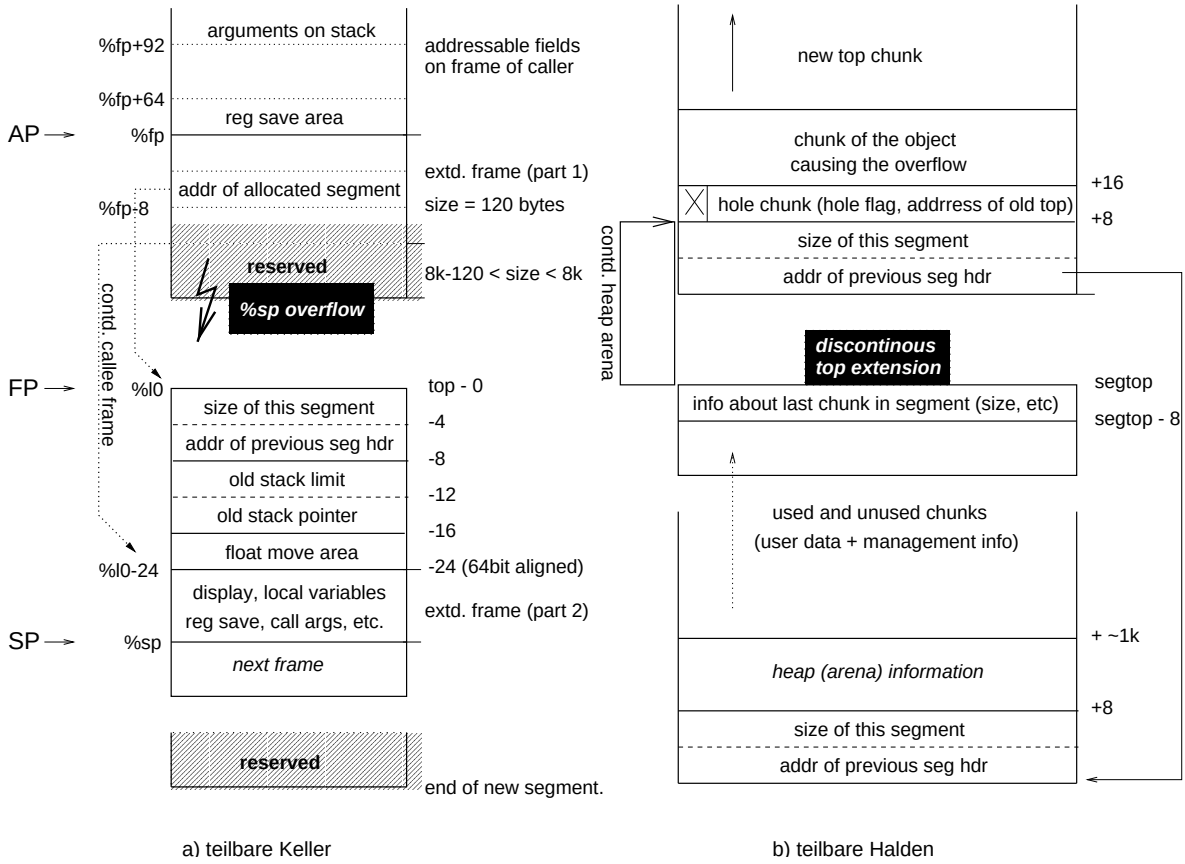


Abbildung 7.29: Separation der Keller- und Haldensegmente

Auf der rechten Seite ist eine nichtlineare Extension eines HSK dargestellt. Als Startpunkt für die Implementierung der Halden auf Basis der Segment-Keller wurde der effiziente und im Quellcode verfügbare Speicherallokator `dlmalloc` [Lea96] verwendet. Dieser gliedert die Halde in freie und belegte *chunks*. Die individuelle Verwaltung jeder einzelnen Z-Komponente in einem *chunk* erlaubt es, die Halde zwischen zwei beliebigen *chunks* zu spalten. Ein spezieller *top chunk* (TC) wird im Falle linearer Extensionen simpel verschoben. Um nichtlineare Extensionen des HSK zu ermöglichen, mußten einige Änderungen an dem Allokator vorgenommen werden. In einem solchen Fall wird mit diesen Modifikationen der aktuelle TC zu einem gewöhnlichen, nutzbarem *chunk* und am oberen Ende des aktuellen *top*-Segements wird ein speziell gekennzeichneter *chunk* plaziert, der den Allokator bei der späteren Suchen nach

Freispeicher auf die Fortsetzung der Halde an anderer Position im VA hinweist. Das neue *top*-Segment des HSK wird an seinem unteren Ende mit einem Segment-Kopf geprägt und oberhalb diesem durch einen *hole chunk* nach unten abgeschlossen. Im Anschluß an den *hole chunk* befindet sich der *chunk* des Objektes, das die Extension verursacht hat und der neue TC, der den noch verfügbaren Rest des Segmentes umfaßt.

**Adaptive Extension der Keller**
Das Konzept der Segment-Keller erlaubt auch im Falle der Akteursphären-Keller die adaptive Anpassung der Größe nach dem Prinzip der *lazy evaluation* ohne ein striktes Limit vorzugeben[18]. Jeder Akteur erhält zum Zeitpunkt seiner Erzeugung einen initialen KSK mit genau einem Segment, dessen Größe von dem Attribut *SGranularity* abhängt. Der Konsum an Kellerspeicher wird während der Berechnung des Akteures durch $M_{dz}$ überwacht und ggf. mit Extensionen des KSK befriedigt. Hierfür wurde die Zielcode-Synthese des Übersetzers auf niedrigem Abstraktionsniveau angepaßt, um insbesondere auch nichtlineare Kellerextensionen, die Spalten des Akteur-Kellers bedeuten, zu ermöglichen. Der Grundstein dafür wurde mit dem expliziten Argumentzeiger (AP, Register `%fp`) auf Seite 207 gelegt. Durch AP und dem zusätzlichen Rahmenzeiger (FP, Register `%l0`) ist es möglich, Kellerrahmen zwischen den Argumenten, die über AP adressiert werden, und den lokalen N-Komponenten, die über FP adressiert werden, zu teilen und an anderer Position im VA fortzusetzten. Das Layout der Sparc V9 Kellerrahmen wurde in diesem Kontext um das Feld *ExtFlag* erweitert, das die Adresse des vorhergehenden Segmentes im KSK beinhaltet. Bei einer nichtlinearen Extension wird das *ExtFlag* in dem Rahmen, der diese Situation ausgelöst hat, gesetzt. Bei dem *pop* des Kellerrahmens wird das *ExtFlag* überprüft und ggf. ein *pop* des aktuellen *top*-Segmentes veranlaßt. Das Back-End des Übersetzers wurde ebenfalls modifiziert, um Überprüfungen auf mögliche Kellerüber- oder -unterläufe in die Funktionsprologe und -epiloge der APIBs zu integrieren.

Die Entscheidung, Akteur-Keller innerhalb eines Kellerrahmens spaltbar zu gestalten ist vermutlich erklärungsbedürftig. Die naheliegende Alternative, den Akteur-Keller zwischen zwei Kellerrahmen aufzuspalten wurde ebenso untersucht und erwies sich als weniger nützlich, da in diesem Fall der Aufrufer die Überprüfung auf einen möglichen Segment-Überlauf durchführen müßte. Dies ist aus mindestens drei Gründen abzulehnen. Erstens, würde dies die Kompositionalität der APIBs herabsetzten, da unterschiedliche Realisierungalternativen verschiedenen Kellerbedarf besitzen können und bei einem Austausch sämtliche Nutzer modifiziert werden müßten[19]. Zweitens würden ähnliche Schnittstellenprobleme im Zuge der inkrementellen Systemkonstruktion auftreten. Drittens müßte der Programmtext für die Überprüfung als Bestandteil von $M_{dz}$ $n$-fach redundant in den $n$ Nutzern vervielfältigt werden. Die Verlagerung der Überprüfung auf Überlauf nach innen mit möglicher Spaltung des Kellerrahmens, wie sie hier erklärt wurde, ist deshalb die wesentlich attraktivere Alternative.

Durch diese Detail-Arbeiten konnte der **Mehraufwand** für die Unterstützung der KSK-Extensionen nahezu eliminiert werden. Pro Unterprogrammaufruf sind im Normalfall – keine Extension ist erforderlich – im Prolog 5 und im Epilog 3 zusätzliche, einfache Instruktionen abzuarbeiten. Dies führt in der Regel zu keiner spürbaren Verzögerung $\approx 1\%$. Durch eine Ausweitung der Perspektive der Kellerüberprüfung auf Rekursions-Dominatoren gemäß dem Attribut *RecEntry* könnte dieser geringe Aufwand nochmals deutlich reduziert werden. Die

[18] mit Ausnahme der Größe der jeweiligen VA Partition
[19] s. Strukturierung der Schnittstellen von Bindungskomplexen in 7.1.4.

zusätzliche Fragmentierung ist ebenso gering. Nur im Falle nichtlinearer Extension entsteht interner Verschnitt in den KSK-Segmenten. Sei $f$ die durchschnittliche Kellerrahmengröße, $r$ die Größe des Sicherheitsbereiches (reserved) und $s$ die durchschnittliche Segmentgröße. Für den ungünstigsten Fall, daß jede Extension nichtlinear wäre, liefert folgende Formel eine Abschätzung für den internen Verschnitt:

$$F_{avg} = \frac{r + ((s - r)\ mod\ f)}{s}; 8k - 120 < r < 8k$$

Für $f = 256, r = 8192$ und $s = 32k$ wäre $F_{avg} = 25\%$. Die Wahl geeigneter Segmentgrößen und die Berücksichtigung möglicher Folgeanforderungen durch den RA ist somit durchaus relevant. Werden adäquate Segmentgrößen an geeigneter Position zur Verfügung gestellt, so ist der interne Verschnitt eher unproblematisch.

Die Organisation der adaptiven Keller und Halden ist insgesamt ein abschließendes Beispiel für die Leistungsfähigkeit des V-PK-Managements, die dadurch erzielt wird, daß:

1. Sämtliche Entscheidungsinstanzen integriert in ein gesamtheitliches Management aufeinander abgestimmt zusammenwirken.

2. Nahezu der gesamte zu Verfügung stehende Spielraum an transformatorischen und interpretierenden Maßnahmen genutzt wird.

3. Die Strukturierung des V-PK-Managements gemäß der $\theta$-Struktur über den AS-Manager berücksichtigt wird; siehe RA.

## 7.3 Zusammenfassung

Orientiert an den Rahmenkonzepten aus Kapitel 5 wurde in diesem Kapitel das integrierte V-PK-Management schrittweise konkretisiert. Zunächst wurde das allgemeine Modell eines Produktionssystems entworfen, das die Zusammenhänge zwischen Anforderungen, konkreten und abstrakten Ressourcen, deren Produktion und vor allem die Planung aller Abläufe durch das Management, verdeutlicht. In dieses Modell wurden im nächsten Schritt verschiedene Sichten auf das INSEL-Gesamtsystem eingeordnet, wodurch einige grundlegende architekturelle Fragen beantwortet werden konnten. Anschließend wurde ein idealisiertes Instrumentarium konzipiert, das die Anforderungen, die an ein Produktionssystem gestellt werden, flexibel und effizient erfüllt. Das wesentliche Charakteristikum dieses Instrumentariums ist die Unterscheidung zwischen repetitiv interpretierenden und einmalig – bzw. selten – transformierenden Maßnahmen. Bei dem Übergang auf eine reale Instrumentierung sind zusätzliche, technische Randbedingungen zu berücksichtigen, deren Überwindung flexibles Handeln erfordert. Anhand der Ressourcen- und Bindungskonzepte wurde eine weites Spektrum an Möglichkeiten für flexibilisierenden Maßnahmen allgemein aufgefächert und mit Beispielen belegt.

Die Erkenntnisse aus der Beobachtung des idealisierten Instrumentariums und der Flexibilisierung wurden dann mit der Managementgrobarchitektur der multiplen Akteursphärenmanager verschmolzen, woraus sich eine konkrete Vorstellung für die Realisierung des V-PK-Managements ableiten ließ. Die durchgeführten Implementierungsarbeiten, die daraufhin erläutert wurden, fokussierten die transformatorischen Maßnahmen des Übersetzers und des

Binders, da in diesem Bereich besondere Defizite erkannt wurden. Es wurde ein hochoptimierender Übersetzer entwickelt, der INSEL-Programme ohne den Umweg über eine Zwischenhochsprache unmittelbar in effizienten Maschinencode transformiert und dabei spezielle Aspekte der physischen Verteilung sowie der Orientierung an den Anforderungen der Berechnungen berücksichtigt. Ebenfalls realisiert wurde ein inkrementeller Binder der mit dem Konzept der Bindungsgraphen sowie unterschiedlichen Bindungsvarianten, die simultan und orthogonal zu den Produkten des Übersetzers eingesetzt werden können, ein weites Spektrum an Fähigkeiten zur flexiblen und effizienten Verwaltung maschinen-interpretierbarer Programme anbietet. Auf dieser Grundlage wird die inkrementelle Systemkonstruktion verwirklicht und die Flexibilität V-PK-Managements, ohne konstanten Mehraufwand während der Ausführung, erheblich erhöht. Im Bereich des Speichermanagements wurden schließlich Realisierungsarbeiten erklärt, welche die Integration und Flexibilität des realisierten V-PK-Managements nutzen, um signifikante qualitative Verbesserungen bei der Verwaltung multipler Ausführungsfäden in einem gemeinsamen virtuellen Adreßraum zu erzielen.

# Kapitel 8

# Fallbeispiele

Zur Evaluation der Leistungsfähigkeit der entwickelten Konzepte und Verfahren, sowohl in quantitativer als auch qualitativer Hinsicht, wurden und werden schritthaltend mit der Realisierung zahlreiche Versuche durchgeführt. Aus dem Spektrum dieser Experimente werden in diesem Kapitel zum Abschluß der vorliegenden Arbeit exemplarisch drei Versuche und die dabei ermittelten Ergebnisse dargestellt. Mit dem ersten ausgewählten Experiment wird der geringe Mehraufwand demonstriert, der durch den sprachbasierten Ansatz u nd das flexibilisierte V-PK-Management gegenüber herkömmlichen, optimierten Systemen besteht. Das zweite Experiment zeigt einerseits das hohe Abstraktionsniveau, auf dem qualitativ hochwertige verteilte Problemlösungen mit den INSEL-Konzepten spezifiziert werden können und andererseits die Effizienz der Ausführung auf einer NOW-Konfiguration, die trotz der vollständigen Opazität der Verteilung erzielt wird. Mit dem letzten Experiment wird weiteres Potential für Leistungssteigerungen aufgezeigt, das aus der Integration der Managementinstanzen entsteht. Dabei wird ein einfaches Beispiel verwendet, in dem zum einen Informationstransfers von dem Übersetzer an das laufzeitbegleitende Management und zum anderen Flexibilität bei der inkrementellen Festlegung von Realisierungspfaden genutzt wird. Die systematische Nutzung des Leistungspotentials, das dabei aufgezeigt wird, ist eine interessante Perspektive und Herausforderung für weiterführende Arbeiten.

Die ausführlichen Meßdaten, sowie eine Beschreibung der Hardwarekonfiguration, auf der die Experimente durchgeführt wurden, finden sich im Anhang C.

## 8.1 INSEL im Vergleich mit C

Die Analysen bezüglich der quantitativen Leistungsfähigkeit verteilter Systeme in Abschnitt 2.2.2 ergaben, daß häufig bereits konstanter lokaler Mehraufwand (Faktor $l$) die erwünschte, absolute Verkürzung von Ausführungszeiten durch den Einsatz eines verteilten Managementsystems und Nutzung verteilter Ressourcen verhindert. Dies gilt in besonders hohem Maße für sprachbasierte Ansätze, die oft zwar geeignete Konzepte auf hohem Abstraktionsniveau zur Verfügung stellen, aber aufgrund inadäquater Managementkonzepte, meist durch Aufschichtung auf unmodifiziert übernommenen Konzepten, unbefriedigende quantitative Leistung erzielen und damit einige wichtige Beweggründe für verteiltes Rechnen in Frage stellen. Bestätigt wurden diese Überlegungen durch die Beobachtung anderer sprachbasierter Ansätze sowie den Erfahrungen mit den prototypischen Realisierungen der MoDiS-Architektur in Kapitel 4. Die Entwicklung des vollständig angepaßten Übersetzers und die Abstimmung der

Maßnahmen zwischen den Transformatoren und den interpretierenden Maßnahmen der eng an die Ausführung gekoppelten Instanzen des V-PK-Managements ermöglichte es, den konstanten lokalen Mehraufwand stark zu reduzieren.

### 8.1.1 Versuchsbeschreibung

Zur Ermittlung des lokalen Mehraufwandes des V-PK-Managements gegenüber herkömmlichen Systemen wurden einige Programme jeweils in der Sprache INSEL und in C implementiert, mit dem entsprechenden Übersetzer in unterschiedlichen Optimierungsstufen übersetzt, nacheinander mehrmals auf genau einem Rechner ausgeführt und die Ausführungszeiten dabei gemessen. Als Plattform für alle Ausführungen diente in eine SUN V9 UltraSparc I mit 166Mhz Taktung, 128MB Arbeitsspeicher und dem Betriebssystem SUN Solaris 2.5.1. Bei diesen und den noch folgenden Versuchen wurde stets darauf geachtet, daß während der Ausführung eines Testprogrammes keine anderen UNIX-Prozesse auf der Testmaschine nennenswerte Rechenleistung konsumierten. Im Folgenden werden aus diesem Kontext die Versuche *Gauss* und *SeqMatrix* diskutiert.

**SeqMatrix:** Im ersten Experiment werden sequentiell und auf naive Weise zwei $300 \times 300$ Matrizen multipliziert. Für jedes Element der Ergebnismatrix wird sowohl in INSEL als auch in C, eine FS-Order bzw. Funktion `Zeile_Mal_Spalte` ausgeführt, die als Parameter die Indizes der zu addierenden Zeile und Spalte erhält. Die Eingabematrizen sind ebenso wie die Ergebnismatrix global sichtbare Variablen in C und $\delta$-äußere DE-Komponenten in INSEL. Nebst den 90000 Unterprogrammaufrufen erfolgt die Addition und Multiplikation der Elemente in Schleifen ohne zusätzliche Unterprogrammaufrufe. Dieses Experiment vergleicht somit in erster Linie die Leistung der Zielcodes der Übersetzer für C und INSEL.

**Gauss:** In diesem Experiment wird die Erzeugung und Auflösung von Akteuren in INSEL mit Threads in C verglichen. Zusätzlich zur quantitativen Analyse wurde das Experiment so gewählt, daß gleichzeitig einige qualitative Merkmale herausgestellt werden.
Um ein Gefühl für den praktischen Nutzen der Aktivitätsträgerkonzepte zu vermitteln, werden keine parallelen Berechnungen ohne Berechnungskörper abgespaltet, sondern es wird die Summe der ersten $n = 20000$ natürlichen Zahlen gebildet, wobei jeder Additionsschritt von einem eigens erzeugten Akteur bzw. Thread durchgeführt wird, der anschließend wieder terminiert, bevor in einer Schleife ein weiterer Akteur bzw. Thread für die nächste Addition gestartet wird. Die zu addierende Zahl wird den parallelen Rechnern als Eingabeparameter übergeben und das Ergebnis in einer globalen Variable (C) bzw. $\delta$-äußeren DE-Komponente (INSEL) gehalten.

Die Quelltexte des *SeqMatrix* Experiments sind im Anhang abgedruckt. Sie unterscheiden sich nur unwesentlich. In Abbildung 8.1 sind die Quelltexte des *Gauss* Experiments abgedruckt, die sich wegen der Verwendung von Threads in C gegenüber Akteuren in INSEL stark voneinander unterscheiden.

### 8.1.2 Ergebnisse

#### 8.1.2.1 Quantitativ

In Tabelle 8.1 sind die Ausführungszeiten der jeweils schnellsten von vier Ausführungen bei höchster gemeinsamer Optimierungsstufe und der daraus abgeleitete Faktor $l$ angegeben. Da

für die Übersetzung der C Variante der GNU C Übersetzer verwendet wurde, der das gleiche Back-End wie der GNU INSEL Übersetzer nutzt und eine identische Aufruf-Schnittstelle besitzt, konnten in beiden Fällen exakt die selben Optimierungsparameter „`-O3 -mcpu=v8plus`“ angegeben werden. Die besonderen Fähigkeiten des INSEL Übersetzers und der interpretierenden Managementinstanzen $M_i$, die Rechenfähigkeit von Akteuren flexibel mit einem neuen LWP oder als sequentielles Unterprogramm zu realisieren, wurde in diesem Versuch deaktiviert, so daß auch in INSEL pro Akteur ein neuer LWP erzeugt wird. Im Gegenzug wurden im C Programm ebenfalls LWPs anstelle von User-Level-Threads gewählt, um Vergleichbarkeit herzustellen.

| **Programm** | **C-Variante** | **INSEL-Variante** | $l$ |
|---|---:|---:|---|
| *SeqMatrix* | 2.4 | 2.8 | 1.167 |
| *Gauss* | 7.9 | 9.1 | 1.152 |

Tabelle 8.1: Ausführungszeiten von C und INSEL-Programmen

Bei dem Versuch *SeqMatrix* wurde für INSEL ein geringer Mehraufwand von $l \approx 17\%$ gegenüber optimiertem C festgestellt. Die Begründung hierfür ist in erster Linie der Aufwand für die Erzeugung der $\alpha$-Displays sowie der unterschiedliche Zugriff auf die Elemente der Matrizen. Während in C ohne Indirektion auf globale Variablen zugegriffen wird, greifen die `Zeile_Mal_Spalte` FS-Order in INSEL aus Schachtelungstiefe eins über ihr Display auf diese Daten zu. Einen weiteren, geringen Anteil an dem Mehraufwand hat die dynamische Überprüfung auf Kellerüberläufe, wie sie in Abschnitt 7.2.5.3 erklärt wurde, die bei diesen Versuchen aktiviert waren.

Der Mehraufwand, der bei dem Experiment *Gauss* für die Erzeugung von Akteuren im Vergleich zu Threads in C ermittelt wurde, liegt auf einem ähnlich niedrigen Niveau. Die Gründe für den Mehraufwand sind hier die Erzeugung und Initialisierung der Daten des dedizierten Manageranteils $M_{dz}$, die bei jeder Erzeugung eines Akteures zu leisten sind.

Insgesamt zeigen diese Versuche, daß auch die lokale Leistungsfähigkeit des auf verteilte Ausführung vorbereiteten V-PK-Managements sehr hoch ist. Trotz der zahlreichen Zusatzmaßnahmen, wie der Kellerüberprüfung und der dedizierten Erzeugung von Managern entsteht nur ein sehr geringer Mehraufwand. Damit ist eine Basis geschaffen, auf der de facto Gewinne aus der Nutzung räumlich verteilter Rechen- und Speicherkapazitäten erzielt werden können.

#### 8.1.2.2 Qualitativ

Bei der Spezifikation paralleler Algorithmen, wie im Falle *Gauss*, werden die Vorteile von INSEL auf Ebene der Problemlösungen augenfällig. Betrachtet man die Quellcodes in Abbildung 8.1, so fällt zunächst auf, daß das C-Programm bei vergleichbarer Funktionalität deutlich länger als das entsprechende INSEL-Programm ausfällt. Der Anweisungsteil, in dem in einer Schleife Akteure bzw. Threads erzeugt und wieder aufgelöst werden, weist gravierende Unterschiede auf. In INSEL ist die Spezifikation einer Akteur-Erzeugung genauso kurz und prägnant, wie ein gewöhnlicher Unterprogrammaufruf. Die Erzeugungsanweisung ist in BS-Order (`BLOCK`) gekapselt, die als DA-Komponenten Abschlußsynchronisation durchführen, wodurch gewährleistet ist, daß die Iteration erst dann fortschreitet, wenn der zuletzt erzeugte Akteur terminiert und aufgelöst ist.

```
MACTOR SYSTEM IS

NUMACTORS : CONSTANT INTEGER := 20000;
P         : CONSTANT INTEGER := 1024;

SUM : INTEGER := 0;

TYPE A_T IS ARRAY[1..P] OF INTEGER;

MACTOR paradd (A:IN A_T) IS
BEGIN
  sum := sum + A[PSIZE/2];
END;

F: A_T;

BEGIN
  FOR I IN 1..NUMACTORS LOOP
    BLOCK ForkOneActor IS
    BEGIN
      F[P/2] := i;
      paradd(F);
    END;
  END LOOP;
  OUTPUT sum;
END;
```

```
#define NUMTHREADS 20000
#define PARAMSIZE  1024

int sum = 0;

typedef int A_T[PARAMSIZE];

void *paradd(void *p)                                   (1)
{
  int *F = (int *) p;
  sum = sum + F[PARAMSIZE/2];
}

void main(int argc,char **argv)
{
   int i,*tmpf,fd = open("/dev/zero", O_RDWR);
   A_T F;
   void *thread_mem;
   thread_t t;
   for (i = 0; i < NUMTHREADS; i++)
     {
      F[PARAMSIZE/2] = i+1;
      tmpf = (int *) malloc(sizeof(A_T));
      memcpy(tmpf,F,sizeof(A_T));                       (2)
      thread_mem = (void*)                              (3)
        mmap((caddr_t) 0x50000000,
          64*1024, PROT_READ | PROT_WRITE,
          MAP_PRIVATE | MAP_FIXED,
          fd, (off_t)0);
      thr_create((void *)0x50000000,64*1024,
           &paradd,(void *)tmpf,
           THR_BOUND ,&t);
      thr_join(t,NULL,NULL);                            (4)
      munmap((caddr_t) 0x50000000,64*1024);
      free(tmpf);                                       (5)
     }
   printf("Sum is: %d\n",sum);
}
```

Abbildung 8.1: Akteurerzeugung in INSEL und Threads in C

In der Sprache C ist die Spezifikation des gleichen – noch sehr einfachen – Verhaltens ungleich schwieriger und fehlerträchtiger. C besitzt keine Möglichkeit, zusammengesetzte Komponenten als `IN`-Parameter zu übergeben. Die Separation von Wirkungsbereichen des erzeugten Threads und seines Erzeugers bezüglich der Argumente muß statt dessen durch explizite Anfertigung von Kopien vor Aufruf des Threads bewerkstelligt werden. Dies erfolgt an der Markierung (2) und der Zeile darüber. An der Position (5) muß dementsprechend die angefertigte Kopie explizit aufgelöst werden. Bei (3) wird Speicher für den Keller des zu erzeugenden Threads angelegt. Dieser Schritt ist diskutabel, da er nicht zwingend erfolgen müßte, sondern auch automatisiert von der Thread-Bibliothek durchgeführt werden kann. Das V-PK-Management gewährleistet allerdings automatisch die Suffizienz der Keller von Akteuren und paßt deren Größe an den tatsächlichen Bedarf an. Eine Annäherung an diese Fähigkeiten in C ist nur dann möglich, wenn auf die automatische Allokation durch die Thread-Bibliothek verzichtet und explizit Speichermanagement betrieben wird. Da in etwas anspruchsvolleren Fällen die Festlegung von Kellergrößen ohnehin sinnvoll ist, erscheint dieser Vergleich gerechtfertigt. Die Laufzeit des C-Programms bleibt von dieser Maßnahme unbeeinflußt[1]. Letztendlich muß in C mittels `join` an der Markierung (4) explizit auf die Terminierung des Threads gewartet werden, bevor weiter iteriert werden kann.

Offensichtlich besticht die Einfachheit und Prägnanz von INSEL gegenüber der wesentlich komplizierteren und fehlerträchtigeren Formulierung paralleler Algorithmen in C. Auch durch höheren Aufwand können einige qualitative Merkmale des integrierten V-PK-Management, wie die dynamische Adaption der Keller, in C nicht nachgebildet werden.

## 8.2 Das Problem des Handlungsreisenden

An diese hohe Qualität verteilter, paralleler und kooperativer Problemlösungsspezifikationen in INSEL knüpft das nächste Fallbeispiel an, in dem ein INSEL Programm zur Lösung des NP-vollständigen Problems des Handlungsreisenden auf einer NOW-Konfiguration physisch verteilt zur Ausführung gebracht wird. Die Eigenschaften von INSEL und dem V-PK-Management, die damit unter Beweis gestellt werden, sind vollständige Opazität der räumlichen Verteilung und die Arbeitsfähigkeit der Verfahren des V-PK-Managements für die physisch verteilte Ausführung von INSEL-Systemen.

Gegeben sei ein gerichteter Graph $G = (V, E)$. $V$ sei die Menge der zu besuchenden Städte und $E \subseteq V \times V$ die Menge der möglichen Wege zwischen den Städten. Die Kanten $e_{ij} \in E$ sind gewichtet mit $d(e_{ij}) :=$ „Entfernung zwischen den Städten $v_i, v_j \in V$ in der Richtung von $v_i$ nach $v_j$“. Gesucht ist das Element $h_{min}$ aus der Menge der Hamiltonzyklen $H$ in $G$, für das die Länge $l : H \to \mathbb{N}$ minimal ist, wobei für $h \in H, \quad l(h) = \sum_{\forall e \in h} d(e)$.

Zur Lösung dieses Problems wurde in INSEL ein iterative Algorithmus entwickelt, der bei seiner Ausführung $n = |V| - 1$ M-Akteure $a_1, ..., a_n$ abspaltet, die autonom und potentiell verteilt ein lokales Minimum berechnen, wobei der von $a_i$ durchsuchte Pfad mit dem Präfix $< v_1, v_{i+1} >$ beginnt. Aus den lokalen Minima der $a_i$ wird abschließend das globale Minimum ermittelt. Durch Beschränkung der Kooperation auf Parameterübergabe bei der Initiierung und Terminierung der Akteure, findet frühes Schneiden am Suchbaum nur bei der Ermittlung der lokalen Minima innerhalb der Akteursphären statt. Eine Propagation gefundener Minima über Akteurgrenzen hinweg findet in dieser Variante nicht statt. Der pragmatische Grund für

[1] Thread-Bibliotheken unternehmen analoge Schritte.

```
MACTOR IT_VISIT(Distl: IN Distances_T; Depth: IN integer; MWays: IN Path_T;
                Lenl : IN OUT integer; Pathl: IN OUT Path_T) IS
  MSum, L, Idx: Integer;
  Left: Left_T;
  P   : Path_T;                                          – temporaerer Pfad (1)
  C   : City_T;                                          – gewaehlte Stadt (2)
  Ps  : P_Solutions_T;                                   – fuer parallele Suche (3)
  Ls  : L_Solutions_T;
BEGIN
  INIT(Distl,Depth,Pathl,P,L,Left,MWays,MSum);
  FOR I IN City_T LOOP Ls[I] := Lenl; END LOOP;
  Idx := Depth+1;
  BLOCK ParSearch IS
  BEGIN
    WHILE (Idx > Depth) LOOP
      C := P[Idx];                                       – vorhergehende Wahl (4)
      IF C > 0 THEN                                      – loesche vorhergehende Wahl (5)
        Left[C] := TRUE;
        L       := L    - Distl[P[Idx-1],C];
        MSum    := MSum + MWays[C];
      END IF;
      C := C + 1; WHILE C <= NB_CITIES AND
        (NOT Left[C] OR Distl[P[Idx-1],C] = 0) LOOP
        C := C + 1;                                      – versuche naechste Moeglichkeit (6)
      END LOOP;
      IF C <= NB_CITIES THEN                             – Abstieg im Suchbaum (7)
        Left[C]:= FALSE;
        P[Idx] := C;
        L      := L + Distl[P[Idx-1],C];
        MSum   := MSum - MWays[C];
        IF L+MSum < Lenl THEN                            – Schneidekriterium (8)
          IF Idx = NB_CITIES THEN                        – Blatt erreicht, neues Minimum ? (9)
            IF Distl[P[NB_CITIES],P[1]]   /= 0 AND
               Distl[P[NB_CITIES],P[1]]+L <   Lenl THEN
               Lenl  := L + Distl[P[NB_CITIES],P[1]];
               Pathl := P;
            END IF;
          ELSIF Idx > 1 THEN Idx   := Idx + 1;
          ELSE
            Ps[C] := P;
            IT_VISIT(Distl,Idx,MWays,Ls[C],Ps[C]);       – Akteur abspalten (10)
          END IF;
        END IF;
      ELSE                                               – backtracking (11)
        P[Idx] := 0;
        Idx    := Idx - 1;
      END IF;
    END LOOP;
  END ParSearch;
  FOR I IN City_T LOOP  IF Ls[I] < Lenl THEN
                          Lenl := Ls[I];
                          Pathl:= Ps[I]; END IF;
  END LOOP;
END IT_VISIT;
```

Abbildung 8.2: Auszug aus dem Programm der INSEL-Problemlösung TSP

diese Einschränkung ist, daß zum Zeitpunkt der Durchführung dieser Versuche noch keine Realisierung eines geeigneten DSM-Systems zur Verfügung stand.

### 8.2.1 Qualitative Eigenschaften

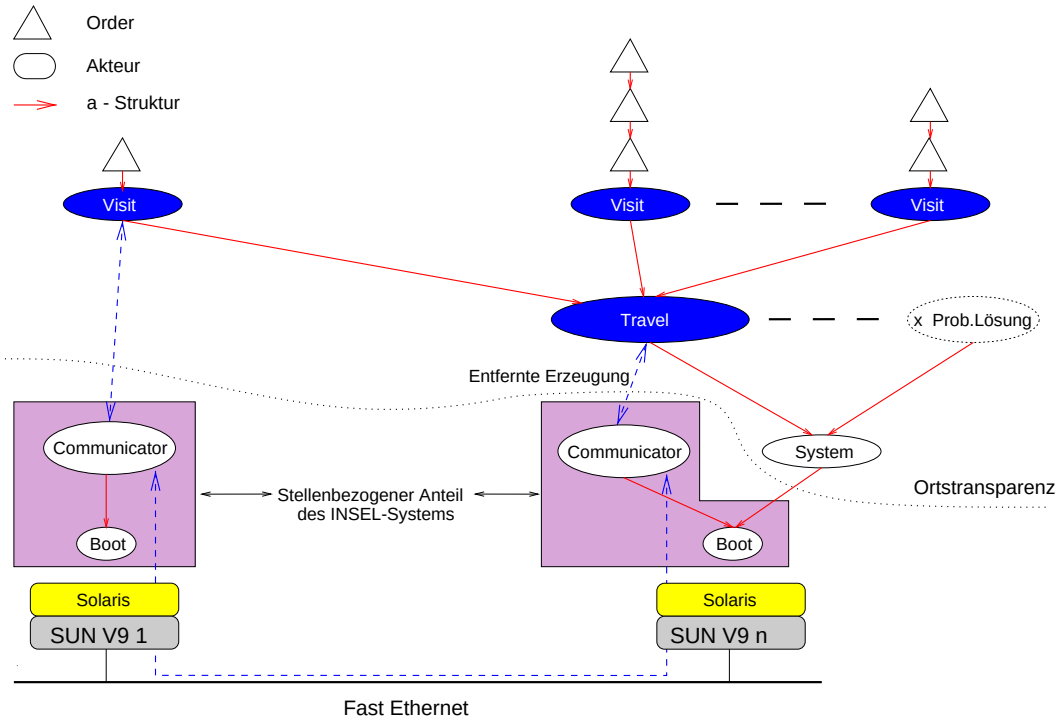


Abbildung 8.3: Verteilte Problemlösungsstruktur

In Abbildung 8.2 ist ein Auszug aus dem vollständigen Quelltext der INSEL TSP-Problemlösung abgedruckt, der das wesentliche Suchverfahren beinhaltet. Das vermutlich interessanteste Merkmal des abgebildeten Programmes ist die Erzeugung der parallelen und möglicherweise physisch verteilt realisierten Sucher in der hervorgehobenen Zeile (10). Mit der Anweisung `IT_VISIT` wird rekursiv[2] ein M-Akteur erzeugt, der $\pi$-innen eingeordnet zum Erzeuger, d. h. parallel zu diesem, seine Berechnung vorantreibt. Von dem V-PK-Management wird über die Plazierung des Akteures auf einer Stelle entschieden. In der Problemlösungsspezifikation ist diese Verteilung völlig opak. Es ist weder eine Referenz auf eine konkrete Stelle erforderlich, noch sind irgendwelche anderweitigen Vorbereitungen, wie das Verpacken von Argumenten in Nachrichten, die über abstrakte Parallelität und Kooperation hinausgehen, zu treffen.

Abbildung 8.3 stellt einen Schnappschuß des INSEL-Systems während der Ausführung einer TSP-Problemlösung dar. Der Hauptakteur des TSP-Subsystems hat bereits $\pi$-innere M-Akteure erzeugt, die auf unterschiedlichen Stellen der Konfiguration rechnen.

[2]Der Algorithmus wurde für Versuchszwecke flexibel gestaltet. Zwischen rekursiver Erzeugung von Akteuren und iterativem, sequentiellen Rechnen kann sehr einfach variiert werden. Bei den Erläuterungen in diesem Abschnitt werden von dem ersten Such-Akteur rekursiv weitere $|V| - 1$ M-Akteure abgespaltet, die dann iterativ die Suche fortsetzen.

### 8.2.2 Leistungsdaten

Zur Quantifizierung der Leistung des INSEL-TSP wurde eine NOW-Konfiguration mit 15 SUN UltraSparc I Arbeitsplatzrechner und einem 100MBit/s FastEthernet Verbindungsnetz eingesetzt (siehe Anhang). Dabei wurden folgende Festlegungen getroffen:

$$|V| = 19, \quad \forall u, v \in V : (u, v) \in E \Leftrightarrow u \neq v, \quad \forall e \in E : d(e) \in [1, 99]$$

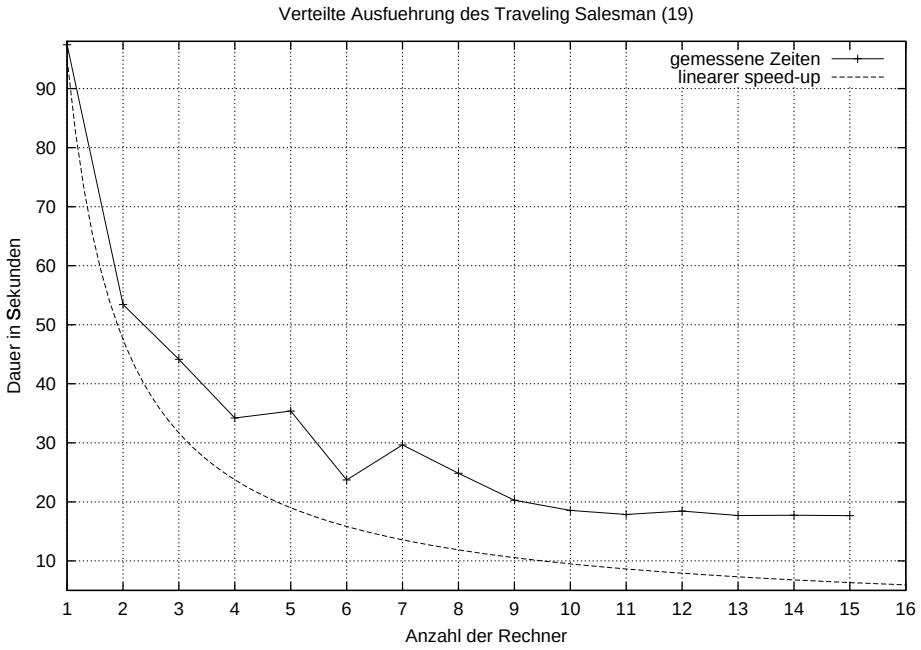


Abbildung 8.4: TSP — Beschleunigung

Durch die adaptive Anpassung der Kellergrößen mit dem Konzept der Segment-Keller sowie dem hybriden Attribut *SGranularity* (S. 201) ist der Bedarf an Speicher der iterativen TSP-Lösung äußerst gering. Pro Such-Akteur werden lediglich 32kByte Kellerspeicher benötigt. Die ermittelten Rechenzeiten sind in Abbildung 8.4 in Abhängigkeit von der Anzahl verwendeter Stellen graphisch dargestellt. Trotz der Beschränkung auf Kooperation zum Zeitpunkt der Initiierung und Terminierung verkürzen sich die Ausführungszeiten bei Hinzunahme zusätzlicher Stellen signifikant, wobei der Einsatz von mehr als zehn Rechnern keine Gewinne mehr bringt. Der Grund dafür befindet sich in dem Algorithmus selbst. Dadurch, daß keine Information zwischen den Suchern ausgetauscht wird, muß stets auf den am längsten rechnenden Akteur gewartet werden. Dies ist auch mitverantwortlich für den unerwarteten Anstieg der Rechenzeit bei sieben Stellen. In diesem ungünstigen Fall zeigt sich eine Schwäche der bislang sehr einfach gehaltenen Lastverteilung, die zwei lang rechnende Akteure auf eine gemeinsame Stelle plaziert.

Dessen ungeachtet demonstriert dieses Beispiel die Gewinne, die mit dem V-PK-Management mit wenig Aufwand seitens der Anwendungen durch das hohe Abstraktionsniveau von INSEL erzielt werden. Dabei ist zu beachten, daß nach den Vergleichen von C und INSEL die dargestellte Beschleunigung als absolute Beschleunigung gegenüber einem vergleichbaren, optimierten C-Programm auf einer Stelle zu bewerten ist.

## 8.3 Flexible Realisierung der Rechenfähigkeit

Das INSEL-Beispielprogramm aus dem Experiment *Gauss* (s. o.), bei dessen Ausführung 20000 feinstgranulare Akteure erzeugt werden, wurde in weiterführenden Experimenten herangezogen, um weitere Erkenntnisse über die Integration transformatorischer und interpretierender Managementmaßnahmen und die Abstufung von Realisierungspfaden zu sammeln.

Offensichtlich ist die Erzeugung eines Kernel-Threads bzw. LWP für jeden individuellen Akteur wenig sinnvoll. In einer Programmiersprache mit niedrigem Abstraktionsniveau, wie C mit multi-threading, würde ein Programmierer bereits bei der Spezifikation des Algorithmus die Entscheidung treffen, keine neuen Threads zu erzeugen, sondern stattdessen sequentielle Unterprogramme zu verwenden. Dies ist allerdings eine realisierungsbedingte Entscheidung, die nicht auf Eigenschaften des parallelen Problemlösungsverfahren beruht[3] sondern Ressourcenmanagement ist. Das Problem dieser Vorgehensweise ist, daß das Treffen dieser Entscheidung in weniger trivialen Fällen sehr schwierig wird, umfangreiches Wissen erfordert und sich zudem die Entscheidungsgrundlage, z. B. Kosten für die Erzeugung von Threads, laufend ändert. Dies ist somit ein typisches Beispiel für das Problem der Festlegung geeigneter Verbindlichkeitsintervalle. Die Entscheidung, die von dem Programmierer bei der Formulierung des Programms für die eine oder andere Alternative getroffen wird, hat in aller Regel eine wesentlich längere Verbindlichkeit als die Grundlage, auf der sie getroffen wurde!

Mit dem V-PK-Management wird deshalb langfristig ein anderer Weg angestrebt. Parallele und kooperative Problemlösungen sollen auf hohem Abstraktionsniveau mit der Parallelität spezifiziert werden, über die der abstrakte Algorithmus verfügt. Es ist die Aufgabe des V-PK-Managements aus den Berechnungen, mit ggf. sehr feiner Zeit-Granularität, Einheiten zu bilden, die sich für die lokale oder entfernte Erzeugung von Threads bzw. LWPs eignen. Dabei wird das gesamte Instrumentarium des V-PK-Managements eingesetzt um hohe Leistung zu erzielen. Die Notwendigkeit und das Potential der Integration kann im Hinblick auf diese hohen Anforderungen mit den bisher realisierten Verfahren bereits demonstriert werden.

Für die Realisierung der Rechenfähigkeit von Akteuren sieht das Instrumentarium drei Alternativen vor:

- Entfernte Erzeugung eines Kontrollflusses
- Lokale Erzeugung eines Kontrollflusses
- Weiterverwendung eines existierenden Kontrollflusses („sequentiell“)

Die Produktion dieser Realisierungsalternativen erfolgt durch alternative Zielcode-APIBs des Übersetzers, deren flexible Bindung durch den Binder FLink, Maßnahmen im dedizierten Anteil $M_{dz}$ der AS-Manager und der Bereitstellung von LWPs durch das gemeinsamen Management $M_{sh}$. Dieses Instrumentarium wird genutzt, um angepaßt an die Qualität der Information, zum frühestmöglichen Zeitpunkt eine Entscheidung zu treffen und dadurch möglichst viele Maßnahmen transformatorisch statt interpretierend durchzuführen.

In Abbildung 8.5 sind die möglichen Entscheidungsverläufe eingezeichnet. Aus diesem Spektrum werden hier exemplarisch zwei Varianten erklärt:

- Kann der Übersetzer auf Basis des Quelltextes und dem Attribut *TGranularity* (S. 201) noch keine Entscheidung treffen, so schiebt er die Entscheidung auf. Das heißt, er produziert eine Standardalternative, die sich sowohl für die sequentielle als auch parallele

[3] Es sei von dem einfachen Beispiel abstrahiert.

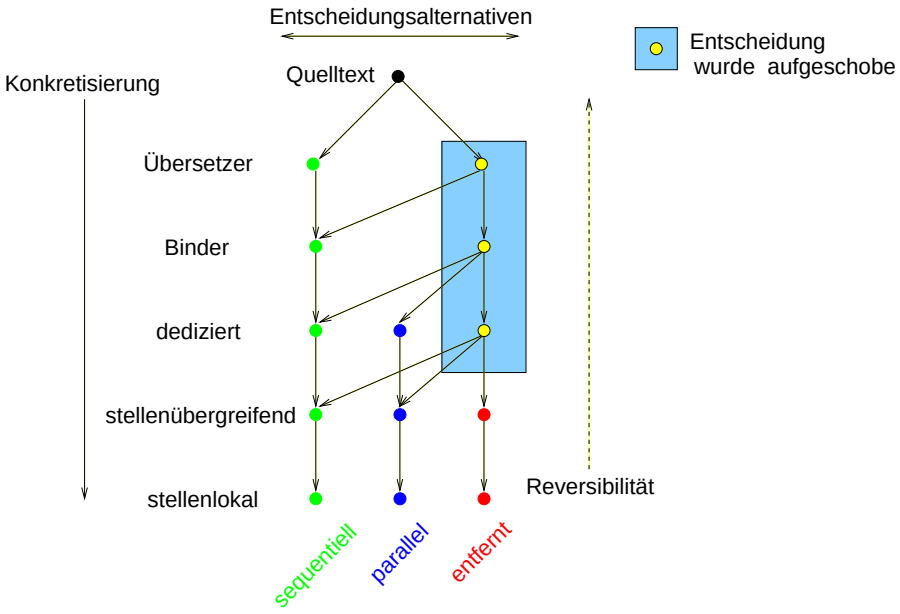


Abbildung 8.5: Abgestufte Realisierung der Rechenfähigkeit

oder verteilte Ausführung eignet und reicht die Information, die mit *TGranularity* bisher ermittelt wurde, an die folgenden Instanzen weiter. Diese reichern ihrerseits die erhaltene Information an, versuchen eine Realisierung zu treffen und reichen ihre Produkte weiter. Auf der rechten Seite der Darstellung fiel die Entscheidung spät zugunsten eines entfernten LWPs.

- Erkennt bereits der Übersetzer, daß die zu realisierende Komponente einer sehr geringe Zeit-Granularität aufweist, so trifft bereits er die Entscheidung für sequentielle Ausführung durch Produktion eines entsprechenden APIB. Diese Entscheidung ist verbindlich für alle folgenden Instanzen; siehe Entscheidungsverlauf links.

Für Experimente mit diesem Spektrum wurde der Übersetzer um zwei weitere Aufrufparameter erweitert, mit deren Hilfe separat veranlaßt werden kann, daß der Übersetzer das *TGranularity* Attribut erarbeitet und weiterreicht oder selbst eine Entscheidung trifft.

| | Durchsetzung der Entscheidung | |
|---|---|---|
| Entscheidung | Übersetzer | $M_{sh}$ |
| parallel | — | 9.1s |
| sequentiell | 0.5s | 4.8s |

Tabelle 8.2: Dauer der Berechnung — Realisierung der Rechenfähigkeit

In Tabelle 8.2 sind die Zeiten angegeben, die für die Ausführung des unveränderten Programms *Gauss* mit den neuen Fähigkeiten des V-PK-Managements für die Alternativen lokal parallel und sequentiell gemessen wurden. Der Wert rechts oben von 9.1 Sekunden entspricht der Entscheidungsalternative und Leistung, die in diesem Kapitel bereits erklärt wurde. Wird

die Informationsgewinnung des Übersetzers in das Management miteinbezogen und $M_{sh}$ entscheidet sich später für die sequentielle Ausführung statt der Erzeugung neuer LWPs, so verkürzt sich die Ausführungszeit um $\approx 50\%$. Der Grund hierfür ist, daß der Aufwand für die Erzeugung von Kontrollflüssen eingespart wird. Auf der anderen Seite bleibt der Gewinn überraschend gering, was darauf zurückzuführen ist, daß sämtliche vorbereitenden Maßnahmen, wie z. B. Aufbereitung der Übergabeparameter für möglichen Transport, bereits stattfinden, bevor die Entscheidung getroffen wird. Ist es dahingegen dem Übersetzer möglich, die Entscheidung für „sequentiell“ selbst zu treffen, so verkürzt sich die Ausführungszeit rapide auf lediglich 0.5 Sekunden!

Offensichtlich besitzt das integrierte Instrumentarium mit seiner Flexibilität ein enormes Potential für Leistungssteigerungen, das durch zusätzliche, abgestufte Verfahren, wie dem Klonen von Akteurrealisierungen [WP98] oder der Aggregation von Erzeugungsaufträgen, noch im erheblichem Umfang sinnvoll erweiterbar ist. Die systematische Nutzung dieses Potentials, des sprachbasierten, integrierten Ansatzes ist Gegenstand aktueller und weiterführender Arbeiten.

# Kapitel 9

# Schlußbetrachtung

## 9.1 Zusammenfassung und Bewertung der Ergebnisse

In der vorliegenden Arbeit wurden grundlegend neue Sichten auf das Ressourcenmanagement von Betriebssystemen erarbeitet, die es erlauben sollen, langfristig die Defizite bestehender Systeme durch eine Verbesserung der Methoden ihrer Konstruktion zu überwinden. Aus dieser veränderten Perspektive wurde exemplarisch ein leistungsfähiges Instrumentarium für verteiltes Ressourcenmanagement, das sich durch Flexibilität sowie die vertikale und horizontale Integration seiner Maßnahmen auszeichnet, entworfen, realisiert und evaluiert.

### 9.1.1 Analysen

Den Anlaß für diese Untersuchungen lieferte die zunächst naheliegende Zielsetzung, physisch verteilte Rechen- und Speicherkapazitäten, die mit **vernetzten Arbeitsplatzrechnern** nahezu ubiquitär sind, universell für verteiltes Rechnen zu nutzen. Die umfangreiche Analyse von Ansätzen, die diese globale Zielsetzung ebenfalls verfolgen, hat gezeigt, daß hierfür bislang keine zufriedenstellenden Lösungen zur Verfügung stehen. Existierende Plattformen für verteiltes Rechnen sind entweder kompliziert zu nutzen, was qualitative Defizite nach sich zieht, oder ihre quantitative Leistungsfähigkeit ist unbefriedigend. Bislang existiert kein System, das **sowohl Qualität als auch Quantität** in dem Maße verbindet, das erforderlich wäre, um verteiltes Rechnen **universell und im Standardfall** zu betreiben.

Die **Ursachen**, die hierfür identifiziert wurden, sind in beiden Fällen eine Konzentration auf Details anstelle einer gesamtheitlichen Vorgehensweise. Einerseits werden mit großen Anstrengungen Mechanismen auf niedrigem Abstraktionsniveau entwickelt, die leistungsfähig sind aber ihren Nutzern eine Vielzahl von Detailmaßnahmen abverlangen, wie z. B. die komplizierte Transformation von Identifikatoren zwischen Adreßräumen. Die Entwicklungskosten von Programmen, die auf Basis dieser Konzepte entwickelt werden, sind für den Normalfall inakzeptabel hoch, so daß sich ihr Einsatz auf Sonderfälle im technisch-wissenschaftlichen Bereich beschränkt. Andererseits beschäftigen sich einige Ansätze, vor allem sprachbasierte, mit adäquaten Konzepten auf hohem Abstraktionsniveau, vernachlässigen dabei aber das Instrumentarium, das zur effizienten Realisierung der abstrakten Konzepte benötigt wird.

Als Wurzel dieser Defizite wurden drei **Einflußfaktoren** herausgearbeitet: Erstens der Mangel an **Methodik** bei der Entwicklung von Betriebssystemen. Die vielfältigen Entscheidungen, die bei dem Entwurf und der Realisierung eines solchen Systems zu treffen sind,

werden überwiegend empirisch, nach dem *trial & error* Prinzip oder schlichtweg durch Übernahme existierender Verfahren ohne Kenntnis der Konsequenzen getroffen. Dabei ist zu bemerken, daß in einer räumlich verteilten Umgebung bereits minimale Abweichungen von der Ideallösung erhebliche Ineffizienzen nach sich ziehen können, da die Kosten für die Kommunikation zwischen den Stellen um Größenordnungen höher sind als im lokalen Fall. Das Beispiel der statischen Verweisketten, die in zentralen Systemen (siehe Übersetzer S. 207) effizient, aber in einer verteilten Umgebung völlig inakzeptabel sind, demonstriert dieses allgemeine Problem auf drastische Weise. Der zweite Einflußfaktor, der die mangelnde Methodik mit verschuldet und umgekehrt von ihr geprägt wird, ist der enorme **Aufwand**, der für die Realisierung komplexer Software-Systeme generell und für Betriebssysteme im besonderen kaum mehr zu bewältigen ist und zu Kompromissen bzw. Übernahme von Existierendem zwingt. Gerade im Bereich der Systemprogrammierung sind die Realisierungsmethoden primitiv und die Entwicklungskosten besonders hoch. Der dritte Faktor aus dem Dreieck wechselseitig abhängiger Faktoren ist die **Kreativität**, die erforderlich ist, um neue, geeignete Verfahren zu finden. Der einfache und einleuchtende Sachverhalt, daß Kreativität benötigt wird, um Neues zu konstruieren, findet in der Realität nicht die gebührende Berücksichtigung. In der Regel werden die Eigenschaften der existierenden Verfahren und Instrumente, z. B. eine einmalige Übersetzung, bei der nahezu die gesamte Analyseinformation nach erfolgter Transformation vernichtet wird, als gegeben und invariant betrachtet, obwohl es sich lediglich um Artefakte handelt, die ihre Berechtigung in einem spezifischen Kontext besitzen, in einem anderen allerdings in Frage zu stellen sind.

Anhand dieser und weiterer Beobachtungen wurden die **Anforderungen** an das im Rahmen der vorliegenden Arbeit zu entwickelnde, verteilte Managementsystem präzisiert und eine Vorgehensweise für dessen Konstruktion abgeleitet. Aus dem Spektrum der vielfältigen Motive für verteiltes Rechnen, das von Hochleistungsrechnen über Mobilität bis hin zu Tele- und Gruppenarbeit reicht, wurde zunächst eine Vorauswahl getroffen, um ein konkretes Realisierungsziel erreichen zu können. Da ein gewisses Maß an quantitativer Leistung für viele andere Zielsetzungen eine Grundvoraussetzung und ein hohes Abstraktionsniveau für die Qualität der konstruierten Systeme entscheidend ist, wurde konsequent das Ziel verfolgt, diese beiden divergierenden Aspekte zu vereinen. Das langfristig übergeordnete Ziel dieser Arbeiten ist die universelle Nutzung vernetzter Konfiguration als Plattformen für hochwertige, verteilte, parallele und kooperative Problemlösungen (**V-PK-Systeme**). Die Analyse verwandter Arbeiten hat deutlich gezeigt, daß dieses Ziel nicht durch Detailverbesserungen, sondern nur durch eine **gesamtheitliche, integrierende Vorgehensweise** erreicht werden kann, wobei der Realisierungsaufwand zunächst so weit wie möglich in den Hintergrund gerückt werden muß, um kreativen Freiraum für die erforderlichen Entwicklungsschritte zu schaffen.

In dem Projekt **MoDiS**, das sich durch die Kombination von Modell- und Top-Down–Orientierung, Sprachbasierung und gesamtheitlichem Vorgehen auszeichnet, wird genau diese Haltung eingenommen. Die imperative, objekt-basierte Sprache **INSEL** und die MoDiS-Management-Grobarchitektur wurden als Startpunkt für die eigenen Entwicklungsarbeiten gewählt. Ein INSEL-System besteht aus einer strukturierten Menge paralleler und kooperativer Berechnungen, den Akteuren. Jedem Akteur ist genau ein **Manager** assoziiert, der sämtliche Aufgaben, die im Zusammenhang mit der Realisierung des Abstraktums „Akteur“ stehen, transparent für den Akteur erfüllt. Manager kooperieren gemäß den strukturellen Abhängigkeiten zwischen den Akteuren und bilden zusammen das Gesamtmanagement, bzw. das Betriebssystem des abstrakt und physisch verteilten INSEL-Systems. Auf der gesamten Konfiguration gibt es genau ein solches System, das durch genau ein Programm beschrieben ist.

### 9.1.2 Synthese

Im Verlauf der vorliegenden Arbeit hat sich immer wieder gezeigt, wie unerwartet schwierig das Schaffen der geforderten Freiräume bereits auf gedanklicher Ebene ist. Die Entwicklung einer Vorstellung von dem, was Ressourcenmanagement ist und wie es konstruiert werden sollte, war und ist ein schwieriger und langwieriger Prozeß. Wiederholt konnte beobachtet werden, wie vorgeprägte Sichten die erforderlichen Entwicklungsschritte behindern. Zwei sicherlich diskussionswürdige Beispiele wurden mehrfach genannt: die beschränkte Betrachtung von Betriebssystemen als Betriebssystem-Kerne und die Limitierung des Begriffs der Ressourcen auf Hardware-Ressourcen.

Die Betrachtung von **Ressourcen**, wie sie sich in der vorliegenden Arbeit entwickelt hat, ist geprägt von den vielfältigsten **Abhängigkeiten** zwischen den Komponenten eines Systems und dem Wunsch, die Menge der Komponenten inklusive ihrer Abhängigkeiten in einem uniformen Modell vollständig zu erfassen. Weiter wurde der Begriff der Ressource in der vorliegenden Arbeit von der Überlegung geprägt, daß in einem Rechensystem jede Komponente eine Ressource ist, die ihrerseits entweder ein reales Angebot der Hardware, ein definitorisch festgelegtes Nutzungskonzept oder eine gedachte Kombination aus den bereits vorhandenen Ressourcen ist.

Das Resultat dieser Eindrücke wurde in einem dreidimensionalen **Systemmodell** festgehalten, das aus abstrakten und konkreten Ressourcen, sowie **Bindungen** zwischen diesen besteht. Die **drei Dimensionen** des Modells sind Raum, Zeit und Konkretisierungsniveau, wodurch das Modell in der Lage ist, sämtliche Abhängigkeiten zwischen Ressourcen zu erfassen. Ein Rechensystem ist gemäß diesem Modell ein hoch dynamisches und kompliziertes Geflecht aus mehr oder weniger stark abhängigen Ressourcen, was der Realität gut entspricht und das Verständnis für die Aufgabe der Konstruktion eines komplexen Managementsystems schärft. Gleichzeitig wurden einige wichtige Zusammenhänge geklärt, die bei der herkömmlichen Betrachtung zweidimensionaler Architekturmodelle mit den Dimensionen Raum und Abstraktionsniveau nicht sichtbar werden. So veranschaulicht das Modell deutlich, daß der Versuch, Schichten in das System einzuziehen, unweigerlich zu langen Bindungspfaden von abstrakt nach konkret führt, die als ineffizient zu deuten sind. Für das Verständnis der weiteren Entwicklungsschritte des V-PK-Managements und insbesondere der sukzessive zu treffenden Realisierungsentscheidungen erwies sich der anschauliche Umgang mit Ressourcen und Bindungen, den dieses einfache Modell induziert, als sehr hilfreich.

Vor der Konkretisierung des V-PK-Managements mußte ein Lösungsansatz für ein Problem gefunden werden, das der **Ein-Programm-Ansatz** von MoDiS aufwirft. Das Motiv für diese Vorstellung ist, präzise **Kenntnis über alle Berechnungen**, die auf der verteilten Konfiguration ablaufen, sowie deren Abhängigkeiten zu besitzen. Dieses Wissen ist eine unerläßliche Voraussetzung um wichtige Entscheidungen, wie die Plazierung von Berechnungen im Zuge der Lastverwaltung oder die Replikation von Datenobjekten durch das Speichersubsystem, zielgerichtet betreiben zu können. Verfügbar ist diese Information nur dann, wenn das System als Ganzes (Ein-System) betrachtet und als solches präzise beschrieben ist (Ein-Programm). Damit diese Vorstellung tragfähig werden kann und dabei keine isolierten, kurzlebigen Anwendungswelten produziert werden, wurden neue Konzepte für die **inkrementelle Konstruktion** des Systems und seines Programmes entwickelt. Mit der konzeptionell verankerten, unvollständigen Spezifikation von Eigenschaften und deren dynamische Vervollständigung wurde eine Basis für die **kontrollierte Evolution** des Systems und die Gewährleistung seiner Langlebigkeit geschaffen. Relativ zu anderen Ansätzen im Bereich erweiterbarer Syste-

me wurden diese Konzepte detailliert ausgearbeitet und führen weit über das mechanistische Hinzufügen bei tabellengesteuerten Ansätzen hinaus.

Mit der Sprache INSEL, ihrer Erweiterung für die inkrementelle Systemkonstruktion und den Ressourcen- und Bindungskonzepten war der Rahmen für die Aufgaben des **V-PK-Managements** festgelegt, das anschließend schrittweise, bis zur Implementierung auf Hardwareniveau, konkretisiert wurde. In den ersten Phasen dieser Konkretisierung wurden mögliche Sichten auf das INSEL-Gesamtsystem inklusive seinem Management separiert und deren Zusammenhang geklärt. Daraufhin wurde ein **Modell-Instrumentarium** entworfen, welches die Ressourcen und Bindungen des Rechensystems auf idealisierte Weise produziert und nutzt. Das wesentliche Charakteristikum des Instrumentariums ist die Separation von einmalig **transformierenden** und repetitiv **interpretierenden** Maßnahmen, die gezielt und aufeinander abgestimmt eingesetzt werden. Dabei wird das große Leistungspotential einer engen Integration aller Verwaltungsmaßnahmen eines Betriebssystems deutlich. Für den Übergang zu einer Implementierung und um eine feinere Abstufung zu erzielen, wurde das Modell-Instrumentarium, das zunächst aus einem Übersetzer und einem Interpretierer gebildet wurde, auf das Repertoire Übersetzer, Binder, Laufzeitsystem und weitere Instanzen erweitert.

Mit dem Instrumentarium einerseits und der MoDiS-Grobarchitektur der multiplen Manager andererseits bestanden nun zwei verschiedene Sichten auf das Gesamtmanagement, deren Zusammenhang vor weiteren Entwicklungsschritten geklärt werden mußte. Gezeigt wurde, daß sich die Manager-Architektur von MoDiS und die Instrumentierung eines Managements nicht widersprechen, sondern einander sinnvoll ergänzen. Jeder Manager wird individuell durch das gesamte Spektrum der Möglichkeiten des Instrumentariums realisiert. Im einfachsten Fall ist ein Manager durch ein elementares Datum bzw. ein kurzes Code-Fragment (*inline*), das vom Übersetzer produziert wird, realisiert. Stellt dahingegen ein Akteur komplexe Anforderung, z. B. Sicherheitsanforderungen, so wird der assoziierte Manager durch das integrierte Instrumentarium individuell mit den benötigten Fähigkeiten ausgestattet und kann dann aus mehreren parallelen Kontrollflüssen des Kerns, umfangreichen Datenstrukturen des Laufzeitsystems, etc. komponiert sein. Die **flexible**, **nicht-uniforme** und **dedizierte Realisierung** des vertikalen Konzepts der **Akteur-Manager** durch das **integrierte Instrumentarium** deckt das gesamte Spektrum von extrem leichtgewichtig bis nahezu beliebig umfangreiche Funktionalität ab. Dieser Ansatz ist ein zentrales Ergebnis der vorliegenden Arbeit und bietet eine günstige Ausgangsbasis, auf deren Grundlage bei hohem Abstraktionsniveau der Programmierschnittstelle anwendungsbezogenes und effizientes Ressourcenmanagement in einer verteilten Umgebung automatisiert bzw. systemintegriert durchgeführt werden kann.

Im Zuge der folgenden, umfangreichen **Implementierungsarbeiten** kam der dritte Einflußfaktor – **Aufwand** – zum tragen. Es mußte eine Weg gefunden werden, der es erlauben würde, die im Abstrakten entworfenen Konzepte mit vertretbarem Aufwand, d. h. einer realistischen Anzahl an Personenjahren, zu einem signifikanten Anteil in die Realität umzusetzen. Das additive Hinzufügen auf eine ansonsten übernommene Plattform war aufgrund der erläuterten Beobachtungen und der gesamten Vorgehensweise klar abzulehnen. Ebenso ist eine vollständige Neuimplementierung eines gesamten Managementsystems inklusive Übersetzer, Kern und vieles mehr unrealistisch und darüber hinaus überflüssig. Die dritte und eigentlich zu bevorzugende Alternative wäre der Einsatz generativer Techniken gewesen, so wie dies im Übersetzerbau üblich ist. Wegen der unbefriedigenden Situation seitens der Methodik fehlt eine Unterstützung dieser Art im Bereich der Betriebssysteme jedoch nach wie vor. Als Kompromiß wurde die Strategie verfolgt, **im Quelltext verfügbare** und qualitativ hochwertige **Software** zu **modifizieren** und an die veränderten Anforderungen in V-PK-Systemen

anzupassen. Auf diese Weise konnte unter anderem ein vollständiger, hoch optimierender **Übersetzer** realisiert werden, der INSEL in Maschinensprache transformiert. Einen weiteren Schwerpunkt der Realisierungsarbeiten bildete der dynamische und inkrementelle **Binder** *FLink*, der simultan unterschiedliche Bindungsvarianten mit abgestufter Intensität einsetzt und damit die Freiheitsgrade des V-PK-Managements erheblich erhöht. Realisierungsalternativen mit unterschiedlichen Eigenschaften, wie z. B. Übergabe der Parameter in Registern (lokale Variante) oder Erzeugen einer Nachricht und Verpacken der Parameter (entfernte Variante), werden vom Übersetzer produziert und können durch die flexiblen Bindetechniken des Binders, von den laufzeitbegleitenden Instrumenten angepaßt an die Anforderungen, dynamisch und sehr effizient eingesetzt werden. Trotz weiterer Maßnahmen der Flexibilisierung und dem hohen Abstraktionsniveau von INSEL ist der Mehraufwand, den das V-PK-Management relativ zu hoch optimierten C Programmen verursacht, auch bei lokaler Ausführung mit ca. 20% sehr gering. Im Zuge der weiteren **Evaluierung** wurde ferner gezeigt, daß das V-PK-Management in der Lage ist, bei verteilter Ausführung absolute Beschleunigungen zu erbringen. Die ursprüngliche Zielsetzung, die Machbarkeit verteiltes Rechnen mit hoher qualitativer und quantitativer Leistung zu demonstrieren und dafür geeignete Managementkonzepte zu realisieren, wurde erreicht.

#### 9.1.2.1 Übertragbarkeit der Erkenntnisse

Die Ergebnisse der vorliegenden Arbeit sind mit Ausnahme der Realisierungsarbeiten an dem INSEL-Übersetzer nicht auf den Kontext von MoDiS beschränkt. Wenngleich MoDiS und physisch verteilte Konfigurationen den Rahmen der Untersuchungen vorgaben, sind die konzeptionellen Grundlagen – Ressourcen und Bindung, das Systemmodell, inkrementelle Systemkonstruktion und die Arbeitsprinzipien des integrierten Instrumentariums – auf andere Arbeiten im Bereich der Betriebssysteme übertragbar.

Die im Rahmen der Implementierungsarbeiten modifizierten Systemprogramme und Werkzeuge für verteiltes Rechen fließen überdies in die Weiterentwicklung dieser Verfahren ein und werden in gängigen, arbeitsfähigen Systemen eingesetzt. Ein Beispiel hierfür ist die Thread-Bibliothek des Betriebssystems Linux, die in der vorliegenden Arbeit modifiziert wurde und in künftigen Versionen von Linux voraussichtlich mit diesen Änderungen vertrieben wird.

## 9.2 Ausblick

Ohne Zweifel wurden nicht alle Fragen, die im Zuge der Entwicklung eines verteilten Ressourcenmanagements auftreten, beantwortet und es wurden neue Fragen aufgeworfen, deren Beantwortung aufgrund des konkreten Realisierungszieles im Rahmen der vorliegenden Arbeit zu einem gewissen Grad offen bleiben mußte.

Auf die ausstehende Präzisierung der Ressourcen- und Bindungskonzepte wurde bereits hingewiesen. Mit diesen Konzepten wurde ein erster Ansatz für die dringend notwendige Systematisierung der Konstruktion von Ressourcenmanagementsystemen erarbeitet, der sich bei der Erklärung einiger wichtiger Sachverhalte bereits als sehr hilfreich erwiesen hat. Es bleibt zu klären, ob dieses Modell auch außerhalb der vorliegenden Arbeit geeignet ist, um interessante Eigenschaften von Systemen zu untersuchen und adäquate Antworten auf schwierige Problemstellungen zu finden.

Ebenfalls unbeantwortet blieb die Frage nach geeigneten Strategien, die es ermöglichen würden, auf Grundlage von Information, Schwellwerten und fundierten Kriterien, systema-

tisch Ressourcenklassen, Ressourcen und Bindungen mit adäquater Raum-/Zeit-Granularität festzulegen. Ein erster Schritt wäre das Sammeln unterschiedlicher Verfahren aus dem erweiterten Bereich der Betriebssysteme und deren Klassifikation mit den beschriebenen Rahmen- und Flexibilisierungskonzepten. In der vorliegenden Arbeit wurde dies an verschiedenen Stellen u. a. für Verfahren mit *prefetching* oder *lazy* Charakteristik angedeutet.

Mit der Systematisierung geht die entscheidende Frage nach der künftigen Bewältigung des Aufwands für die Realisierung einher. Es ist schwer vorzustellen, daß bedeutende Fortschritte mit den bisherigen, primitiven Methoden der Systemprogrammierung im Bereich der Betriebssysteme Realität werden können. Elementare Funktionalität muß nach wie vor mit enorm hohen Aufwand manuell und darüber hinaus vielfach redundant entwickelt werden, selbst wenn oft nur geringe Veränderungen notwendig sind. Der Grund ist auch hier die mangelnde Systematik und Flexibilität im Umgang mit Komponenten bzw. Ressourcen und Abhängigkeiten zwischen diesen. Der Verbesserung dieser Situation ist insbesondere bei der Entwicklung von Systemprogrammen höchste Priorität beizumessen.

Abgesehen von diesen konzeptionellen und methodischen Perspektiven sind selbstverständlich weitere „aufwendige“ Realisierungsarbeiten an dem V-PK-Management zu leisten. Ein konkret geplanter Schritt ist die Integration eines leistungsfähigen Verfahrens zur flexiblen Realisierung entfernter Zugriffe auf Daten in dem gemeinsamen Ein-Adreßraum.

# Anhang A

# INSEL Syntax

Die umfassende Darstellung der Programmiersprache INSEL erfolgt in separaten technischen Berichten [RW96, Piz97]. Für das bessere Verständnis der Beispielprogramme und um der Tatsache Rechnung zu tragen, daß INSEL im Rahmen der Forschungsarbeiten kontinuierlich weiterentwickelt wird, erfolgt hier eine Kurzdarstellung der Sprache in Form eines Schnappschußes der aktuellen INSEL–Syntax.

## Reservierte Wörter

| | | | |
|---|---|---|---|
| **ACCEPT** | **ACCESS** | **AND** | **ARRAY** |
| **BEGIN** | **BLOCK** | **CACTOR** | **CASE** |
| **CONSTANT** | **DEPOT** | **ELSE** | **ELSIF** |
| **END** | **ENTRY** | **ENUM** | **EXIT** |
| **EXTERN** | **FALSE** | **FOR** | **FUNCTION** |
| **GENERIC** | **INCOMPLETE** | **LOOP** | **MACTOR** |
| **MOD** | **NEW** | **NONE** | **NOT** |
| **NULL** | **OTHERS** | **OUT** | **PROCEDURE** |
| **RECORD** | **RETURN** | **RTMANAGER** | **SELECT** |
| **SPEC** | **TERMINATE** | **THEN** | **TRUE** |
| **TYPE** | **WHEN** | **WHILE** | **WITH** |
| **XOR** | | | |

## Konkrete Syntax

Die folgende Auflistung der grammatikalischen Regeln von INSEL wurde unmittelbar aus der Spezifikation des Zerteilers generiert. Die semantischen Aktionen zur Erzeugung der Termrepräsentation aus dem konkreten Syntaxbaum sind zugunsten besserer Lesbarkeit aus dieser Darstellung entfernt.

1. <*compilation-unit*>::
   declaration-part$^{2}$
2. <*declaration-part*>::
   $\epsilon$ | declaration-part$^{2}$ declaration$^{3}$ `';'` | declaration-part$^{2}$ `error ';'`
3. <*declaration*>::
   de-generator$^{32}$ | da-generator$^{4}$ | generic-generator$^{27}$ | generic-generator-incarnation$^{30}$ | object-declaration$^{43}$

4. <*da-generator*>::
   specification-part[5] | implementation-part[9]
5. <*specification-part*>::
   interface-specification[6] | no-interface-specification[7] | function-specification[8]
6. <*interface-specification*>::
   interface-generator-type[13] `SPEC IDENTIFIER` formal-in-parameter-part[21] interface-part[15]
7. <*no-interface-specification*>::
   no-interface-generator-type[14] `SPEC IDENTIFIER` formal-parameter-part[17]
8. <*function-specification*>::
   `FUNCTION SPEC IDENTIFIER` formal-in-parameter-part[21] `RETURN` name[92]
9. <*implementation-part*>::
   interface-implementation[10] | no-interface-implementation[11] | function-implementation[12]
10. <*interface-implementation*>::
    interface-generator-type[13] `IDENTIFIER` formal-in-parameter-part[21]
    declaration-and-implementation-part[16]
11. <*no-interface-implementation*>::
    no-interface-generator-type[14] `IDENTIFIER` formal-parameter-part[17]
    declaration-and-implementation-part[16]
12. <*function-implementation*>::
    `FUNCTION IDENTIFIER` formal-in-parameter-part[21] `RETURN` name[92]
    declaration-and-implementation-part[16]
13. <*interface-generator-type*>::
    `CACTOR` | `DEPOT`
14. <*no-interface-generator-type*>::
    `MACTOR` | `PROCEDURE` | `ENTRY`
15. <*interface-part*>::
    `IS` | `IS` declaration-part[2] `END` opt-identifier[26]
16. <*declaration-and-implementation-part*>::
    `IS` declaration-part[2] `BEGIN-T` statement-part[49] `END` opt-identifier[26] | `IS EXTERN STRING-LITERAL`
17. <*formal-parameter-part*>::
    $\epsilon$ | `'('` formal-parameter-list[18] `')'`
18. <*formal-parameter-list*>::
    formal-parameter[19] | formal-parameter-list[18] `';'` formal-parameter[19] | `error ';'`
19. <*formal-parameter*>::
    identifier-list[24] `':'` parameter-mode[20] name[92]
20. <*parameter-mode*>::
    `IN` | `OUT` | `IN OUT` | $\epsilon$
21. <*formal-in-parameter-part*>::
    $\epsilon$ | `'('` formal-in-parameter-list[22] `')'`

22. <*formal-in-parameter-list*>::
formal-in-parameter[23] | formal-in-parameter-list[22] `';'` formal-in-parameter[23] | `error ';'`

23. <*formal-in-parameter*>::
identifier-list[24] `':'` `IN` name[92] | identifier-list[24] `':'` name[92]

24. <*identifier-list*>::
`IDENTIFIER` | identifier-list[24] `','` `IDENTIFIER`

25. <*name-list*>::
name[92] | name-list[25] `','` name[92]

26. <*opt-identifier*>::
$\epsilon$ | `IDENTIFIER`

27. <*generic-generator*>::
`GENERIC` formal-generic-parameter-part[28] da-generator[4]

28. <*formal-generic-parameter-part*>::
$\epsilon$ | formal-generic-parameter-part[28] formal-generic-parameter[29] `';'` |
formal-generic-parameter-part[28] `error ';'`

29. <*formal-generic-parameter*>::
`WITH IDENTIFIER` | `WITH FUNCTION IDENTIFIER` formal-parameter-part[17] `RETURN` name[92] |
`WITH PROCEDURE IDENTIFIER` formal-parameter-part[17]

30. <*generic-generator-incarnation*>::
`NEW` interface-generator-type[13] `IDENTIFIER IS` name[92] actual-generic-parameter-part[31]

31. <*actual-generic-parameter-part*>::
$\epsilon$ | `'('` name-list[25] `')'`

32. <*de-generator*>::
`TYPE IDENTIFIER IS` type-part[45]

33. <*type-constructor*>::
array-type-constructor[34] | enumeration-constructor[35] | pointer-type-constructor[36] |
range-constructor[37] | record-type-constructor[40]

34. <*array-type-constructor*>::
`ARRAY '['` range-part-list[38] `']' OF` type-part[45]

35. <*enumeration-constructor*>::
`ENUM '('` identifier-list[24] `')'`

36. <*pointer-type-constructor*>::
`ACCESS` type-part[45]

37. <*range-constructor*>::
simple-expression[84] `RANGE-POINTS` simple-expression[84]

38. <*range-part-list*>::
range-part[39] | range-part-list[38] `','` range-part[39]

39. <*range-part*>::
name[92] | range-constructor[37]

40. <*record-type-constructor*>::
    `RECORD` field-declaration-list[41] `END RECORD`

41. <*field-declaration-list*>::
    field-declaration[42] `';'` | field-declaration-list[41] field-declaration[42] `';'`

42. <*field-declaration*>::
    identifier-list[24] `':'` type-part[45]

43. <*object-declaration*>::
    identifier-list[24] `':'` constant-part[44] type-part[45] init-part[48]

44. <*constant-part*>::
    $\epsilon$ | `CONSTANT`

45. <*type-part*>::
    name[92] actual-parameter-part[46] | type-constructor[33]

46. <*actual-parameter-part*>::
    $\epsilon$ | `'('` expression-list[47] `')'`

47. <*expression-list*>::
    expression[82] | expression-list[47] `','` expression[82]

48. <*init-part*>::
    $\epsilon$ | `ASSIGN-OP` expression[82]

49. <*statement-part*>::
    $\epsilon$ | statement-part[49] statement[50] `';'` | statement-part[49] `error ';'`

50. <*statement*>::
    da-related-statement[51] | compound-statement[52] | simple-statement[53]

51. <*da-related-statement*>::
    call-statement[54] | accept-statement[55] | select-statement[56] | block-statement[62]

52. <*compound-statement*>::
    loop-statement[63] | if-statement[66] | case-statement[70]

53. <*simple-statement*>::
    assignment[76] | return-statement[77] | exit-statement[78] | incomplete-statement[79] | empty-statement[80]

54. <*call-statement*>::
    name[92] actual-parameter-part[46]

55. <*accept-statement*>::
    `ACCEPT IDENTIFIER`

56. <*select-statement*>::
    `SELECT` select-alternatives[57] select-else-part[61] `END SELECT`

57. <*select-alternatives*>::
    select-alternative[58] | select-alternatives[57] `OR` select-alternative[58]

58. <*select-alternative*>::
    when-part[59] accept-alternative[60] | when-part[59] `TERMINATE ';'`

59. <*when-part*>::
    $\epsilon$ | `WHEN` condition[81] `DO`

60. <*accept-alternative*>::
    accept-statement[55] `';'` statement-part[49]

61. <*select-else-part*>::
    $\epsilon$ | `ELSE` statement-part[49]

62. <*block-statement*>::
    `BLOCK-T IDENTIFIER IS` declaration-part[2] `BEGIN-T` statement-part[49] `END` opt-identifier[26]

63. <*loop-statement*>::
    label-part[64] for-while-part[65] `LOOP` statement-part[49] `END LOOP` opt-identifier[26]

64. <*label-part*>::
    $\epsilon$ | `IDENTIFIER ':'`

65. <*for-while-part*>::
    $\epsilon$ | `FOR IDENTIFIER IN` range-part[39] | `WHILE` condition[81]

66. <*if-statement*>::
    `IF` condition[81] `THEN` statement-part[49] elsif-part[67] else-part[69] `END IF`

67. <*elsif-part*>::
    $\epsilon$ | elsif-part[67] elsif[68]

68. <*elsif*>::
    `ELSIF` condition[81] `THEN` statement-part[49]

69. <*else-part*>::
    $\epsilon$ | `ELSE` statement-part[49]

70. <*case-statement*>::
    `CASE` expression[82] `IS` case-alternatives[71] `END CASE`

71. <*case-alternatives*>::
    case-alternative[72] | case-alternatives[71] case-alternative[72]

72. <*case-alternative*>::
    `WHEN` choices-or-others[73] `DO` statement-part[49]

73. <*choices-or-others*>::
    choices[74] | `OTHERS`

74. <*choices*>::
    choice[75] | choices[74] `'|'` choice[75]

75. <*choice*>::
    expression[82] | range-constructor[37]

76. <*assignment*>::
    name[92] `ASSIGN-OP` expression[82]

77. <*return-statement*>::
    `RETURN` expression[82]

78. <*exit-statement*>::
    `EXIT` opt-identifier[26]

79. <*incomplete-statement*>::
    `INCOMPLETE`

80. <*empty-statement*>::
    `NULL-T`

81. <*condition*>::
    expression[82]

82. <*expression*>::
    relation[83] | expression[82] logical-operator[94] relation[83]

83. <*relation*>::
    simple-expression[84] | simple-expression[84] relational-operator[96] simple-expression[84]

84. <*simple-expression*>::
    term[85] | simple-expression[84] adding-operator[97] term[85] | simple-expression[84] bit-operator[95] term[85]

85. <*term*>::
    factor[86] | term[85] multiplying-operator[98] factor[86]

86. <*factor*>::
    operand[87] | factor[86] exponentiating-operator[99] operand[87]

87. <*operand*>::
    primary[88] | unary-operator[100] operand[87]

88. <*primary*>::
    literal[89] | variable-or-function-call[91] | generating-expression[93] | `'('` expression[82] `')'`

89. <*literal*>::
    `CHARACTER-LITERAL` | `STRING-LITERAL` | `INTEGER-LITERAL` | `REAL-LITERAL` | boolean-literal[90] | `NULL-T`

90. <*boolean-literal*>::
    `TRUE-T` | `FALSE-T`

91. <*variable-or-function-call*>::
    name[92] actual-parameter-part[46]

92. <*name*>::
    `IDENTIFIER` | name[92] `'.'` `IDENTIFIER` | name[92] `DEREF` | name[92] `'['` expression-list[47] `']'` | `RTMANAGER`

93. <*generating-expression*>::
    `NEW` name[92] actual-parameter-part[46]

94. <*logical-operator*>::
    `AND` | `OR` | `XOR`

95. <*bit-operator*>::
    `BIT-AND` | `BIT-OR` | `BIT-SHL` | `BIT-SHR` | `BIT-XOR`

96. <*relational-operator*>::
    '=' | '<' | '>' | NE | LE | GE
97. <*adding-operator*>::
    '+' | '-' | '&'
98. <*multiplying-operator*>::
    '*' | '/' | MOD
99. <*exponentiating-operator*>::
    EXP
100. <*unary-operator*>::
    '+' | '-' | NOT | BIT-NOT

## Abstrakte INSEL Syntax

Die abstrakte Syntax spezifiziert die Baumstruktur, auf der die Attributauswertung vorgenommen wird und die im Zuge der Synthese des Front-Ends des INSEL Übersetzer *gic* in abstrakte Syntaxbäume des Back-Ends des GNU C Übersetzers *gcc* gic transformiert wird.

1. **CompUnit** ( DeclList[2] )
2. **DeclList** $*$ Decl[3]
3. **Decl** = DaGen[4] | DeGen[29] | GenGen[40] | GenGenIncarn[41] | ObjectDecl[42]
4. **DaGen** ( DefId[121] DaGenType[19] ParamList[7] Import[13] Export[16] TypePart[31] DaBody[5] )
5. **DaBody** = String[133] | DeclAndImplPart[6]
6. **DeclAndImplPart** ( DeclList[2] StatementList[47] OptUsedId[118] )
7. **ParamList** $*$ Param[8]
8. **Param** ( DeclIdList[120] ParamMode[9] TypeName[32] )
9. **ParamMode** = In[10] | Out[11] | InOut[12]
10. **In** ( )
11. **Out** ( )
12. **InOut** ( )
13. **Import** ( ImportPart[14] )
14. **ImportPart** = All[18] | TypeNameList[15]
15. **TypeNameList** $*$ TypeName[32]
16. **Export** ( ExportPart[17] )
17. **ExportPart** = All[18] | DeclIdList[120]
18. **All** ( )
19. **DaGenType** = AkteurGen[20] | DepotGen[23] | OrderGen[24]
20. **AkteurGen** = MAkteurGen[21] | KAkteurGen[22]
21. **MAkteurGen** ( )
22. **KAkteurGen** ( )

23. **DepotGen** ( )
24. **OrderGen** = SOrderGen[25] | KOrderGen[28]
25. **SOrderGen** = FSOrderGen[26] | PSOrderGen[27]
26. **FSOrderGen** ( )
27. **PSOrderGen** ( )
28. **KOrderGen** ( )
29. **DeGen** ( DefId[121] TypePart[31] )
30. **TypePartList** ∗ TypePart[31]
31. **TypePart** = TypeName[32] | DeGenType[33]
32. **TypeName** ( Name[114] ExpList[78] )
33. **DeGenType** = Array[34] | Enum[35] | Pointer[36] | Range[37] | Record[38] | Empty[125]
34. **Array** ( TypePartList[30] TypePart[31] )
35. **Enum** ( DeclIdList[120] )
36. **Pointer** ( TypePart[31] )
37. **Range** ( Exp[79] Exp[79] )
38. **Record** ∗ FieldDecl[39]
39. **FieldDecl** ( DeclIdList[120] TypePart[31] )
40. **GenGen** ( DeclId[123] DeclList[2] DaGen[4] )
41. **GenGenIncarn** ( DeclId[123] DaGenType[19] TypeName[32] TypeNameList[15] )
42. **ObjectDecl** = VarObjectDecl[43] | ConstObjectDecl[44]
43. **VarObjectDecl** ( DeclIdList[120] TypePart[31] InitPart[45] )
44. **ConstObjectDecl** ( DeclIdList[120] TypePart[31] InitPart[45] )
45. **InitPart** = Empty[125] | Exp[79]
46. **Predeclared** ( DeclList[2] )
47. **StatementList** ∗ Statement[48]
48. **Statement** = CallStatement[72] | AcceptStatement[61] | SelectStatement[54]
49. **IfStatement** ( IfRuleList[50] ElsePart[52] )
50. **IfRuleList** ∗ IfRule[51]
51. **IfRule** ( Cond[53] StatementList[47] )
52. **ElsePart** ( StatementList[47] )
53. **Cond** ( Exp[79] )
54. **SelectStatement** ( SelectList[55] ElsePart[52] )
55. **SelectList** ∗ SelectItem[56]
56. **SelectItem** ( OptCond[57] TermAcceptPart[58] )
57. **OptCond** = Empty[125] | Cond[53]
58. **TermAcceptPart** = Terminate[59] | AcceptPart[60]

59. **Terminate** ( )
60. **AcceptPart** ( AcceptStatement[61] StatementList[47] )
61. **AcceptStatement** ( UsedId[124] )
62. **BlockStatement** ( DeclId[123] DeclList[2] StatementList[47] OptUsedId[118] )
63. **LoopStatement** ( OptDeclId[119] ForWhilePart[64] StatementList[47] OptUsedId[118] )
64. **ForWhilePart** = Empty[125] | For[65] | While[66]
65. **For** ( DeclId[123] TypePart[31] )
66. **While** ( Cond[53] )
67. **CaseStatement** ( Exp[79] CaseList[68] )
68. **CaseList** $*$ CaseItem[69]
69. **CaseItem** ( ChoiceList[70] StatementList[47] )
70. **ChoiceList** $*$ ChoiceItem[71]
71. **ChoiceItem** = Exp[79] | Range[37]
72. **CallStatement** ( Name[114] ExpList[78] )
73. **Assignment** ( Name[114] Exp[79] )
74. **ReturnStatement** ( Exp[79] )
75. **ExitStatement** ( OptUsedId[118] )
76. **IncompleteStatement** ( )
77. **EmptyStatement** ( )
78. **ExpList** $*$ Exp[79]
79. **Exp** = Literal[80] | VarOrFctAppl[81] | Operation[88] | GenExp[82]
80. **Literal** = Int[131] | Char[132] | String[133] | Real[83] | TrueVal[84] | FalseVal[85]
81. **VarOrFctAppl** ( Name[114] ExpList[78] )
82. **GenExp** ( TypeName[32] )
83. **Real** ( String[133] )
84. **TrueVal** ( )
85. **FalseVal** ( )
86. **NilVal** ( )
87. **ManagerVal** ( )
88. **Operation** ( Operator[89] ExpList[78] )
89. **Operator** ( OpType[90] File[126] LineNo[127] )
90. **OpType** = LogOpType[91] | RelOpType[92] | ArithOpType[93] | StringOpType[94]
91. **LogOpType** = AndOp[95] | OrOp[96] | XorOp[97] | NotOp[98]
92. **RelOpType** = EqOp[99] | LessOp[100] | GreaterOp[101] | NeqOp[102] | LeqOp[103] | GeqOp[104]
93. **ArithOpType** = AddOp[105] | SubOp[106] | MultOp[107] | DivOp[108] | ModOp[109]
94. **StringOpType** = ConcOp[113]

95. **AndOp** ( )
96. **OrOp** ( )
97. **XorOp** ( )
98. **NotOp** ( )
99. **EqOp** ( )
100. **LessOp** ( )
101. **GreaterOp** ( )
102. **NeqOp** ( )
103. **LeqOp** ( )
104. **GeqOp** ( )
105. **AddOp** ( )
106. **SubOp** ( )
107. **MultOp** ( )
108. **DivOp** ( )
109. **ModOp** ( )
110. **ExpOp** ( )
111. **PosOp** ( )
112. **NegOp** ( )
113. **ConcOp** ( )
114. **Name** $*$ NameItem$^{115}$
115. **NameItem** = UsedId$^{124}$ | Deref$^{116}$ | ArrayAppl$^{117}$ | ManagerVal$^{87}$
116. **Deref** ( )
117. **ArrayAppl** ( ExpList$^{78}$ )
118. **OptUsedId** = Empty$^{125}$ | UsedId$^{124}$
119. **OptDeclId** = Empty$^{125}$ | DeclId$^{123}$
120. **DeclIdList** $*$ DeclId$^{123}$
121. **DefId** = SpecId$^{122}$ | DeclId$^{123}$
122. **SpecId** ( Ident$^{130}$ File$^{126}$ LineNo$^{127}$ )
123. **DeclId** ( Ident$^{130}$ File$^{126}$ LineNo$^{127}$ )
124. **UsedId** ( Ident$^{130}$ File$^{126}$ LineNo$^{127}$ )
125. **Empty** ( )
126. **File** = Reference$^{134}$
127. **LineNo** = Int$^{131}$
128. **LimitPart** ( Import$^{13}$ Export$^{16}$ )
129. **Constant** ( )
130. **Ident** ( )

131. **Int** ( )
132. **Char** ( )
133. **String** ( )
134. **Reference** ( )

# Anhang B

# INSEL Beispielprogramme

## Das Problem des Handlungsreisenden

```
-- Verteilt iterative Loesung fuer das Problem des Handlungsreisenden
-- Author: Markus Pizka
-- $Id: anhang.tex,v 1.14 1999/06/24 11:53:32 pizka Exp $

-- In dem Programm von IT_VISIT kann an der Stelle bevor eine rekursive
-- Erzeugung stattfindet mit Idx > y festgelegt werden bis zu welcher Tiefe
-- IT-kooperative Akteure erzeugt werden und ab wann iterativ weitergerechnet
-- werden soll.

------------------------------------------------------------
-- Das Akteur fuer das Hauptprogramm
------------------------------------------------------------

MACTOR SYSTEM IS

   NB_CITIES: constant INTEGER := 19;       -- Anzahl der Staedte
   MAX_LEN  : constant INTEGER := 1000000;  -- Maximale Weglaenge (hypotetisch)

   type City_T        is 1..NB_CITIES;      -- Nutzung der Typisierung
   type Distances_T   is array[City_T,City_T]  of INTEGER;
   type Left_T        is array[City_T]         of BOOLEAN;
   type L_Solutions_T is array[City_T]         of INTEGER;
   type Path_T        is array[City_T]         of City_T;
   type P_Solutions_T is array[City_T]         of Path_T;

   ------------------------------------------------
   -- Globale Variablen
   ------------------------------------------------

   D : Distances_T;                         -- Entfernungsmatrix

   ------------------------------------------------------------
   -- Ausgabe eines Pfades und dessen Laenge
   ------------------------------------------------------------
```

```
procedure WritePath(Dist: in Distances_T; P: in Path_T; L: in Integer) is
begin
   WriteString("Shortest roundtrip:\n");
   for I in City_T loop
      WriteCardFmt("%2d ",P[I]);
   end loop;
   WriteCardFmt("%2d\n ",1);

   for I in 2..NB_CITIES loop
      WriteCardFmt("%2d+",Dist[P[I-1],P[I]]);
   end loop;
   WriteCardFmt("%2d= ",Dist[P[NB_CITIES],P[1]]);
   WriteCardFmt("%d\n",L);
end;

-------------------------------------------------------------
-- Berechne zu jeder Stadt die Distanz zur naechstliegenden Stadt
-- wird genutzt zur Abschaetzung des minimalen Restweges
-------------------------------------------------------------

procedure MinDists(Dist: in Distances_T; MWays: out Path_T) is
   M : Integer;
begin
   for I in City_T loop
      M := MAX_LEN;
      for J in City_T loop
         if Dist[I,J] < M  and
            Dist[I,J] /= 0 then
            M := Dist[I,J];
         end if;
      end loop;
      MWays[I] := M;
   end loop;
end;

-------------------------------------------------------------
-- Initialisierung der Datenstutrukturen fuer die Suche
-- ab einer gewissen Tiefe
-------------------------------------------------------------

procedure INIT(D:    in  Distances_T; Depth: in  INTEGER; Path: in Path_T;
               P:    out Path_T; L: out Integer;
               Left: out Left_T; MWays: in Path_T; MSum:  out Integer) is
begin
   L := 0;
   for I in City_T loop
      Left[I] := TRUE;
   end loop;
   for I in City_T loop
      if I <= Depth then
         P[I] := Path[I];
         Left[P[I]] := FALSE;
         if I > 1 then
```

```
            L := L + D[P[I-1],P[I]];
         end if;
      else
         P[I] := 0;
      end if;
   end loop;
   MSum := P[1];
   for I in City_T loop
      if Left[I] then
         MSum := MSum+MWays[I];
      end if;
   end loop;
end;

-------------------------------------------------------------
-- Vert./Par/Sequentielle Suche nach dem kuerzesten [Rest]weg
-------------------------------------------------------------

mactor IT_VISIT(Distl: in  Distances_T; Depth: in Integer; MWays: in Path_T;
                Lenl : in out INTEGER;  Pathl: in out Path_T) is
   MSum, L, Idx: Integer;
   Left: Left_T;
   P   : Path_T;                    -- temporaerer Pfad
   C   : City_T;                    -- gewaehlte Stadt
   Ps  : P_Solutions_T;             -- for parallel search
   Ls  : L_Solutions_T;
begin
   INIT(Distl,Depth,Pathl,P,L,Left,MWays,MSum);
   for I in City_T loop
      Ls[I] := Lenl;
   end loop;
   Idx := Depth+1;

   Block ParSearch is
   begin
      while (Idx > Depth) loop
            C := P[Idx];               -- previous selection
            if C > 0 then              -- delete prev. selection
               Left[C] := True;
               L       := L    - Distl[P[Idx-1],C];
               MSum    := MSum + MWays[C];
            end if;
            C := C + 1; while C <= NB_CITIES and
              (not Left[C] or Distl[P[Idx-1],C] = 0) loop
               C := C + 1;             -- try next possibility
            end loop;

            if C <= NB_CITIES then     -- descend
               Left[C]:= FALSE;
               P[Idx] := C;
               L      := L + Distl[P[Idx-1],C];
               MSum   := MSum - MWays[C];
               if L+MSum < Lenl then      -- bounding criteria
```

```
                  if Idx = NB_CITIES then -- reached a leaf => new minimum ?
                     if Distl[P[NB_CITIES],P[1]]   /= 0 and
                        Distl[P[NB_CITIES],P[1]]+L <  Lenl then
                        Lenl  := L + Distl[P[NB_CITIES],P[1]];
                        Pathl := P;
                        WritePath(Distl,Pathl,Lenl);
                     end if;
                  elsif Idx > 1 then
                     Idx   := Idx + 1;
                  else
                     Ps[C] := P;
                     IT_VISIT(Distl,Idx,MWays,Ls[C],Ps[C]);
                  end if;
               end if;
            else                          -- ascend
               P[Idx] := 0;
               Idx    := Idx - 1;
            end if;
      end loop;
   end ParSearch;
   for I in City_T loop
      if Ls[I] < Lenl then
         Lenl := Ls[I];
         Pathl:= Ps[I];
      end if;
   end loop;
end IT_VISIT;

-----------------------------------------------------------------
-- in einem C Unterprogramm wird die Entfernungsmatrix aufgebaut
-----------------------------------------------------------------
procedure DIST_INIT(X: in INTEGER;D: in Distances_T)
   is EXTERN "travel_dist_init";

-----------------------------------------------------------------
-- einfache Startheuristik: waehle naechsten Nachbar
-----------------------------------------------------------------
procedure StartHeuristik(D: in Distances_T;Path: out Path_T;Len: out integer) is
  Min       : Integer;
  Left      : Left_T;
  Selection : City_T;
  TmpDist   : Integer;
begin
  Len := 0;                           -- Initialisierung
  for I in City_T loop                -- alle Staedte sind zu besuchen
     Left[I] := TRUE;
  end loop;

  for Step in City_T loop             -- Besuche jede Stadt einmal
     Min := 9999999;
     if Step = 1 then                 -- beginne in der ersten Stadt
        Min       := 0;
        Selection := 1;
```

```
        else
           for City in City_T loop      -- finde den naechsten Nachbar
              TmpDist := D[Path[Step-1],City];
              if (Left[City] = TRUE) then
                 if (TmpDist < Min) then
                    Min := TmpDist;
                    Selection := City;
                 end if;
              end if;
           end loop;
        end if;
        Path[Step]      := Selection;   -- trage diesen in den Weg ein
        Left[Selection] := FALSE;       -- und nicht noch einmal besuchen
        if Step > 1 then                -- Weglaenge updaten
          Len  := Len + D[Path[Step-1],Selection];
        end if;
     end loop;
   end;

------------------------------
-- Variablen des Hauptprogrammes
------------------------------

L,I     : Integer := MAX_LEN;
P,MWays : Path_T;

------------------------------------------------------------
-- Hauptprogramm
------------------------------------------------------------
begin
   Dist_Init(NB_CITIES,D);              -- initialisiere die Entfernungsmatrix

   for I in 1..NB_CITIES loop           -- Ausgabe der Entfernungsmatrix
      for J in 1..NB_CITIES loop
         WriteInt(D[I,J]);WriteString(" ");
      end loop; Newline;
   end loop;

   Startheuristik(D,P,L);               -- verwende einfache Startheuristik
   WritePath(D,P,L);                    -- gib diesen Weg aus

   P[1] := 1;                           -- fixe Start-Stadt: 1
   MinDists(D,MWays);

   Block PSearch is begin
     IT_VISIT(D,1,MWays,L,P);           -- finde den kuerzesten Weg !!!!!!
   end PSearch;

   WritePath(D,P,L);                    -- gib diesen Weg aus
END;
```

### Eingabe und Ergebnis

Die abgedruckten Daten sind Ausgaben einer Ausführung des INSEL-Programmes zur Lösung des Problems des Handlungsreisenden.

```
19x19 distance matrix:
 0 99 60 57 71 21 31 14 78 92 89 17 79 79 49 97 14 90 36
89  0 14 81 82 43 65 44 69 70  5 31 45  2 72 63 28 74 45
59 57  0 90  8 78 85 35  7 53 14 52 59  8 93  4 97 45 18
58 75 67  0 42 48 10 24  7 55 36  5 37 92 15 49 91 14 71
66  5 26 87  0 79 80 93 61  8 20 94 68 12 85 45 23  9 87
60 99 75 74 26  0 22 30 54 56 52 10 46 34 32 54 48 94 41
 2 73 63 51 88 19  0 87 16 56 90 50 44 20 67 86 23 17  2
 9  9 76 34 46  8 76  0 50  6 70 37 16 95 64 21 17 30 28
10 47 33 95 14 28  8 95  0  7 75 98 37 66 63 55 20 14 67
71 17 46 30 42 46 22 27 27  0 48 82 38 64 73 39 68 13 19
21 52 66 36 63 71 55 47 78 80  0 65 20 60 44 53 82 57 57
62 19 10 73 77 16 21 75 43 33 47  0 83 99 82  8 15 75 88
44 33 49 64 74 25 22 82 84 52 14  8  0 98 76 80 55 98 80
26 54 89 48 36 95 48 86 42 37 49 20 36  0 84 50 77 43 88
39 38 34 97 50 56 27 44 87 30 44 46 86 64  0 13 51 19 81
22 75  5 14 43 21 76 22 31 12 31 91 50 73  3  0 47 38 48
55 74 10 81 97  7 18 73 66 27 67 30  1 41 43 58  0 94 16
44 85 46 22 76 46 20 46 44 71 17 23 77  3  1  5 34  0 23
37  2 70 30 63 87 91 96 36 14 49 71 84 39 51 92 24 38  0

Shortest roundtrip:
 1  8  6  7 19  2 14 12  3  5 10 18 15 16  4  9 17 13 11  1
 14+ 8+22+ 2+ 2+ 2+20+10+ 8+ 8+13+ 1+13+14+ 7+20+ 1+14+21= 200
```

## Sequentielle Matrixmultiplikation

Die naive sequentielle Matrixmultiplikation war eines der ersten Programme, daß von dem INSEL Übersetzer *gic* übersetzt und anschließend zur Ausführung gebracht werden konnte. In Kapitel 8 diente es als zweites Beispiel zur Demonstration des geringen lokalen Mehraufwandes des V-PK-Managements im Vergleich mit optimierten C-Programmen. Im Folgenden sind sowohl die INSEL als auch die C-Variante der Matrixmultiplikation abgedruckt.

### INSEL Programm

```
MACTOR SYSTEM is

TYPE Range_T is 1..300;
TYPE matrix_t IS ARRAY [Range_t,Range_t] OF INTEGER;
In1, In2, ErgMatrix : Matrix_T;

PROCEDURE InitErgMatrix is
begin
  for I in Range_T loop
```

```
    for J in Range_T loop
      ErgMatrix[I,J] := 0;
    end loop;
  end loop;
end;

PROCEDURE ZeileMalSpalte(Z,S: in Range_T) is
Erg,A : Integer := 0;
begin
   for I in Range_T loop
      Erg := Erg+(In1[Z,I]*In2[I,S]);
   end loop;
   ErgMatrix[Z,S] := Erg;
end;

PROCEDURE Init is
begin
  InitErgMatrix;
  for I in Range_T loop
    for J in Range_T loop
      In1[I,J] := 2*(I+J) MOD 15;
      In2[J,I] :=   (I+J) MOD 14;
    end loop;
  end loop;
end;

PROCEDURE WriteErgMatrix is
begin
  for I in Range_T loop
    for J in Range_T loop
      WriteInt(ErgMatrix[I,J]);
      WriteChar(' ');
    end loop;
    WriteChar(10);             -- new line
  end loop;
end;

begin
  Init;
  for I in Range_T loop
     for J in Range_T loop
        ZeileMalSpalte(I,J);
     end loop;
  end loop;
--  WriteErgMatrix;
--  WriteChar(10);
end;
```

## C Programm

```
#include <stdio.h>

#define RANGE  300
```

```
typedef int Matrix_T[RANGE][RANGE];

Matrix_T ErgMatrix, In1, In2;

ZeileMalSpalte(int Z, int S)
{
  int i, Erg = 0, A = 0;

  for (i=0; i < RANGE; i++)
    Erg = Erg+(In1[Z][i]*In2[i][S]);
  ErgMatrix[Z][S] = Erg;
}

int
WriteMatrix(int M[RANGE][RANGE])
{
  int I,J;

  for (I=0; I < RANGE; I++)
    {
      for (J=0; J < RANGE; J++)
        printf("%d ",M[I][J]);
      printf("\n");
    }
  printf("\n");
}

void
main(void)
{
  int I,J;

  for (I=0; I < RANGE; I++)
    for (J=0; J < RANGE; J++)
      {
        In1[I][J] = 2*(I+1+J+1) % 15;
        In2[J][I] = (I+1+J+1) % 14;
      }

  for (I=0; I < RANGE; I++)
    for (J=0; J < RANGE; J++)
        ZeileMalSpalte(I,J);
/*  WriteMatrix(ErgMatrix);  */
}
```

## Primzahlen

In diesem Beispielprogramm werden auf naive Weise die ersten $n = 200$ Primzahlen errechnet. Der Programmtext illustriert die Verwendung von Depots, dynamischen Datenstrukturen mittels anonymer Komponenten (siehe `NEW`) sowie die Akteur-Generatorfamilie `Primetest`, mit der M-Akteurgeneratoren und M-Akteure extrem feiner Granularität erzeugt werden. Dabei ist zu beachten, daß in der BS-Order `PrimeTestBlock` lediglich spezifiziert wird, daß `Primtest`-M-Akteure elementare Teilbarkeitstests für einen Kandidaten nebenläufig durchführen können. Es ist die Aufgabe des V-PK-Managements Effizienz der Ausführung zu gewährleisten. Im konkreten Fall der Primzahlentests erfordert dies die Produktion von Realisierungsalternativen durch den Übersetzer. Die benötigten Alternativen realisieren die Ausführung eines Akteurs als sequentielles Unterprogramm, eingebettet in die Ausführung des Erzeugers, oder erzeugen einen neuen Ausführungsfaden zur gemeinsamen Durchführung akkumulierter Mengen von `Primtest`-M-Akteure.

```
MACTOR System IS

PROCEDURE OutChar(c: IN CHARACTER) IS EXTERN "putchar";
PROCEDURE OutInt (c: IN INTEGER)   IS EXTERN "WriteInt";

  isPrim : boolean;                    -- shared result variable

DEPOT listmanager is                   -- Passive object generator

  TYPE listpointertype IS ACCESS listitem;

  TYPE listitem IS
    RECORD
      next : listpointertype;
      item : integer;
    END RECORD;

  TYPE myrecord IS
    RECORD
      laenge   : integer;
      listhead : listpointertype;
    END RECORD;

  help, helpto, helpact : listpointertype;
  counter : integer;
  tmp : myrecord;

  PROCEDURE PrintList IS
  BEGIN
    help := tmp.listhead;
    WHILE help /= NULL LOOP
      OutInt(help^.item);
      OutChar('>');
      help := help^.next;
    END LOOP;
    OutChar('\n');
  END PrintList;
```

```
  FUNCTION GetNextItem RETURN INTEGER IS
  BEGIN
    help := helpact;
    IF helpact^.next /= NULL THEN
      helpact := helpact^.next;
    ELSE
      helpact := tmp.listhead;
    END IF;
    RETURN (help^.item);
  END GetNextItem;

  FUNCTION Count RETURN INTEGER IS
  BEGIN
    counter := 0;
    help := tmp.listhead;
    WHILE help /= NULL LOOP
      counter := counter + 1;
      help := help^.next;
    END LOOP;
    tmp.laenge := counter;
    RETURN counter;
  END Count;

  PROCEDURE Addnext(add : IN integer) IS
  BEGIN
    helpact := tmp.listhead;
    help := tmp.listhead;
    helpto := NULL;
    WHILE help /= NULL LOOP
      helpto := help;
      help := help^.next;
    END LOOP;
    helpto^.next := NEW listpointertype;
    helpto^.next^.item := add;
    helpto^.next^.next := NULL;
    tmp.laenge := tmp.laenge + 1;
  END Addnext;

  PROCEDURE Createfirst IS
  BEGIN
    tmp.listhead := NEW listpointertype;
    tmp.listhead^.item := 2;
    tmp.listhead^.next := NULL;
    tmp.laenge := 1;
    helpact := tmp.listhead;
  END Createfirst;
BEGIN                                   -- Canonic operation of the depot
  tmp.laenge := 0;
  tmp.listhead := NULL;
END listmanager;
                                        -- Actor generator !
MACTOR Primtest(c : IN integer; p : IN integer) IS
BEGIN
```

```
  IF c MOD p = 0 THEN
    isPrim := false;
  END IF;
END Primtest;

  prim   : listmanager;                -- variable local to SYSTEM
  length : integer;
  cand   : integer := 3;
  wurzel : integer;

BEGIN
  prim.Createfirst;
  WHILE prim.Count <= 200 LOOP
    isPrim := true;
    BLOCK PrimeTestBlock is            -- Block syncing the testers
    BEGIN
      FOR i IN 1 .. prim.Count LOOP
         wurzel := prim.GetNextItem;
         IF wurzel * wurzel <= cand THEN
            Primtest(cand, wurzel);
         END IF;
      END loop;
    END PrimeTestBlock;                -- end of sync block
    IF isPrim THEN
      prim.Addnext(cand);
    END IF;
    cand := cand + 2;
  END LOOP;
  prim.PrintList;
END System;
```

# Anhang C

# Leistungsdaten

## Testumgebung

| Rechnername | Rechner, Prozessor | Hauptspeicher (MB) | Netzanbindung |
|---|---|---|---|
| sun1 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun3 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun4 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun5 | SUN Ultra 1, V9 167Mhz | 192 | 100 Mbps FastEthernet |
| sun6 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun7 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun8 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun9 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun10 | SUN Ultra 1, V9 167Mhz | 128 | 10 Mbps Ethernet |
| sun11 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun12 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun13 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun14 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun15 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |
| sun16 | SUN Ultra 1, V9 167Mhz | 128 | 100 Mbps FastEthernet |

Tabelle C.1: Konfiguration der Testumgebung

| Stufe | Option des GNU Übersetzers | Anmerkung |
|---|---|---|
| 0 | -O0 | keine Optimierung |
| 1 | -O1 | peephole, common subexpressions, Sprungopt., etc. |
| 2 | -O3 | +Inlining von Funktionen |
| 3 | -O3 -mcpu=v8plus | +Nutzung von Prozessorspezifika; z. B. Division |
| 4 | *Stufe 3* + -fno-do-stack-check | +keine Kellerüberprüfung (nur INSEL) |

Tabelle C.2: Gewählte Optimierungsstufen bei Übersetzungen

## Handlungsreisender

| **Messung/Stellen** | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|
| Messung 1 | 91.70 | 53.42 | 44.15 | 34.21 | 35.39 | 23.71 | 29.64 | 24.84 | 20.27 |
| Messung 2 | 94.72 | 53.51 | 44.16 | 34.23 | 35.41 | 23.82 | 29.64 | 24.87 | 20.29 |
| Messung 3 | 95.66 | 55.04 | 44.19 | 34.24 | 35.53 | 23.83 | 29.65 | 24.89 | 20.35 |
| Messung 4 | 97.42 | 55.69 | 44.19 | 34.36 | 35.58 | 23.88 | 29.76 | 25.00 | 20.32 |
| **Messung/Stellen** | *10* | *11* | *12* | *13* | *14* | *15* | | | |
| Messung 1 | 18.55 | 17.86 | 18.43 | 17.69 | 17.73 | 17.67 | | | |
| Messung 2 | 18.71 | 18.31 | 18.55 | 17.70 | 17.75 | 17.71 | | | |
| Messung 3 | 18.81 | 24.56 | 18.46 | 17.73 | 17.74 | 17.75 | | | |
| Messung 4 | 18.62 | 18.22 | 18.49 | 17.79 | 17.75 | 17.77 | | | |

Tabelle C.3: Laufzeiten der iterativen TSP Variante mittels IT-Kooperation in Sekunden

Die Wahl konkreter Stellenrechner aus der in Tafel C.1 erläuterten Konfiguration hatte keinen Einfluß auf die Meßergebnisse. Einerseits sind die Recheneinheiten und Hauptspeicher sämtlicher Stellen identisch und andererseits ist das Kommunikationsaufkommen im Beispiel des Traveling Salesman zu gering, als daß die geringere Bandbreite des Rechners *sun10* oder die geringfügig unterschiedliche Anbindung der einzelnen Stellen an das Netz Einfluß auf die Gesamtzeit genommen hätten.

## Sequentielle Matrixmultiplikation

Das einfache Beispiel der sequentiellen Matrixmultiplikation wurde genutzt, um einerseits den Mehraufwand von sequentiellem INSEL gegenüber C zu evaluieren und andererseits die Wichtigkeit der Optimierung des Übersetzers zu untermauern. Sämtliche Messungen in diesem Kontext wurden nacheinander auf der selben Maschine vorgenommen.

| **Messung\Optimierungsstufe** | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Messung 1 | 12.2 | 5.7 | 5.2 | 2.9 | 2.8 |
| Messung 2 | 12.9 | 6.0 | 5.2 | 2.8 | 2.7 |
| Messung 3 | 12.4 | 5.7 | 5.3 | 2.9 | 2.8 |
| Messung 4 | 12.6 | 5.6 | 5.1 | 2.9 | 2.7 |

Tabelle C.4: Laufzeiten der INSEL Matrixmultiplikation in Sekunden

| **Messung\Optimierungsstufe** | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Messung 1 | 10.7 | 5.0 | 4.6 | 2.5 |
| Messung 2 | 10.9 | 5.0 | 4.5 | 2.4 |
| Messung 3 | 10.8 | 4.9 | 4.6 | 2.4 |
| Messung 4 | 10.7 | 5.0 | 4.6 | 2.4 |

Tabelle C.5: Laufzeiten der C Matrixmultiplikation in Sekunden

## Lokale Akteur-Erzeugung

Bei Vergleichen der Kosten für die Erzeugung von 20000 Akteuren in INSEL inklusive ihrer Laufzeitmanger gegenüber 20000 Threads in einem C-Programmen wurden folgende Laufzeiten gemessen (zugehörige Programme siehe Abschnitt 8.1).

In den letzten beiden Spalten sind die Laufzeiten angegeben, die erzielt werden, wenn von der engeren Integration des V-PK-Managements Gebrauch gemacht wird. Im Falle `-fiactor-info` analysiert der Übersetzer die Zeit-Granularität der Akteurgeneratorfamilie und das Laufzeitsystem entscheidet. Im Falle `-fiactor-optimize` trifft der Übersetzer selbst die Entscheidung.

| **Messung\Sprache,Opt.** | C,0 | I,0 | C,3 | I,3 | I,3 -fiactor-info | I,3 -fiactor-optimize |
|---|---|---|---|---|---|---|
| Messung 1 | 7.9 | 9.3 | 8.0 | 9.1 | 4.5 | 0.5 |
| Messung 2 | 7.8 | 9.4 | 8.2 | 9.3 | 4.7 | 0.5 |
| Messung 3 | 8.0 | 9.3 | 7.9 | 9.3 | 4.6 | 0.5 |
| Messung 4 | 8.4 | 9.6 | 7.9 | 9.2 | 4.6 | 0.6 |

Tabelle C.6: Laufzeiten der Testprogramme Akteur-/Thread-Erzeugung in Sekunden

# Definitionsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Literaturverzeichnis


[ABG+86] Accetta, Mike, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian und Michael Young: *Mach: A New Kernel Foundation For UNIX Development.* Technischer Bericht, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, August 1986.

[ABLL92] Anderson, T. E., B. N. Bershad, E. D. Lazowska und H. M. Levy: *Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism.* In: *ACM Transactions on Computer Systems*, Seiten 53–79, February 1992.

[Abr82] Abramson, D. A.: *Computer Hardware to Support Capability Based Addressing in a Large Virtual Memory.* Doktorarbeit, Dept. of Computer Science, Monash University, 1982.

[ACD+96] Amza, C., A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu und W. Zwaenepoel: *TreadMarks: Shared Memory Computing on Networks of Workstations.* IEEE COMPUTER, 29(2):18–28, Februar 1996.

[ACD+99] Amza, C., A. L. Cox, S. Dwarkadas, L-J. Jin, K. Rajamani und W. Zwaenepoel: *Adaptive Protocols for Software Distributed Shared Memory.* Proc. of the IEEE, Special Issue on Distributed Shared Memory, 87(3):467–475, März 1999.

[Ada83] Ada: *The Programming Language Ada Reference Manual*, Band 155 der Reihe *LNCS*. Springer–Verlag, Berlin, 1983.

[AK85] Abramson, D. A. und J. L. Keedy: *Implementing a Large Virtual Memory in a Distributed Computer System.* In: *Proc. of the 18th Hawaii Int'l Conf. on System Sciences (HICSS-18)*, Seiten 515–522, Januar 1985.

[Amd67] Amdahl, G.: *The validity of the single processor approach to achieving large scale computing capabilities.* In: *AFIPS conference proceedings, Spring Joint Computing Conference*, Band 30, Seiten 483–485, 1967.

[AMM+95] Agerwala, T., J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias und M. Snir: *SP2 system architecture.* IBM Systems Journal, 34(2):152–184, 1995.

[And82] Andrews, Gregory R.: *The Distributed Programming Language SR–Mechanisms, Design and Implementation.* Software–Practice and Experience, 12:719–753, 1982.

[Ano93] ANONYMOUS: *MPI: A Message Passing Interface.* In: IEEE (Herausgeber): *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, Seiten 878–883, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.

[App97] APPLETON, BRAD: *Patterns and Software: Essential Concepts and Terminology.* World Wide Web, November 1997. http://www.enteract.com/~bradapp/docs/patterns-intro.html.

[ARD97] ANDREAS R. DISTELI, PATRIK REALI: *Combining Oberon with Active Objects.* In: *Proc. of Joint Modular Languages Conf. (JMLC). LNCS 1024*, Linz, Austria, März 1997. Springer Verlag.

[Arn86] ARNOLD, JAMES Q.: *Shared Libraries on UNIX System V.* In: USENIX ASSOCIATION (Herausgeber): *Summer conference proceedings, Atlanta 1986: June 9–13, 1986, Atlanta, Georgia, USA*, Seiten 395–404, P.O. Box 7, El Cerrito 94530, CA, USA, Summer 1986. USENIX.

[AS96] A. SKOUSEN, D. S. MILLER, R. G. FEIGEN: *The Sombrero Operating System – An Operating System for a Distributed Single Very Large Address Space.* Technischer Bericht, TR-96-005, CS Department, Arizona State University, April 1996.

[ASU86] AHO, A. V., R. SETHI und J. D. ULLMAN: *Compilers: principles, techniques, tools.* Addison-Wesley, 1986.

[B+92] BURKHART, HELMAR und OTHERS: *BAKS — Basler Algorithmen Klassifikations–Schema.* Technischer Bericht, Institut für Informatik der Universität Basel, Dezember 1992.

[BASK95] B. A. SHIRAZI, A. R. HURSON und K. M. KAVI: *Scheduling and Load Balancing in Parallel and Disributed Systems.* IEEE CS Press, 1995.

[BBD+91] BALTER, R., J. BERNADAT, D. DECOUCHANT, A. DUDA, A. FREYSSINET, S. KRAKOWIAK, M. MEYSEMBOURG, P. LE DOT, H. NGUYEN VAN, E. PAIRE, M. RIVEIL, C. ROISIN, X. ROUSSET DE PINA, R. SCIOVILLE und G. VANDÔME: *Architecture and Implementation of GUIDE, an Object–Oriented Distributed System.* Computing Systems, 4(1):31–67, Winter 1991.

[BBD+94] BERRENDORF, RUDOLF, HERIBERT C. BURG, ULRICH DETERT, RUEDIGER ESSER, MICHAEL GERNDT und RENATE KNECHT: *Intel Paragon XP/S - Architecture, Software Environment, and Performance.* Technischer Bericht, KFA-ZAM-IB-9409, KFA Research Centre, Jülich, 1994.

[BCE+94] BERSHAD, B.N., C. CHAMBERS, S. EGGERS, C. MAEDA, D. MCNAMEE, P. PARDYAK, S. SAVAGE und E.G. SIRER: *SPIN – An Extensible Microkernel for Application-specific Operating System Services.* In: *Proceedings of the 6th ACM SIGOPS European Workshop, Matching Operating Systems to Application Needs*, Seiten 68 – 71, Dagstuhl Castle, Germany, September 1994.

[BCFT95] Brandis, Marc M., R. Crelier, Michael Franz und Josef Templ: *The Oberon system family.* Software Practice and Experience, 25(12):1331–1366, Dezember 1995.

[BCSL93] Bellur, U., G. Craig, K. Shank und D. Lea: *DIAMONDS: Principles and Philosophy.* Technischer Bericht, CASE Center 9313, SONY Oswego, Dept. of Computer Science, June 1993.

[BDLS96] Bode, Arndt, Jack Dongarra, T. Ludwig und V. Sunderam (Herausgeber): *Parallel virtual machine, EuroPVM '96: third European PVM conference*, Lecture Notes in Computer Science, Munich, Germany, Oktober 1996.

[BDM+90] Brown, A.L., A. Dearle, R. Morrison, D.S. Munro und J. Rosenberg: *A Layered Persistent Architecture for Napier88.* In: *Proc. of International Workshop on Security and Persistence of Information*, Workshops in Computing, Seiten 155 – 172, Bremen, August 1990. Springer Berlin.

[Ber98] Berrendorf, Rudolph: *Benutzer- und datengesteuertes Schleifen-Scheduling auf Parallelrechnern mit Distributed Shared Memory.* Doktorarbeit, Forschungszentrum Jülich GmbH, Juni 1998.

[BFKR92] Burkhardt, H., S. Frank, B. Knobe und J. Rothnie: *Overview of the KSR1 Computer System.* Technischer Bericht, KSR-TR-9202001, Kendall Square Research, Februar 1992.

[BH96] Bancroft, Peter und Ian Hayes: *Refinement in a Type Extension Context.* Technischer Bericht, FIT-TR-96-08, Queensland University of Technology. Faculty of Information Technology, Juli 3, 1996.

[BK93] Bal, H. E. und M. F. Kaashoek: *Object Distribution in Orca using Compile-Time and Run-Time Techniques.* In: *Proc. of the Eighth Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, Seiten 162–177, September 1993.

[BK98] Buck, B. und P. Keleher: *Locality and Performance of Page- and Object-Based DSMs.* In: *Proc. of the First Merged Symp. IPPS/SPDP*, Seiten 687–693, März 1998.

[BKK+98] Bodin, F., T. Kisuki, P. Knijnenburg, M. O'Boyle und E. Rohou: *Iterative Compilation in a Non-Linear Optimisation Space.* In: *Workshop on Profile and Feedback-Directed Compilation*, Paris, France, Oktober 1998.

[BKT92] Bal, H.E., M.F. Kaashoek und A.S. Tanenbaum: *Orca: A Language for Parallel Programming of Distributed Systems.* IEEE Transactions on Software Engineering, 18(3):190–205, 1992.

[Blo92] Bloomer, John: *Power programming with RPC.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, Februar 1992.

[BMR+96] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Pattern-Oriented Software Architecture - A System of Patterns.* Wiley and Sons, 1996.

[Bro98] Broy, Manfred: *Informatik — Eine grundlegende Einführung*, Band 1. Springer Verlag, 2. Auflage, 1998.

[BS90] Baumgarten, Uwe und Peter Paul Spies: *Ansätze zu verteiltem Ada und ihre Beiträge zur Konstruktion von Verteilten Systemen.* Interner Bericht SA/90/2, Universität Oldenburg, Abteilung Systemarchitektur, Oldenburg, November 1990.

[BS98] Borghoff, U. M. und J. H. Schlichter: *Rechnergestützte Gruppenarbeit - Eine Einführung in Verteilte Anwendungen.* Springer Verlag, 2. Auflage, Oktober 1998.

[BSP+95] Bershad, B. N., S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers und C. Chambers: *Extensibility, safety and performance in the SPIN operating system.* In: *Proceedings of the 15th ACM Symposium on Operating System Principles*, Seiten 267–284, Copper Mountain Resort, Colorado, Dezember 1995.

[BST89] Bal, Henri E., Jennifer G. Steiner und Andrew S. Tanenbaum: *Programming Languages for Distributed Computing Systems.* ACM Computing Surveys, 21(3):261–322, September 1989.

[Bur98] Burd, Tom: *General Processor Information.* World Wide Web, September 1998. http://infopad.eecs.berkeley.edu/CIC.

[Byn99] Bynum, Tim: *DIPC Announcement.* World Wide Web, April 1999. http://wallybox.cei.net/dipc/.

[BZS93] Bershad, B. N., M. J. Zekauskas und W. A. Sawdon: *The Midway Distributed Shared Memory System.* In: *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON)*, Seiten 528–537, Februar 1993.

[C+92] Cahill, Vinny und others: *Supporting the Amadeus Platform on UNIX.* Technischer Bericht, TCD-CS-92-25, Distributed Systems Group, Dept. of Computer Science, Trinity College, Dublin, Ireland, July 1992.

[CAK+96] Cowan, Crispin, Tito Autrey, Charles Krasic, Carlton Pu und Jonathan Walpole: *Fast Concurrent Dynamic Linking for an Adaptive Operating System.* In: *Proceedings of the International Conference on Configurable Distributed Systems*, Annapolis, 1996.

[Car98] Carter, John B.: *Distributed Shared Memory: Past, Present, and Future.* Vortragsfolien; 3rd Int. Workshop on High-Level Parallel Programming Models and Supportive Environments, März 1998.

[CBH+94] Cahill, Vinny, Roland Balter, David Harper, Neville Harris, Xavier Rousset de Pina und Pedro Sousa: *The Comandos Distributed Application Platform.* The Computer Journal, 37(6):478–486, 1994.

[CBZ95] Carter, J. B., J. K. Bennett und W. Zwaenepoel: *Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems.* ACM Trans. on Computer Systems, 13(3):205–243, August 1995.

[CD94] Cheriton, D. R. und K. J. Duda: *A Caching Model of Operating System Kernel Functionality.* In: *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[Cha97] Charlesworth, Alan: *The Starfire SMP Interconnect.* In: ACM (Herausgeber): *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. ACM Press and IEEE Computer Society Press.

[CI93] Campbell, Roy H. und Nayeem Islam: *Choices: A Parallel Object-Oriented Operating System.* In: Agha, Gul, Peter Wegner und Akinori Yonezawa (Herausgeber): *Research Directions in Concurrent Object-Oriented Programmi ng.* MIT Press, 1993.

[CJMR89] Campbell, Roy H., Gary M. Johnston, Peter W. Madany und Vincent F. Russo: *Principles of object-oriented operating system design.* Technischer Bericht, 1510, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1989.

[CKP93] Chien, Andrew, Vijay Karamcheti und John Plevyak: *The Concert System: Compiler and Runtime Support for Fine-Grained Concurrent Object-Oriented Languages.* Technischer Bericht, R-93-1815, University of Illinois, Department of Computer Science, 1993.

[CL95] Cheung, W. H. und Anthony H. S. Loong: *Exploring Issues of Operating Systems Structuring: from Microkernel to Extensible Systems.* OSR, 29(4):4–16, Oktober 1995.

[Com92] Comandos Consortium: *The Comandos Distributed Application Platform.* 1992.

[Con96] *International Conference on Configurable Distributed Systems*, Annapolis, MD, Mai 1996.

[Cra95] Cray Research: *Silicon Graphics - Cray T90 Product Overview.* World Wide Web, 1995. http://www.sgi.com/t90/overview.html.

[Cri98] Critchlow, Terence: *A Distributed Garbage Collection Algorithm.* Technischer Bericht, UUCS-92-011, University of Utah, Department of Computer Science, Januar 2, 1998.

[Cro97] Crowley, Charles: *Operating Systems – A Design-Oriented Approach.* IRWIN, 1997.

[CT93] Clarke, David und Brendan Tangney: *Microeconomic Theory Applied to Distributed Systems.* Technischer Bericht, TCD-DSG#TCD-CS-93-30, Trinity College Dublin. Distributed Systems Group, Februar 1993.

[CV95] Chiueh, T-C. und M. Verma: *A Compiler-Directed Distributed Shared Memory System.* In: *Proc. of the 9th ACM Int'l Conf. on Supercomputing*, Seiten 77–86, Juli 1995.

[de 99] DE MELO, ALBA CRISTINA MAGALHAES ALVES: *Defining Uniform and Hybrid Memory Consistency Models on a Unified Framework.* In: *Proc. of the 32st Hawaii Int'l Conf. on System Sciences (HICSS-32) CD-ROM*, Januar 1999.

[Dea87] DEARLE, A.: *Constructing Compilers in a Persistent Environment.* In: *Proc. of 2nd Int. Workshop on Persistent Object Systems*, Appin, Scotland, 1987.

[Dea89] DEARLE, A.: *Environments: A flexible binding mechanism to support system evolution.* In: *22nd International Conference on Systems Sciences*, Seiten 46–55, Hawaii, 1989.

[Dea94] DEARLE, ALAN: *Grasshopper: An Orthogonally Persistent Operating System.* Technischer Bericht, GH-10, Universities of Sidney and Stirling, 1994.

[Deb99] DEBRAY, SAUMYA K.: *The alto Project Homepage: Link-Time Code Optimization.* World Wide Web, April 1999. http://www.cs.arizona.edu/alto/.

[Dij68] DIJKSTRA, E. W.: *The Structure of the THE Multiprogramming System.* Communications of the ACM, 11(5):341 – 346, Mai 1968.

[DJD99] DYKES, SANDRA G., CLINTON L. JEFERY und SAMIR DAS: *Taxonomy and Design Analysis for Distributed Web Caching.* In: RALPH H. SPRAGUE, JR. (Herausgeber): *Proc. of the Thirty-Second Annual Hawaii Int. Conf. on System Sciences*, Band Abstracts and CD-ROM. IEEE CS, Januar 1999.

[DKOT91] DOUGLIS, FRED, M. FRANS KAASHOEK, JOHN K. OUSTERHOUT und ANDREW S. TANENBAUM: *A Comparison of Two Distributed Operating Systems : Amoeba and Sprite.* Computing Systems, 4(4):353–384, September 1991.

[DLC+99] DWARKADAS, S., H. LU, A. L. COX, R. RAJAMONY und W. ZWAENEPOEL: *Combining Compile-Time and Run-Time Support for Efficient Software Distributed Shared Memory.* Proc. of the IEEE, Special Issue on Distributed Shared Memory, 87(3):476–486, März 1999.

[DSB86] DUBOIS, MICHEL, CHRISTOPH SCHEURICH und FAYE BRIGGS: *Memory Access Buffering in Multiprocessors.* In: *Proceedings of the 13th Annual International Symposium on Computer Architecture*, Seiten 434–442, Tokyo, Japan, Juni 2–5, 1986. IEEE Computer Society TCCA, ACM SIGARCH, and the Information Processing Society of Japan.

[Dud93] DUDEN: *Informatik*, Band 2. Dudenverlag, 2. Auflage, 1993.

[Dud97] DUDEN: *Das Fremdwörterbuch*, Band 5. Dudenverlag, 6. Auflage, 1997.

[Dum95] DUMOULIN, CEDRIC: *DREAM: A Distributed Shared Memory model using PVM.* In: HERMES (Herausgeber): *EuroPVM'95*, Band 5 der Reihe *Parallelisme, reseaux et repartition*, Seiten 155–160, September 1995.

[DW93] DAGEVILLE, BENOIT und KAM-FAI WONG: *Supporting thousands of threads using a hyrbrid stack sharing scheme.* Technischer Bericht, ECRC93-01, European Computer-Industry Research Centre, Germany, 1993.

[E+99] Eskicioglu, M. R. und others: *Evaluation of the JIAJIA Software DSM System on High Performance Computer Architectures.* In: Ralph H. Sprague, Jr. (Herausgeber): *Proc. of the Thirty-Second Annual Hawaii Int. Conf. on System Sciences*, Band Abstracts and CD-ROM, Seite 287. IEEE CS, Januar 1999.

[EAL93] Engler, Dawson R., Gregory R. Andrews und David K. Lowenthal: *Filaments : Efficient support for fine-grain parallelism.* Technischer Bericht, 93-13, University of Arizona. Dept. of Computer Science, April 1993.

[ECPR97a] Eckert, C., C. Czech, M. Pizka und N. Reimer: *Konzepte und Verfahren zur Konstruktion heteromorph paralleler Systeme.* In: *Antrag auf Weiterführung des Sonderforschungsbereiches*, SFB 342 – Methoden und Werkzeuge für die Nutzung paralleler Architekturen, Seiten 249 – 289. Technische Universität München, Sommer 1997.

[ECPR97b] Eckert, C., C. Czech, M. Pizka und N. Reimer: *Konzepte und Verfahren zur Konstruktion verteilter Systeme für parallele und kooperative Problemlösungen.* In: *Arbeits– und Ergebnisbericht 1995/1996/1997*, SFB 342 – Methoden und Werkzeuge für die Nutzung paralleler Architekturen, Seiten 209 – 244. Technische Universität München, Sommer 1997.

[EGR91] Ellis, C.A., S.J. Gibbs, und G.L. Rein: *Groupware: Some Issues and Experiences.* Communications of the ACM, 34(1):39–58, Januar 1991.

[EGV94] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Oktober 1994.

[EKO95] Engler, D. R., M. F. Kaashoek und J. O'Toole Jr.: *Exokernel: An operating system architecture for application-level resource management.* In: *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, Colorado, Dezember 1995.

[EM97] Eckert, C. und D. Marek: *Developing Secure Applications: A Systematic Approach.* In: *Proceedings of the IFIP TC11 13th International Conference on Information Security*, Kopenhagen, Denmark, Mai 1997. Chapman & Hall.

[Ens78] Enslow Jr., Philip H.: *What is a "Distributed" Data Processing System?* Computer, 1978(1):13–21, January 1978.

[EP99] Eckert, C. und M. Pizka: *Improving Resource Management in Distributed Systems using Language-Level Structuring Concepts.* The Journal of Supercomputing, 13(1):33–55, Januar 1999.

[EW95a] Eckert, C. und H.-M. Windisch: *A New Approach to Match Operating Systems to Application Needs.* In: *IASTED – ISMM International Conference on Parallel and Distributed Computing and Systems*, Seiten 499 – 503, Washington, USA, Oktober 1995.

[EW95b] Eckert, C. und H.-M. Windisch: *A Top-down Driven, Object-Based Approach to Application-Specific Operating System Design.* In: Luis-Felipe Cabrera, Marvin Theimer (Herausgeber): *Proc. of the 15th IEEE International Workshop on Object-Orientation in Operating Systems*, Seiten 153–156, Lund, Sweden, August 1995.

[FBB+97] Ford, Bryan, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin und Olin Shivers: *The Flux OSKit: A Substrate for Kernel and Language Research.* In: *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, Band 31,5 der Reihe *Operating Systems Review*, Seiten 38–51, New York, Oktober 5–8 1997. ACM Press.

[FGM+98] Fielding, R., J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach und T. Berners-Lee: *Hypertext Transfer Protocol – HTTP/1.1.* Internet Draft, IETF, HTTP Working Group, März 1998.

[FHJ94] Fleisch, B. D., R. L. Hyde und N. C. Juul: *MIRAGE+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers.* Software Practice and Experience, 24(10):887–909, Oktober 1994.

[FK97] Franz, Michael und Thomas Kistler: *Slim binaries.* Communications of the ACM, 40(12):87–94, Dezember 1997.

[FLA94] Freeh, Vincent W., David K. Lowenthal und Gregory R. Andrews: *Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations.* In: *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Seiten 201–213, 1994.

[Flo89] Floyd, Michael A.: *Turbo Pascal with objects.* Dr. Dobb's Journal of Software Tools, 14(7):56–63, 95–97, Juli 1989.

[Fly72] Flynn, M. J.: *Some Computer Organizations and Their Effectiveness.* IEEE Transactions on Computers, 21(9):948–960, 1972.

[For94] Forum, MPI: *MPI: A Message-Passing Interface.* Technischer Bericht, CSE-94-013, Oregon Graduate Institute of Science and Technology, April 1994.

[FP89] Fleisch, B. D. und G. J. Popek: *Mirage: A Coherent Distributed Shared Memory Design.* In: *Proc. of the 12th ACM Symp. on Operating Systems Principles (SOSP)*, Seiten 211–223, Dezember 1989.

[FR96] Flynn, Michael J. und Kevin W. Rudd: *Parallel Architectures.* ACM Computing Surveys, 28(1):67–70, März 1996.

[Fra94a] Franz, Michael: *Code-Generation On-the-Fly: A Key for Portable Software.* Doktorarbeit, Institute for Computer Systems, ETH Zürich, 1994.

[Fra94b] Franz, Michael: *Compiler Optimizations Should Pay for Themselves.* In: *Proceedings of the Joint Modular Languages Conference*, Seiten 111 – 121. Universitätsverlag Ulm, September 1994.

[FSGF97] Fröhlich, Norbert, Rolf Schlagenhaft, Andreas Ganz und Josef Fleischmann: *Object Orientation in Time Warp Simulation.* In: Arabnia, H. (Herausgeber): *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA'97*, Las Vegas, NV, July 1997.

[Gan97] Ganz, Andreas: *Dynamic Load Distribution for Parallel Applications*, Band 24 der Reihe *Teubner-Texte zur Informatik*, Kapitel Automatic Test Pattern Generation, Seiten 146–159. Teubner Verlag, 1997.

[GBD+94] Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek und Vaidy Sunderam: *PVM: Parallel Virtual Machine.* MIT press, 1994.

[Gel85] Gelernter, David: *Generativ Communication in Linda.* ACM Transactions on Programming Languages and Systems, 7(1):80–112, 1985.

[GG96] Gellerich, W. und M. M. Gutzmann: *Massively Parallel Programming Languages - A Classification of Design Approaches.* In: Yetongnon, K. und S. Hariri (Herausgeber): *Proceedings of the ISCA 9th International Conference on Parallel and Distributed Computing Systems*, Band 1, Seiten 110–119. ISCA, 1996.

[GHR98] G. Heiser, F. Lam und S. M. Russell: *Resource Management in the Mungi Single-Address-Space Operating System.* In: *Proc. of the Australasian Computer Science Conference*, LNCS, Seiten 417–428, Perth, Februar 1998. Springer Verlag.

[Gos91] Goscinski, A.: *Distributed Operating Systems: The Logical Design.* Addison-Wesley Publishing Company, 1991.

[GP97] Groh, S. und M. Pizka: *A Different Approach to Resource Management in Distributed Systems.* In: *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications - PDPTA*, Las Vegas, NV, July 1997.

[GPR97] Groh, S., M. Pizka und J. Rudolph: *Shadow Stacks—A Hardware-supported DSM for Objects of any Granularity.* In: *Proc. of IEEE 3rd Int'l Conf. on Algorithms & Architectures for Parallel Processing (ICA3PP'97)*, Dezember 1997.

[GPRV98] Ghormley, D. R., D. Petrou, S. H. Rodrigues und A. M. Vahdat: *GLUnix: A Global Layer Unix for a Network of Workstations.* Software Practice and Experience, 28(9):929 ff., 1998.

[GR85] Goldberg, Adele und David Robson: *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, 1985.

[GR88] Gibbons, Alan und Wojciech Rytter: *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[Gro98] Groh, Sascha: *Ein agentenbasiertes, flexibel anpassungsfähiges Ressourcenmanagement für verteilte, parallele und kooperative Systeme.* Doktorarbeit, Technische Universität München, 1998.

[GS96] Gold, C. und T. Schnekenburger: *Using the ALDY Load Distribution System for PVM Applications.* Lecture Notes in Computer Science, 1156, 1996.

[Gus88] Gustafson, J. L.: *Reevaluating Amdahl's law.* Commun. of the ACM, 31, 5:532–533, 1988.

[Gus92] Gustavson, D.: *The Scalable Coherent Interface Related Standard Projects.* IEEE Micro, 12(1):10–22, February 1992.

[GW89] Gutknecht, J. und N. Wirth: *The Oberon System.* Software Practice and Experience, 19(9):857–893, September 1989.

[HA95] Hölzle, Urs und Ole Agesen: *Dynamic versus Static Optimization Techniques for Object-Oriented Languages.* Theory and Practice of Object Sytems, 1(3):167–188, 1995.

[Han99] Hankins, Greg: *The Linux Documentation Project.* World Wide Web, 1999. http://www.go.dlr.de/linux/LDP/.

[HB96] Hassen, S. B. und H. Bal: *Integrating Task and Data Parallelism Using Shared Objects.* In: ACM (Herausgeber): *FCRC '96: Conference proceedings of the 1996 International Conference on Supercomputing: Philadelphia, Pennsylvania, USA, May 25–28, 1996*, Seiten 317–324, New York, NY 10036, USA, 1996. ACM Press.

[HERV93] Heiser, G., K. Elphinstone, S. Russell und J. Vochteloo: *Mungi: A Distributed Single Address-Space Operating System.* Technischer Bericht, SCS &E Report 9314, School of Computer Science and Engineering, University of New South Wales, November 1993.

[Hew92] Hewlett-Packard Company: *HP-UX Reference*, 3. Auflage, 1992.

[HK93] Hamilton, Graham und Panos Kougiouris: *The Spring Nucleus: A Microkernel for Objects.* In: *Proceedings of the USENIX Summer 1993 Technical Conference*, Seiten 147–160, Berkeley, CA, USA, Juni 1993. USENIX Association.

[HL93] Hogen, Guido und Rita Loogen: *A New Stack Technique for the Management of Runtime Structures in Distributed Environments.* Technischer Bericht, 93-03, RWTH Aachen, 1993.

[Hwa93] Hwang, Kai: *Advanced Computer Architecture - Parallelism, Scalability, Programmability.* McGraw-Hill Series in Computer Science, 1993.

[ID98] Ioannidis, S. und S. Dwarkadas: *Compiler and Run-Time Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems.* In: *Proc. of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Mai 1998.

[IEE95] IEEE: *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language).* IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995.

[Int97] INTEL: *MultiProcessor Specification Version 1.4*. Specification, Intel Corporation, Mai 1997.

[Iye93] IYENGAR, A.: *Parallel Dynamic Storage Allocation Algorithms*. In: *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, Seiten 82–91. IEEE Computer Society Press, Dezember 1993.

[Jek] JEKIMOV, MIKAIL: *Weiterentwicklung des inkrementellen Binders FLINK*. Systementwicklungsprojekt, Technische Universität München.

[JKW95a] JOHNSON, K. L., M. F. KAASHOEK und D. A. WALLACH: *CRL: High-Performance All-Software Distributed Shared Memory*. In: *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP)*, Seiten 213–228, Dezember 1995.

[JKW95b] JOHNSON, K. L., M. F. KAASHOEK und D. A. WALLACH: *CRL: High-Performance All-Software Distributed Shared Memory*. In: *Proc. of the Fifth Workshop on Scalable Shared Memory Multiprocessors*, Juni 1995.

[JS89] JOHNSON, MARGARET und MARK SOLINSKI: *Dynamic Link Libraries Under Microsoft Windows*. Dr. Dobb's Journal of Software Tools, 14(3):28, 29, 32, 35, 37, 82, 84, 86–91, März 1989.

[KCC+92] KIRBY, G. N. C., R. C. H. CONNOR, Q. I. CUTTS, A. DEARLE, A. M. FARKAS und R. MORRISON: *Persistent Hyper-Programs*. In: ALBANO, A. und R. MORRISON (Herausgeber): *Persistent Object Systems*, Workshops in Computing, Seiten 86–106. Springer-Verlag, 1992. 5th International Workshop on Persistent Object Systems (POS5), San Miniato, Italy.

[Ken95] KENNER, RICHARD: *Targetting and Retargetting the GNU C Compiler*. slides, November 1995.

[Kha96] KHANDEKAR, D. R.: *QUARKS: Distributed Shared Memory as a Building Block for Complex Parallel and Distributed Systems*. Diplomarbeit, Department of Computer Science, The University of Utah, März 1996.

[Kis96] KISTLER, THOMAS: *Dynamic Runtime Optimization*. Technischer Bericht, 96-54, Department of Information and Computer Science, University of California, Irvine, November 1996.

[KLS99] KARL, WOLFGANG, MARKUS LEBERECHT und MARTIN SCHULZ: *Supporting Shared Memory and Message Passing on Clusters of PCs with a SMiLE*. In: *Proc. of CANPC '99*, Orlando, FL, Januar 1999.

[Kno98] KNOOP, JENS: *Eliminating partially dead code in explicitly parallel programs*. Theoretical Computer Science, 196(1–2):365–393, April 1998.

[Koc96] KOCH, MICHAEL: *Iris - An environment for supporting collaborative writing in wide area networks*. In: *Proc. of Demonstrations at ACM Conference on CSCW'96*, Boston, MA, November 1996.

[Kou91] KOUGIOURIS, PANAGIOTIS: *A Device Management Framework for an Object Oriented Operating System.* Diplomarbeit, University of Patras, 1991.

[KR88] KERNIGHAN, BRIAN W. und DENNIS M. RITCHIE: *The C Programming Language.* Prentice–Hall, Englewood Cliffs, NJ, 2. Auflage, 1988.

[KSV96] KNOOP, JENS, BERNHARD STEFFEN und JÜRGEN VOLLMER: *Parallelism for free: efficient and optimal bitvector analyses for parallel programs.* ACM Transactions on Programming Languages and Systems, 18(3):268–299, Mai 1996.

[KvRvST93] KAASHOEK, M. F., R. VAN RENESSE, H. VAN STAVEREN und A. TANENBAUM: *FLIP: An Internetwork Protocol for Supporting Distributed Systems.* ACM Transactions on Computer Systems, 11, 1:75–106, 1993.

[Lam79] LAMPORT, LESLIE: *How to Make a Multiprocessor Computer That Correctly Executes Multi Process Programs.* IEEE Transactons on Computers, 28(9):690–691, September 1979.

[LDCZ97] LU, H., S. DWARKADAS, A. L. COX und W. ZWAENEPOEL: *Quantifying the Performance Differences between PVM and TreadMarks.* Journal of Parallel and Distributed Computing, 43(2):65–78, Juni 1997.

[Lea96] LEA, DOUG: *A Memory Allocator.* World Wide Web, Dezember 1996. http://g.oswego.edu/dl/html/malloc.html.

[Li86] LI, K.: *Shared Virtual Memory on Loosely Coupled Multiprocessors.* Doktorarbeit, Department of Computer Science, Yale University, September 1986.

[Lie95] LIEDTKE, J.: *On $\mu$-Kernel construction.* In: *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, Colorado, Dezember 1995.

[LJP93] LEA, R., C. JACQUEMOT und E. PILLEVESSE: *COOL: System Support for Distributed Programming.* Communications of the ACM, 36(9):37–46, September 1993.

[LKBT92] LEVELT, W. G., M. F. KAASHOEK, H. E. BAL und A. S. TANENBAUM: *A Comparison of Two Paradigms for Distributed Shared Memory.* Software– Practice and Experience, 22(11):985–1010, November 1992.

[LLG+89] LENOSKI, D. E., J. LAUDON, K. GHARACHORLOO, A. GUPTA, J. L. HENNESSY, M. HOROWITZ und M. LAM: *Design of the Stanford DASH multiprocessor.* Technischer Bericht, CSL-TR-89-403, Computer Systems Laboratory, Stanford University, Dezember 1989.

[LS88] LIPTON, R. J. und J. S. SANDBERG: *PRAM: A Scalable Shared Memory.* Technischer Bericht, CS-TR-180-88, Dept. of Computer Science, Princeton University, September 1988.

[LYI95] LEA, R., Y. YOKOTE und J. ITHO: *Adaptive Operating System Design Using Reflection.* In: *Proceedings of the 5th Workshop on Hot Topics on Operating Systems*, Seiten 95 – 100, 1995.

[MBC+96] Morrison, R., A. L. Brown, R. C. H. Connor, Q. I. Cutts, A. Dearle, G. N. C. Kirby und D. S. Munro: *Napier88 Reference Manual (Release 2.2.1)*. University of St Andrews, 1996.

[MBWD94] McHale, Ciaran, Séan Baker, Bridgit Walsh und Alexis Donnelly: *Synchronisation Variables*. Technischer Bericht, TCD-CS-94-01, Distributed Systems Group, Department of Computer Science, University of Dublin, 1994.

[MDPP] M., Nikolaidou, Lelis D., Anagnostopoulos P. und Georgiadis P.: *Distributed System Design: An Expert System Approach*. Technischer Bericht, TR97-0007, National and Capodistrian University of Athens. Department of Informatics.

[MH94] Mirchandaley, R. und S. Hiranandani: *Improving the Performance of DSM Systems via Compiler Involvement*. In: *Proc. of Supercomputing'94*, Seiten 763–772, November 1994.

[ML95] Matthews, D. C. J. und T. Le Sergent: *LEMMA: A Distributed Shared Memory with Global and Local Garbage Collection*. In: *Proc. of the Int'l Workshop on Memory Management (IWMM)*, Seiten 297–311, September 1995.

[MM97] Mowbray, Thomas und Raphael Malveau: *CORBA Design Patterns*. Wiley Computer Publishing, Januar 1997.

[Mös94] Mössenböck, H.: *Extensibility in the Oberon System*. Technischer Bericht, CS-SSW-P94-01, Johannes Kepler University Linz, Austria, Januar 1994.

[Muc97] Muchnick, Steven S.: *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

[MvRT+90] Mullender, S.J., G. van Rossum, A.S. Tanenbaum, R. van Renesse und H. van Staveren: *Amoeba: A Distributed Operating System for the 1990s*. IEEE Computer Magazine, Seiten 44–53, May 1990.

[NC96] Nagaratnam, N. und G. L. Craig: *Fuzzy–based Dynamic Program Reconfiguration*. In: *Florida AI Research Symposium*, Mai 1996.

[NH94] Nelson, M. N. und Graham Hamilton: *High Performance Dynamic Linking Through Caching*. Technischer Bericht, Sun Microsystems, Mountain View, CA, 1994.

[NS98] Nehmer, Jürgen und Peter Sturm: *Systemsoftware: Grundlagen moderner Betriebssysteme*. dpunkt Verlag, Heidelberg, 1998.

[Nut92] Nutt, Gary J.: *Centralized and Distributed Operating Systems*. Prentice–Hall Inc., Englewood Cliffs, N. J., 1992. ISBN 0–13–122383–6.

[Obe98] World Wide Web, 1998. http://www-cs.inf.ethz.ch/Oberon.html.

[OCDN88] Ousterhout, J. K., A. R. Cherenson, F. Douglis und M. N. Nelson B. B. Nelson: *The Sprite network operating system*. Computer, 21(2):23–36, 1988.

[OM98] Orr, Douglas B. und Robert W. Mecklenburg: *OMOS – An Object Server for Program Execution.* Technischer Bericht, UUCS-92-033, University of Utah, Department of Computer Science, Januar 2, 1998.

[OMG95] OMG: *The Common Object Request Broker: Architecture and Specification.* Technischer Bericht, Object Management Group, Juli 1995.

[OMG97] OMG: *CORBA Services:Common Object Services Specification, Security Services.* Technischer Bericht, OMG and X/Open Co Ltd., Juli 1997.

[OMHL98] Orr, Douglas B., Robert W. Mecklenburg, Peter Hoogenboom und Jay Lepreau: *Dynamic Program Monitoring and Transformation Using the OMOS Object Server.* Technischer Bericht, UUCS-92-034, University of Utah, Department of Computer Science, Januar 2, 1998.

[Ope92] Open Software Foundation: *Introduction to OSF DCE.* Technischer Bericht, Revision 1.0, Update 1.0.1, Open Software Foundation, Cambridge, MA 02142, August 1992.

[Ope96] Open Group: *DCE Release 1.2.2 New Features.* World Wide Web, 1996. http://www.opengroup.org/dce/info/papers/tog-dce-ds-1296.htm.

[Org72] Organick, Elliott I.: *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, MA, USA, 1972.

[PAB+95] Pu, C., T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole und K. Zhang: *Optimistic Incremental Specialization: Streamlining a Commercial Operating System.* In: *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, Colorado, Dezember 1995.

[Pac92] Packard, Hewlett: *Berkeley IPC Programmer's Guide*, 2. Auflage, Juli 1992.

[PBG+85] Pfister, G. F., W. C. Brantley, D. A. George, S. L. Harvey, W. L. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton und J. Weiss: *The IBM Research Parallel Prototype (RP3): Introduction and Architecture.* In: *Proc. of the 1985 Int'l Conf. on Parallel Processing (ICPP'85)*, Seiten 764–771, August 1985.

[PDZ97] Peter Druschel, Vivek S. Pai und Willy Zwaenepoel: *Extensible Kernels are Leading OS Research Astray.* In: *In Proc. of the Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, MA, Mai 1997.

[PE96] Philbin, James und Jan Edler: *Very Lightweight Threads.* In: *First International Workshop on High-Level Programming Models and Supportive Environments*, April 1996.

[PE97] Pizka, M. und C. Eckert: *A Language-Based Approach to Construct Structured and Efficient Object-Based Distributed Systems.* In: *Proc. of the 30th Hawaii Int. Conf. on System Sciences*, Band 1, Seiten 130–139, Maui, Hawai, Januar 1997. IEEE CS Press.

[PEG97] Pizka, M., C. Eckert und S. Groh: *Evolving Software Tools for New Distributed Computing Environments.* In: Arabnia, H. (Herausgeber): *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA'97*, Seiten 87–96, Las Vegas, NV, July 1997.

[PH] Poetzsch-Heffter, A.: *Programming Language Specification and Prototyping Using the MAX System.* Cornell University, NY.

[PH94] Poetzsch-Heffter, Arnd: *Übersetzung von Programmiersprachen.* Skriptum zur Vorlesung, TU München, 1994.

[Piz94] Pizka, Markus: *Implementierung eines CAN-Treibers unter LynxOS.* Diplomarbeit, Technische Universität München, Mai 1994.

[Piz97] Pizka, Markus: *Design And Implementation of the GNU INSEL-Compiler gic.* Technischer Bericht, TUM-I9713, Technische Universität München, Dept. of CS, 1997.

[Piz99a] Pizka, Markus: *Thread Segment Stacks.* In: *Proc. of PDPTA '99*, Las Vegas, NV, Juni 1999.

[Piz99b] Pizka, Markus: *Distributed Virtual Address Space Management in the MoDiS-OS.* Technischer Bericht, TUM-I9817, Technische Universität München, vor. 4. Q. 1999.

[Ple93] Pleier, Christoph: *The Distributed C Development Environment.* Technischer Bericht, TUM-I9324, Technische Universität München, September 1993.

[Pro98] *Workshop on Profile and Feedback-Directed Compilation*, Paris, France, Oktober 1998.

[PS95] Plainfossé, David und Marc Shapiro: *A Survey of Distributed Garbage Collection Techniques.* In: Baker, Henry (Herausgeber): *Proc. of Int'l Workshop on Memory Management*, Band 986 der Reihe *LNCS*, ILOG, Gentilly, France, and INRIA, Le Chesnay, France, September 1995. Springer-Verlag.

[Rad95] Radermacher, Ralph: *Eine Ausführungsumgebung mit integrierter Lastverteilung für verteilte und parallele Systeme.* Doktorarbeit, Technische Universität München, 1995.

[Rat97] Rational, Software Corporation: *Unified Modeling Language V. 1.0.1 - Summary.* Documentation available at http://www.rational.com, Rational Software Corp., 3/1997 1997.

[RBDL97] Reps, Thomas, Thomas Ball, Manuvir Das und James R. Larus: *The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem.* Technischer Bericht, CS-TR-97-1335, University of Wisconsin, Madison, Januar 1997.

[RDH+96] Rosenberg, John, Alan Dearle, David Hulse, Anders Lindström und Stephen Norris: *Operating system support for persistant and recoverable computations.* Communications of the ACM, 39(9):62–69, September 1996.

[Reh98a] Rehn, Christian: *Implementierung des FLINK.* Web Documentation of the Source Code, 1998.

[Reh98b] Rehn, Christian: *Inkrementelles und dynamisches Binden in einer verteilten Umgebung.* Diplomarbeit, Technische Universität München, Institut für Informatik, Februar 1998.

[Rei96] Reinhold, Jörg: *Implementierung des MaX SVM-Systems und Erweiterung seiner Konfigurierbarkeit.* Diplomarbeit, Technische Universität München, Institut für Informatik, März 1996.

[RHKP95] Radia, Sanjay, Graham Hamilton, Peter Kessler und Michael L. Powell: *The Spring Object Model.* In: USENIX Association (Herausgeber): *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS): June 26–29, 1995, Monterey, California, USA*, Seiten 159–172, Berkeley, CA, USA, Juni 1995. USENIX.

[Ric93] Rice University, Houston, Texas: *High Performance Fortran Language Specification*, 1.1. Auflage, Mai 93.

[RP97a] Rechenberg, P. und G. Pomberger: *Informatik-Handbuch.* Hanser, 1997.

[RP97b] Reimer, N. und M. Pizka: *Dynamic Load Distribution for Parallel Applications*, Band 24 der Reihe *Teubner-Texte zur Informatik*, Kapitel Load Distribution Strategies of the Distributed Thread Kernel DTK. Teubner Verlag, 1997.

[RP99] Rehn, C. und M. Pizka: *BOPS: Balancing Objects and Pages in a Shared Space.* In: *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, Juni 1999.

[RW96] Radermacher, R. und F. Weimer: *INSEL Syntax–Bericht.* Technischer Bericht, TU München, März 1996. SFB–Bericht 342/08/96 A TUM–I9617.

[Sch94] Schmidt, Douglas C.: *The ADAPTIVE Communication Environment: An O.O. Network Programming Toolkit for developing Communication Software.* Technischer Bericht, Comp. Science Department, Washington University, 1994.

[Sch95] Schnepel, M.: *Ein INSEL–Compiler auf der Basis von DTK.* Diplomarbeit, TU München, 1995.

[Sch99] Schulthess, Peter: *Plurix Hompage.* World Wide Web, Mai 1999. http://www-vs.informatik.uni-ulm.de/projekte/plurix.html.

[See74] Seegmueller, G.: *Einführung in die Systemprogrammierung.* Bibliographisches Institut, Mannheim, 1974.

[SEL+96] Spies, P. P., C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer und H.-M. Windisch: *Sprachkonzepte zur Konstruktion verteilter Systeme.* Technischer Bericht, TU München, März 1996. SFB–Bericht 342/09/96 A TUM–I9618.

[SESS94] Seltzer, M., Y. Endo, C. Small und K. Smith: *An Introduction to the Architecture of the VINO-kernel.* Technischer Bericht, TR-34-94, Computer Science, Harvard University, 1994.

[SESS96] Seltzer, M. I., Y. Endo, C. Small und K. A. Smith: *Dealing with Disaster: Surviving Misbehaved Kernel Extensions.* In: *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Seiten 213–227, Seattle, Washington, Oktober 1996.

[SF85] Schreiner, A. T. und G. Friedman: *Compiler bauen mit UNIX: eine Einführung.* PC professionell. HANSER, 1985.

[SGDM94] Sunderam, V. S., G. A. Geist, J. Dongarra und R. Manchek: *The PVM concurrent computing system: Evolution, experiences, and trends.* Parallel Computing, 20(4):531–545, April 1994.

[Shn96] Shnier, Mitchell: *Dictionary of PC Hardware and Data Communication Terms.* O'Reilly, April 1996.

[SL94] Shank, K. und D. Lea: *ODL: Preliminary Language Report.* Technischer Bericht, Syracuse University, Dezember 1994.

[SL95] Shasha, Dennis und Cathy Lazere: *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists.* Copernicus, New York, Inc./ 175 Fifth Avenue/New York, NY 1, 1995.

[SPE97] *SPEC CPU95 Benchmarks.* World Wide Web, März 1997. http://www.spec.org/osg/cpu95.

[SPG91] Silberschatz, A., J. Peterson und P. Galvin: *Operating System Concepts.* Addison–Wesley, 3. Auflage, 1991.

[Spi89] Spies, Peter Paul: *Verteilte Systeme – Eine noch zu nutzende Chance.* Interner Bericht SA/89/2, Universität Oldenburg, Abteilung Systemarchitektur, Oldenburg, Juli 1989.

[SRKM96] Schlichter, Johann, Ralf Reichwald, Michael Koch und Kathrin Möslein: *Telekooperation an der Technischen Universität München - Bericht aus einem Projektvorhaben an der Fakultät für Informatik.* In: *Workshop CSCW in großen Unternehmungen*, Seiten 12 – 18, Darmstadt, Germany, Mai 1996.

[SS96] Small, Christopher und Margo Seltzer: *A Comparison of OS Extension Technologies.* In: USENIX Association (Herausgeber): *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, USENIX Conference Proceedings 1996, Seiten 41–54, Berkeley, CA, USA, Januar 1996. USENIX.

[Sta95] Stallman, Richard M.: *Using and Porting GNU CC.* Free Software Foundation, November 1995.

[Str91] Stroustrup, Bjarne: *The C++ Programming Language.* Addison–Wesley, Reading, MA, 2. Auflage, 1991.

[Str96] Strobl, Christian: *Integration der Sprache INSEL in den GCC*. Fortgeschrittenenpraktikum, Technische Universität München, Juli 1996.

[Sun93] Sunderam, V.: *The PVM Concurrent Computing System*. In: Anonymous (Herausgeber): *The commercial dimensions of parallel computing: UNICOM seminar — April 1993, London*, Seiten 20–84. Unicom Seminars Ltd, 1993.

[Sun95a] Sun Microsystems, Mountain View, CA: *The Java Language Specification*, 1.0 Beta Auflage, Oktober 1995.

[Sun95b] SunSoft, Mountain View, CA: *Solaris Multithreaded Programming Guide*, 1995.

[Sun96] SunSoft, Mountain View, CA: *Linker and Libraries Guide*, 1996.

[Sun97] SunSoft: *elf - object access library*, Februar 1997. SunOS 5.5.1.

[SV95] Schmidt, Douglas C. und Steve Vinoski: *Object Interconnections*. SIGS C++ Report magazine, Mai 1995.

[SY97] Subhlok, Jaspal und Bwolen Yang: *A New Model for Integrated Nested Task and Data Parallel Programming*. In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice od Parallel Programming (PPOPP-97)*, Band 32, 7 der Reihe *ACM SIGPLAN Notices*, Seiten 1–12, New York, Juni 18–21 1997. ACM Press.

[SZ88] Sestoft, Peter und Alexander V. Zamulin: *Annotated Bibliography on Partial Evaluation and Mixed Computation*. In: Bjørner, D., A. P. Ershov und N. D. Jones (Herausgeber): *Partial Evaluation and Mixed Computation*, Seiten 589–622. North-Holland, 1988. Most recent version: May 23rd, 1991.

[Tan92] Tanenbaum, Andrew S.: *Modern Operating Systems*. Prentice Hall, New Jersey, 1992.

[Tan95] Tanenbaum, Andrew S.: *Distributed Operating Systems*. Prentice–Hall International, 1995.

[The92] The Chorus Team: *Overview of the Chorus distributed operating system*. In: *USENIX Workshop on Microkernels and other Kernel-Architectures*, Seattle, WA, 1992.

[TMC91] *The Connection Machine CM-5 Technical Summary*. Thinking Machines Corporation, Oktober 1991.

[TP99] Theel, Oliver und Markus Pizka: *Distributed Caching and Replication, Introduction to the Minitrack*. In: *Proc. of the 32st Hawaii Int'l Conf. on System Sciences (HICSS-32) CD-ROM*, Januar 1999.

[Wec89] Weck, G.: *Prinzipien und Realisierung von Betriebssystemen*. B.G. Teubner, Stuttgart, 1989.

[Weg87] WEGNER, P.: *Dimensions of Object–Based Language Design.* In: MEYROWIT, NORMAN (Herausgeber): *OOPSLA'87 Conference Proceedings*, Seiten 168–182, Orlando, Florida, October 4–8 1987.

[Wei97] WEIMER, FRANK: *DAViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL–Programmen.* Technischer Bericht, TU München, April 1997. SFB–Bericht 342/15/97 A TUM–I9721.

[WG92] WIRTH, NIKLAUS und JÜRGEN GUTKNECHT: *Project OBERON — The Design of an Operating System and Compiler.* Addison Wesley, 1992.

[WG94] WEAVER, D. L. und T. GERMOND: *The SPARC Architecture Manual, Version 9.* SUN Microsystems, Mountain View, CA, 2. Auflage, 1994.

[Wil94] WILSON, PAUL R.: *Uniprocessor Garbage Collection Techniques.* Technischer Bericht, University of Texas, Januar 1994. Expanded version of the IWMM92 paper.

[Win95] WINDISCH, H.-M.: *Improving the Efficiency of Object Invocations by Dynamic Object Replication.* In: ARABNIA, H. R. (Herausgeber): *Proc. of PDPTA*, Seiten 115 – 131, November 1995.

[Win96a] WINDISCH, H.-M.: *The Distributed Programming Language INSEL – Concepts and Implementation.* In: *First International Workshop on High-Level Programming Models and Supportive Environments*, Seiten 17 – 24, April 1996.

[Win96b] WINDISCH, HANS-MICHAEL: *Speicherverwaltung für konzeptionell strukturierte verteilte Systeme.* Doktorarbeit, Technische Universität München, 1996.

[Wir88] WIRTH, NIKLAUS: *The Programming Language Oberon.* Software Practice and Experience, 18(7), Juli 1988. The Language Report.

[WM91] WIRTH, N. und H. MOESSENBOECK: *The Programming Language Oberon-2.* Structured Programming, 12(4):179–195, April 1991.

[WM96] WILHELM, R. und D. MAURER: *Übersetzerbau: Theorie, Konstruktion, Generierung.* Springer-Verlag, Berlin, 2. Auflage, 1996.

[WO90] WILSON, W. und R. A. OLSSON: *An Approach to Genuine Dynamic Linking.* Technischer Bericht, Dept. of CS, University of Califonia, Davis, CA, Oktober 1990.

[WP98] WAY, TOM und LORI POLLOCK: *Using Path Spectra to Direct Function Cloning*, 1998. University of Delaware, Newark.

[YD96] YANG, Z. und K. DUDDY: *CORBA: A Platform for Distributed Object Computing.* Operating Systems Review, 30(2):4 – 31, April 1996.

[YMFT91] YOKOTE, Y., A. MITSUZAWA, N. FUJINAMI und M. TOKORO: *Reflective Object Management in the Muse Operating System.* In: *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, Seiten 16 – 23. IEEE Computer Society Press, 1991.

[Yok92] YOKOTE, Y.: *The Apertos reflective operating system – the concept and its implementation.* In: *Proceedings of the Conference on Object-Orientated Programming Systems, Languages and Applications (OOPSLA)*, Seiten 414 – 434. ACM Press, 1992.