TECHNISCHE UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

Architektur und Konzept des Dycos-Kerns

Christian B. Czech

TUM-I9717
SFB-Bericht Nr. 342/12/97 A
April 97

TUM–INFO–04-I9717-060/1.–FI

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

Druck: Fakultät für Informatik der
Technischen Universität München

Architektur und Konzept des Dycos-Kerns

Christian B. Czech
Institut für Informatik
Technische Universität München
D-80290 München
czech@informatik.tu-muenchen.de

April 1997

Zusammenfassung

Der im Teilprojekt A8 des SFB 342 verfolgte Ansatz beschäftigt sich mit der Konzeption eines sprachbasierten verteilten Gesamtsystems. Anwender werden damit in die Lage versetzt, parallele Problemlösungen auf einer abstrakten Sprachebene zu entwerfen und transparent auf einer verteilten Hardwarekonfiguration auszuführen. Die dafür eingesetzte Sprache INSEL verfügt über Konzepte, mit deren Hilfe sich strukturierte Mengen aktiver und passiver Komponenten bilden lassen. INSEL Komponenten werden vom System über ein Transformationsinstrumentarium aus Übersetzer, Binder und verteiltem Management auf die Stellen der Hardware abgebildet. Komponenten-spezifische Manager sorgen dabei für die Bereitstellung und Verwaltung der anwendungsseits benötigten Ressourcen und entwickeln sich adaptiv mit der Anforderungen der Anwendung weiter. Die so entstehende angepaßte verteilte Managementarchitektur stellt sehr vielschichtige und veränderliche Anforderungen an die hardwarenahe Ressourcenverwaltung auf den jeweiligen Stellen. Herkömmliche Stellenkerne, wie sie in klassischen Client-Server Systemen eingesetzt werden, können diese Anforderungen nicht oder nur sehr eingeschränkt erfüllen. Ein für diese Systeme geeigneter Stellenkern sollte ein Instrumentarium bereitstellen, das anstelle der in herkömmlichen Systemen anzutreffenden statischen Kerndienste ein Konzept zur dynamischen Entwicklung und Anpassung von Basisdiensten beinhaltet. Dies schließt die Verankerung der Fähigkeit zur funktionalen Erweiterbarkeit mit ein.

Der *Dynamic Context Switching (Dycos-)* Kern bietet ein dafür geeignetes Basisinstrumentarium. Mit Hilfe der elementaren Kernoperationen (Toolset) lassen sich mit Dycos funktionale Einheiten unterschiedlicher Granularität baukastenartig kombinieren und damit neue Kernabstraktionen schaffen. Auf der Grundlage dieser Kompositionsfähigkeit kann das verteilte Management individuelle Kontexte kreieren, die zur effizienten Realisierung von Rechen-, Speicher- und Kommunikations-fähigen Komponenten auf den Stellen genutzt werden können.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Im Teilprojekt A8 des SFB342 werden Konzepte und Verfahren zur Konstruktion verteilter Systeme für parallele und kooperative Problemlösungen erarbeitet. Der dabei verfolgte Top-Down Ansatz stützt sich anwendungsseits auf die imperative, objektbasierte Sprache INSEL ab. Die Konzepte der Sprache sind geeignet heteromorph parallele Problemlösungen auf unterschiedlichen Abstraktionsniveau zu formulieren. INSEL unterstützt dies durch die Bereitstellung von aktiven und passiven Einheiten auf deren Grundlage sich abstrakte Komponentenmengen mit konzeptionell festgelegten Abhängigkeiten spezifizieren lassen. Die Abhängigkeiten sind über entsprechende Strukturrelationen festgelegt. Das abstrakt verteilte System wird auf der Grundlage der zur Verfügung stehenden anwendungsspezifischen Information mit Hilfe eines geeigneten Transformationsinstrumentariums (TI) sukzessive auf der vernetzten Hardwarekonfiguration konkretisiert. Dabei wird jeder abstrakten Komponente im Zuge der Transformation ein abstrakter Manager beigestellt, der für die Ressourcenverwaltung dieser Einheiten verantwortlich ist. Ausgehend von diesem Prinzip entsteht eine reflexive Managerarchitektur, die an Struktur und Ressourcenbedarf der Anwendung angepaßt ist und sich mit den Anwendungsanforderungen evolutionär weiterentwickelt.

Für die Realisierung der Manager, als auch der Komponenten wird auf ein Spektrum von Realisierungsalternativen auf den unterschiedlichen Transformationsebenen zurückgegriffen. Die Manager des sich entwickelnden verteilt-parallel, kooperativen (V-PK) Systems sollen mit Hilfe des Transformationsinstrumentariums in die Lage versetzt werden, beliebig fein- und grobgranulare Komponenten auf verschiedenen Konkretisierungsebenen zu verwalten. Der Einsatz des TI soll dabei weitestgehend reversibel erfolgen. Dies bedeutet, daß bereits getroffene Entscheidungen zur Realisierung von Komponentenmengen jederzeit revidiert werden können und durch neue, an die Situation besser angepaßte Realisierungsschritte ersetzt werden.

An dieser Stelle entsteht die Problematik, ein stellenbezogenes Basisinstrumentarium bereitzustellen, das auf der einen Seite eine derartige reversible und beliebig feingranulare Verwaltung der hardwarenahen Ressourcen zuläßt und auf der anderen Seite Konzepte der Erweiterbarkeit und Anpaßbarkeit verankert und damit eine verteilten Managerarchitektur wirkungsvoll unterstützt. Es wird ein Basisinstrumentarium benötigt, das eine reversible Konstruktion von grob- und feingranularen Abstraktionen und Basisdiensten ermöglicht, um die hardwarenahe Ressourcenverwaltung den wechselnden Anforderungen einer übergeordneten, verteilten Managementarchitektur dynamisch anpassen zu können. Aus diesen Rahmenbedingungen entstand die Anforderung einen flexiblen, dynamisch-anpaßbaren und gleichzeitig minimalen Stellenkern als Basis für das beschriebene V-PK System bereitzustellen.

Kapitel 2

Charakteristika minimaler Kernarchitekturen

Die Ende der achtziger Jahre entstandenen Mikrokern-Architekturen der ersten Generation, wie Amoeba [T⁺90], MACH 3.0 [Loe91] und Chorus [Cho92] dienten als Basis für die Konstruktion von zentralen und verteilten Client-Server Architekturen und Betriebssystem-Emulationen, vgl. [B⁺92]. Charakteristisch für den Mikrokernansatz war die Minimalisierung und Vereinfachung der Kernfunktionalität. An die Stelle komplexer, aufwendiger Kerndienste, wie sie in ehemals monolithischen Ansätzen zu finden waren, traten bei der Mikrokernen wenige, fundamentale Basisabstraktionen für Prozeß- und Speicherverwaltung und zur Interprozeßkommunikation. Die Verwendung dieser Abstraktionen bei der Betriebssystemkonstruktion ermöglichte eine bessere – gleichzeitig jedoch nur grobgranulare – Anwendungsanpassung von Betriebssystemdiensten und erhöhte damit die Freiheitsgrade in der Entwicklung. Mit der Verlagerung der ursprünglich im Kern angesiedelten Strategien der Ressourcenverwaltung (z.B. Speicherverwaltung, Dateisysteme) in eigenständige User-level Server konnten unterschiedliche Betriebssysteme und Dienste auf *einer* einfachen Basis realisiert werden. Durch den Übergang von monolithischen auf Mikrokern-basierte Architekturen erhoffte man sich inbesondere Vorteile bei der Konstruktion verteilter, modularer Gesamtsysteme.

Der Mikrokern hatte in diesen Systemen die Aufgabe, wenige fundamentale Basisabstraktionen bereitzustellen und die maschinenabhängigen Bereiche der Ressourcenverwaltung zu kapseln. Verteilte Komponenten wurden auf Basis der stellenbezogenen Dienste realisiert und konnten über Nachrichtenkanäle interagieren. Durch eine einfachere Portierung des Kerns konnten auf diese Weise Gesamtsystems schneller an neue Zielplattformen angepaßt werden. Trotz dieser prinzipiellen Vorteile wiesen die Mikrokerne der ersten Generation jedoch einige Einschränkungen auf.

2.1 Probleme des klassischen Mikrokernansatzes

Ausgehend von der Randbedingung, daß viele neue Betriebssysteme die Kompatibilität zu einer vorhandener Softwarebasis berücksichtigen müssen, wurden Mikrokerne vorrangig in Kombination mit User-level UNIX-Servern eingesetzt und bewertet, vgl. [Ous90]. Die im direkten

Vergleich zu monolithischen UNIX-Ansätzen stark erhöhte Nachrichtenkommunikation und die damit verbundenen häufigen Kontextwechsel führten teilweise zu erheblichen Leistungseinbußen durch zusätzliche *cache misses*, vgl. [CB93]. Derartige Leistungseinschränkungen galten lange Zeit als charakteristisch für Mikrokern-basierte Systeme. Untersuchungen von Liedtke, GMD [Lie95] zeigen jedoch, daß die Effizienzeinbußen weitestgehend auf unnötig komplexe Softwarearchitekturen, v.a. aber auf verlustbehaftete Portierungen von Mikrokernen zurückzuführen sind.

Weitere Probleme entstanden durch die in den letzten Jahren manifestierte Diversifizierung der Anwendungen (Multimedia, verteilte Datenbank, Video on demand, etc.) und den damit verbundenen Paradigmenwechsel bei der Betriebssystementwicklung hin zu anwendungsangepaßten und erweiterbaren Architekturen. Auf diese Weise entstanden neue Anforderungen an die Fähigkeiten von Mikrokernen. Gleichzeitig traten bei der Nutzung von Diensten und Abstraktionen des klassischen Mikrokernansatzes Einschränkungen und Engpässe auf, die sich wie folgt klassifizieren lassen:

- ◇ **Variabilität:** Die Kerndienste sind in ihrer Funktion statisch festgelegt. Dadurch ist keine Anwendungsanpassung der Basisdienste an der Kernschnittstelle möglich. Inbesondere ist dadurch keine Einflußmöglichkeit auf die innerhalb des Kerns genutzten Strategien gegeben.

- ◇ **Erweiterbarkeit:** Funktionale Erweiterbarkeit ist meist nur off-line möglich und mit hohem Aufwand verbunden. Dies hat seine Ursache in der mangelnden konzeptionellen Verankerung und in einer geringen internen Software-Modularität.

- ◇ **Granularität:** Die Kernabstraktionen (z.B. ein einfaches Task/Thread Konzept) sind zu grobgranular, um Ressourcenverwaltung mit der notwendigen Feinkörnigkeit durchzuführen zu können.

- ◇ **Orthogonalität:** Die Interferenzen bei Kernabläufen und die Abhängigkeiten zwischen den Abstraktionen sind sehr hoch. Bei Nutzung der Kerndienste läßt dies wenig Spielraum für Realisierungsalternativen. Insbesondere ist auf dieser Grundlage keine orthogonale[1] Ressourcenverwaltung mehr möglich.

- ◇ **Kommunikation:** In nachrichtenorientierten Systemen werden Basisdienste zur Nachrichtenkommunikation (IPC, RPC) sehr häufig eingesetzt, um andere Systemdienste bzw. -objekte zu nutzen. Durch eine fehlende Differenzierung hinsichtlich Güte und Umfang der Kommunikationsdienste, müssen hohe Transferzeiten beim Zugriff auf viele Ressourcen hingenommen werden.

Die genannten Einschränkungen des klassischen Mikrokernansatzes sollen nun anhand von MACH 3.0 – einem populären Vertreter der Mikrokerne erster Generation – verdeutlicht werden.

[1] Der Begriff *orthogonal* bezieht sich hier auf die Unabhängigkeit zwischen Speicher-, Prozeßverwaltung sowie Kommunikation.

2.2 Das Beispiel MACH 3.0

Die von Mach 3.0 (und der Weiterentwicklung Mach 4.0) angebotenen Mechanismen umfassen Abstraktionen für Speicherobjekte und private Adreßräume zur Speicherverwaltung, ein einfaches Aktivitätskonzept (Kernel-Threads) zur Prozeßverwaltung und ein nachrichtenorientiertes Kommunikationsverfahren auf der Grundlage von Ports. Der Mach Mikrokern wurde in erster Linie für die Realisierung grobgranularer Client-Server Architekturen entwickelt.

Betriebssystemdienste wurden dort als separate Serverprozesse mit privaten Adreßräumen realisiert und von Anwendungsprozessen über Interprozeßkommunikation (RPCs) genutzt. Charakteristisch für Systeme mit privaten Adreßräumen ist die enge Verzahnung von Ausführungsfäden (Threads) und Zugriffsidentifikatoren (virtuelle Adressen) mit den hardware–unterstützten Schutz– und Kontrollbereichen[2] des Kerns. Ports wie auch Threads sind dort für die Dauer ihrer Lebenszeit konzeptionell jeweils an einen *einzigen* Schutzbereich (linearer 32-bit Adreßraum) gebunden. Ein Adreßraum unter Mach 3.0 realisiert demzufolge einen abgegrenzten Schutzbereich, dessen Überwindung (z.B. bei einem RPC) mit hohen Kosten verbunden ist (schwergewichtiger Kontextwechsel).

Bei der Weiterentwicklung von Mach 3.0 zu Mach 4.0 wurden konzeptionelle Änderungen vorgenommen, die Kernel-Threads in die Lage versetzen, lokal mehrere Adreßräume zu durchwandern, vgl. [FHL94]. Dieses als Migrating-Thread Konzept bekannte Verfahren wurde anstelle des klassischen RPCs eingesetzt, um effiziente Prozeduraufrufe in fremden Schutzbereichen (z.B. Adreßräumen von Serverprozessen) durchführen zu können. Obwohl ein derartiges Konzept Vorteile für die Realisierung von Client-Server Architekturen hat, bietet es keine signifikante Unterstützung bei der Implementierung eines globalen 64-bit Adreßraums, wie dies auch im Rahmen des in A8 entwickelten V-PK Systems durchgeführt wird.

Ein–Adreßraumansätze zeichnen sich im wesentlichen dadurch aus, daß Speicherobjekte und Aktivitäten als eigenständige und orthogonale Abstraktionen innerhalb des durch die Adreßbezeichner festgelegten globalen Namensraums existieren [OS92] [CLBL92]. Die Kommunikation erfolgt ausschließlich über den gemeinsamen virtuellen Speicher, da jedes Objekt einen eindeutigen Adreßbezeichner im System besitzt. Um differenzierte Zugriffskontrollen und –beschränkungen für Objekte zu realisieren, ist eine Kernunterstützung erforderlich, so daß Objekte und Aktivitäten zu hardware-kontrollierten Schutzbereichen (Protection Domains) zusammengefaßt und effizient und kontrolliert verwaltet werden können.

Das in Mach 3.0 und 4.0 realisierte Konzept der privaten Adreßräume bietet für die effiziente Realisierung von derartigen Schutzdomänen jedoch keine effiziente Unterstützung. Die fehlende Trennung zwischen Adreßbezeichnern (innerhalb eines Adreßraums) und Schutzbereichen wirft hohe Kosten für den Kontextwechsel auf, da ein Domänenwechsel nicht durch den Kern, sondern durch eine User-level Instanz und nur innerhalb *eines* privaten Adreßraums durchgeführt werden kann. Dazu muß ein externer Speichermanager den Speicherkontext durch sequentielles Ein- und Ausblenden der entsprechenden Speichersegmente umladen, was nur durch teure und mehrfache Kerneinsprünge zu realisieren ist.

Eine weitere Einschränkung die sowohl für Mach als auch für klassische UNIX-Kerne zutreffend ist, betrifft das Scheduling von Aktivitäten. Für eine feingranulare Ressourcenverwaltung ist

[2] Schutzbereiche oder *protection domains* sind ausgezeichnete Mengen von Speicherseiten, die hardwareseitig gegen unautorisierte Zugriffe geschützt werden.

ein anwendungsangepaßtes Scheduling erforderlich. Bei der Nutzung der vom Kern verwalteten Aktivitätsträger (Kernel-Threads) hat ein externer Manager keine Einflußmöglichkeit auf die Schedulingstrategien, da diese fest im Kern verankert sind. Die alternative Verwendung von User-level Thread-Paketen ermöglicht hier zwar die Realisierung von angepaßten Strategien, wirft jedoch das aus der Literatur bekannte *two-level Scheduling* Problem [ABLL92] auf. Ein blockierender Aufruf in den Kern führt demnach zu einer Blockade aller User-level Threads. Dieses Problem muß bei Nutzung von externen Thread-Paketen individuell für jede Kernarchitektur gelöst werden. Für die verfügbaren Versionen von Mach waren diesbezüglich noch keine Standardlösungen vorhanden.

Weitere konzeptionelle Defizite betreffen den Bereich der dynamischen Erweiterbarkeit und Anpaßbarkeit der Kerndienste. Der Mach 3.0 Kern bietet diesbezüglich keinerlei Konzepte an. In Mach 4.0 wurde dagegen ein Konzept für *In-Kernel-Server* verankert, das die Ausführung von ausgewählten (Benutzer-)Prozesses im Schutzbereich des Kerns erlaubt. Damit wurde hier eine grobgranulare dynamische Erweiterungsmöglichkeit realisiert.

2.3 Neuere Kernansätze

Aufgrund dieser charakteristischen Probleme der Mikrokerne erster Generation wurden in den letzten Jahren neue Entwicklungsrichtungen eingeschlagen. Diese lassen sich grob in drei Bereiche unterteilen:

Mikrokerne der zweiten Generation

Viele leistungsbezogene Einschränkungen der ursprünglichen Mikrokernansatzes wurden beseitigt. Mikrokerne der nächsten Generation wie L4 [Lie95] und Fluke [F^+96] oder der Spring Nucleus [HK93] sind kompakter, modularer und bieten deutlich verbesserte Leistungswerte, v.a. bei der Interprozeßkommunikation. Allerdings wurde hierbei keine Konzepte zur Flexibilisierung und Erweiterung von Basisdiensten integriert. Dies bedeutet, daß die implementierten Dienste und Abstraktionen lediglich off-line – und mit beträchtlichem Aufwand – erweiter- und anpaßbar sind.

Minimalisierte Architekturen (Nanokerne)

Durch eine weitere Vereinfachung der Kernfunktionalität und die Reduktion von Kernabstraktionen zu unkorrelierten, hardwarenahen Primitiven entstanden sog. Nano- und Picokerne. Die angebotenen Basisprimitive (z.B. Prozessor-Kontexte, MMU-Mappings, Atomic-Locks etc.) werden von höheren Systemeinheiten genutzt, um anwendungsangepaßte Basisdienste zu konstruieren. Diese Aufgabe wird i.d.R. von sog. *Application-Kernels* erfüllt, die für eine spezielle Klasse von Anwendungen jeweils neu entwickelt werden müssen. Die Nutzung feingranulare Basiskomponenten erlaubt hier eine nahezu uneingeschränkte Gestaltung von Systemdiensten, es zeigt sich jedoch, daß die mit der genutzten Feinkörnigkeit verbundene erhöhte Zahl an Kerneinsprüngen tatsächliche Leistungseinbußen mit sich bringen. Beispiel für Nanokernansätze sind u.a. der V++ Cache-Kernel [CD94], μ-Choices [TRC95], und der Exokernel [EKO95].

Erweiterbare Kernarchitekturen

Eine weitere Entwicklungsrichtung verfolgte das Ziel, Erweiterbarkeit als fundamentale Eigenschaft in die Kerne zu integrieren. Die Anwendungsanpassung wird bei diesen Architekturen durch eine kontrollierte Integration von zeitkritischen Codemodulen (z.B. Strategien, Kommunikationsprotokollen) in den Schutzbereich des Kerns vollzogen. Dabei sind i.d.R. zusätzliche Maßnahmen notwendig, die eine Zertifikation der entsprechenden Erweiterungsmodule vornehmen. Die damit verbundenen Techniken schließen Compilerüberprüfungen durch separate Kerncompiler und Sandboxing-Verfahren [WLAG93] mit ein, wie z.B. bei Spin [BSP+95] oder Vino [SESS94]. Andere Architekturen, wie Paramecium [vDHT95], bieten eine Zertifikationsinstanz zur kontrollierten runtime Erweiterung des Kerns. Die daraus resultierenden Architekturen können jedoch konzeptionell nicht mehr als Mikrokerne bezeichnet werden, da die verwendeten Techniken zur Erweiterung komplex sind nur mit Hilfe umfangreicher Software-Architekturen zu realisieren sind.

2.4 Einordnung des Dycos-Stellenkerns

Das im Teilprojekt A8 des SFB 342 entstehende Gesamtsystem stellt jedoch verschiedenartige Anforderungen an die Basis. Zum einen wird ein *funktional minimales* Basisinstrumentarium benötigt, das sowohl fein- als auch grobgranulare Abstraktionen zur Konstruktion von neuen und angepaßten stellenlokalen Basismechanismen bereitstellt. Zum anderen soll durch konzeptionelle Verankerung funktionaler Erweiterbarkeit und Konfigurationsfähigkeit die Grundlage dafür geschaffen werden, anwendungsangepaßte Basisdienste zur Laufzeit zu konstruieren, die sowohl für Aufgaben der zentralen als auch der verteilten Ressourcenverwaltung herangezogen werden können. Der dafür zugeschnittene Stellenkern muß sowohl eine hohe Flexibilität und Dynamik durch feinkörnige Abstraktionen aufweisen, gleichzeitig aber auch Konzepte zur Erweiterbarkeit beinhalten. Ein reiner Nanokern-Ansatz ist für diesen Zweck nicht ausreichend, da dieses Konzept keine Erweiterungen und Anpassungen der Dienste im Schutzbereich des Kerns (Kernel-Modus) zuläßt. Alle auf Erweiterbarkeit fokussierten Ansätze scheiden andererseits aufgrund der hohen Software-Komplexität aus. Die in diesen Kernarchitekturen integrierten Techniken sind vor dem Hintergrund der funktionalen Minimalisierung der Basisschicht nicht adäquat.

Das Entwicklungsziel besteht damit in der Integration der Eigenschaften der Nanokerne und der erweiterbaren Kernansätze.

Kapitel 3

Konzepte und Abstraktionen des Dycos-Kerns

In diesem Kapitel soll Grundlegendes über Konzepte und Abstraktionen des Dycos Stellenkerns beschrieben werden. Wesentliche Aufgabe des Kerns ist es, ein Instrumentarium bereitzustellen, das einen Rahmen für die Konstruktion anwendungsangepaßter Basismechanismen definiert. Die Basis dafür wird durch ein Baukasten-ähnliches Architekturprinzip gewährleistet. Der Kern verfügt über eine variable Menge von fein- und grobgranularen Kernbausteinen (Objekte), die dynamisch (d.h. nach Bedarf) erzeugt und wieder aufgelöst werden können. Neben diesen Objekten ist eine Anzahl elementarer Operationen definiert, die in erster Linie als Werkzeugfunktionen (Toolset) dienen, um mit den Kernbausteinen umzugehen und diese zu konfigurieren. Die Kernbausteine selbst haben dabei besondere Eigenschaften, die sie in die Lage versetzen frei zu kommunizieren und dynamische Bindungen[1] einzugehen. Mit Hilfe dieses allgemeinen Konzepts lassen sich zur Laufzeit anwendungsangepaßte Kompositionen von Bausteinen bilden. Diese Kompositionen besitzen beliebige Funktionalität mit variablen Abstraktionsgraden (d.h. wahlweise einfache oder komplexe Systemdienste) und können über einen flexiblen Schnittstellenmechanismus von höherwertigen Systemkomponenten genutzt werden.

In den folgenden Abschnitten werden die Kernbausteine, die elementaren Operationen, die Kommunikationsmechanismen und die Schnittstelleneigenschaften des Kerns weitergehend erläutert.

3.1 Kernbausteine

Kernbausteine sind in erster Linie Datenabstraktionsmodule, die definierte Zugriffe auf (hardwarenahe) physikalische Ressourcen (z.B. Speicherseiten, Registerinhalte) oder auch abstrakte Ressourcen (z.B. virtuelle Nachrichtenkanäle) der zugrundeliegenden Hardwarekonfiguration ermöglichen. Dabei werden die für die Verwaltung notwendigen Datenstrukturen in den Kernbausteinen gekapselt und sind lediglich über definierte oder definierbare Zugriffsoperationen zugänglich (Objektprinzip). Der Nutzer wird dadurch in die Lage versetzt, Ressourcenmanagement auf einem wahlweise sehr niedrigen (hardwarenahen) und gleichzeitig feinkörnigen Niveau

[1]Damit ist ein Dycos-spezifischer Vorgang bezeichnet, der es erlaubt, Gruppierungen von Objekten zu bilden.

oder auf einem beliebig hohen Abstraktionsgrad durchzuführen. Die Einführung von Kernbausteinen ist gleichzeitig eine Voraussetzung für die Gewährleistung dreier wesentlicher Merkmale und Ziele:

(1) Modularisierung, Strukturierung der Software-Architektur

(2) Einführung unterschiedlich granularer Abstraktionen

(3) Realisierung anwendungsangepaßter Kompositionen

Die Kernbausteine (Objekte) in Dycos haben allgemein folgende Merkmale:

- Kernbausteine sind Instantiierungen von Objektklassen
- Jeder Kernbaustein exportiert eine vordefinierte (d.h. klassenspezifische) Schnittstelle
- Für einige Kernbausteine lassen sich zusätzlich individuelle Schnittstellen festlegen (siehe Kap. 4.2 und 4.3)
- Jeder Baustein hat klassenspezifische und individuelle objektspezifische Eigenschaften (u.a. einen eindeutigen Bezeichner)
- Alle Kernbausteine sind bindungs- und kommunikationsfähig

Feingranulare Kernbausteine (FOs)

Die feingranularen Kernbausteine entsprechen den kleinsten für die Komposition zu Verfügung stehenden Objekten. Sie lassen sich zur Laufzeit als Inkarnationen der entsprechenden FO Klassen erzeugen und bei Bedarf auch wieder vernichten. Jeder Kernbaustein verfügt über ein duale Schnittstelle. Der eine Teil der Schnittstelle wird über das Toolset genutzt und steht somit nur indirekt zur Verfügung. Über den anderen Teil bieten die Bausteine eine Reihe von Funktionen, die von der Nutzungsebene[2] direkt (mittels Event-basierter Kommunikation, siehe Kap. 3.3.1) zugreifbar sind. Weiterhin sind FOs als bindungsfähige Objekte ausgelegt, die durch dynamisches Binden (elementare Operation `Attach()`) mit anderen grob- und feingranularen Objekten gekoppelt werden können. Der Vorgang des dynamischen Bindens ist kernspezifisch definiert und verankert. Im Gegensatz zum klassischen Bindevorgang durch einen inkrementellen Binder, erfolgt in Dycos keine direkte Auflösung von Referenzen[3]. Die Bindungsparter tauschen dagegen beim dynamischen Binden ihre exportierten Schnittstellen aus und ermöglichen damit eine einfachere Objektkommunikation über eine dezentrale Referenzadresse mittels ACE-Aufrufen (siehe Kap. 3.3.2). Im ungebundenen Zustand können Objekte lediglich über Events angesprochen werden. Der zugrundeliegende Mechanismus basiert auf der Dycos-internen Event-Kommunikation.

FOs verfügen über objektspezifische Attribute, die durch Nutzung der Schnittstelle geändert werden können. Zu diesen Attributen zählt im wesentlichen auch der interne Status eines Objekts, der durch Aufruf einer elementaren Operation aus dem Toolset geändert werden kann. Die damit verbundenen Zustandsübergänge sind in Kap. 3.2.1 näher beschrieben.

[2] Mit Nutzungsebene sind die höherwertigen Betriebssysteminstanzen gemeint und nicht das Programm des Anwenders.

[3] Dies wäre irreversibel und deshalb ungeeignet.

FO Klassen	**Beschreibung**	**Aufgabe**
`ACB`	Activation Control Block	Register- und Prozessorstatusverwaltung
`LACB`	Light Activation Control Block	Register- und Prozessorstatusverwaltung
`PQueue`	FIFO-Warteschlange	Warteraum für Aktivitäten
`VMPage`	Virtuelles Speicherobjekt	Seitenbasierte Speicherverwaltung
`VMGroup`	Gruppenobjekt für VMPages	Seitenbasierte Speicherverwaltung
`TLBCache`	Softwarecache	Mapping der TLB Einträge
`Stack`	Stackobjekt	Prozeß- und Speichermanagement
`CStub`	Nachrichtenwarteraum	Synchrone Nachrichtenkommunikation
`MStub`	Mailbox	Asynchrone Nachrichtenkommunikation
`RPCStub`	Endpunkt für entfernte Prozeduraufrufe	Operationen-orientiertes Rendezvous
`Event`	System- oder Benutzerdefiniertes Signal	Event-basierte Kommunikation
`EventHandler`	Signalbehandlungsobjekt	Ausnahmebehandlung, Rahmen für Kernerweiterung
`TObject`	Timer Objekt	Zeitbasis für Ablaufsteuerung
`Sema`	Counting Semaphore	Synchronisationsmittel
`Mutex`	Binäre Semaphore	Synchronisationsmittel
`ECounter`	Ereigniszähler	Synchronisationsmittel

Tabelle 3.1: Übersicht über die FO Klassen

Tabelle 3.1 gibt einen Überblick über die feingranularen Kernbausteine und ihre Aufgaben innerhalb des Kerns.

Grobgranulare Kernbausteine (COs, Kontexte)

Die grobgranularen Kernbausteine (COs oder auch Kontexte genannt) bilden die Rahmenstrukturen (Frameworks) für FOs. Im allgemeinen hat ein Kontext die gleichen Merkmale und Eigenschaften eines FOs. Lediglich seine Funktionalität ist im initialen Zustand noch nicht (vor-)definiert. Diese erlangt er erst durch Bindung mit sinnvoll gewählten FOs. Ein Kontext hat deshalb die primäre Aufgabe eine beliebige Anzahl von FOs zu kapseln und damit größere funktionale Verwaltungseinheiten zu bilden. Mit diesen Kompositionen auf Basis der FOs/COs geht auch eine Anhebung der Abstraktionsebene[4] des Kerns einher, da ein angereicherter Kontext (tagged Kontext) auch durch eine Anwendung genutzt werden kann. Die Anreicherung eines Kontextes mit FOs basiert auf dem Konzept des dynamischen Bindens (elementare Operation `Attach()`). Angereicherte Kontexte können wiederum gegenseitig gebunden werden. Dies ist zugleich die Ausgangsbasis für die Komposition von Anwendungsumgebungen durch dynamisch gebundene Kontexte unterschiedlicher Einordnung (siehe Kap. 4.1.2).

Für höherwertige Betriebssysteminstanzen der Managerarchitektur lassen sich auf diese Weise Abstraktionen für Rechen-, Speicher- und Kommunikationsfähigkeit realisieren. Diese Abstrak-

[4] Eine derartige Komposition kann höherwertige Systemdienste umfassen als dies einzelne FOs bereitstellen.

tionen lassen sich dabei weitestgehend an eine konkrete Anwendungsumgebung anpassen und erlauben es dadurch geeignete, angepaßte Basismechanismen und -abstraktionen zu schaffen. Durch die konkrete Einschränkung auf die individuell funktional notwendigen Abläufe und Mechanismen können auf dieser Grundlage wesentlich effizientere Systemdienste realisiert werden, als dies mit herkömmlichen Kernarchitekturen möglich ist.

Beispiele für *tagged Kontexte* sind Tabelle 3.2 zu entnehmen.

Tagged Kontexte	**Dynamisch gebundene FOs**
ActivationContext	`ACB (LACB), Stack, PQueue`
MemoryDomainContext	`VMGroup, VMPage, TLBCache`
CommunicationContext	`CStub, MStub, RPCStub`

Tabelle 3.2: Beispiele für tagged Kontexte

Ein Kontext ist zugleich die Einheit für einen Kontextwechsel. Der Kontextwechsel zwischen zwei Kontexten A und B wird durch die Folge zweier Objektaufrufe (`A.unload(), B.load()`) vollzogen. Hinter jedem dieser exportierten Objektmethoden eines Kontextes verbirgt sich dabei eine Operationen Sequenz (`OpSeq-`) Liste. Diese kann vom Benutzer frei definiert werden und bestimmt, welche konkreten Schritte und Operationen während des Kontextwechsels durchgeführt werden sollen. Mit Hilfe der `OpSeqs` lassen sich also Kernabläufe frei definieren und somit optimal an die Anforderungen der Anwendung anpassen.

`OpSeqs` sind ebenfalls in den FOs der Klasse `EventHandler` enthalten und ermöglichen dadurch, daß beliebige Events, die für ein Objekt angemeldet wurden, mit diesen `OpSeq`-Listen assoziieren werden können. Mit Hilfe dieses Konzepts lassen sich Event-gesteuerte Ausführungsfolgen für Kontexte und EventHandler festlegen (siehe dazu auch Kap. 4.3).

Das FO/CO Prinzip eröffnet die Möglichkeit mit Hilfe spezieller, angereicherter (tagged) Kontexte beliebig abgestufte Anwendungsumgebungen zu schaffen. Die Kosten für einen Kontextwechsel sind dabei durch die Anzahl und Eigenschaften der gebundenen Kernbausteine in weiten Bereichen frei definierbar. Beispielsweise ist ein Kontextwechsel, der lediglich einen Maschinenregistersatz ändert billiger, als ein entsprechender Vorgang, der gleichzeitig auf eine neuen Schutzdomäne[5] wechselt. In diesem Zusammenhang kann man von der Erzeugung *differenziert-gewichteter Kontexte* sprechen.

3.2 Elementare Operationen

In diesem Abschnitt wird das Dycos-Toolset beschrieben. Das Toolset besteht aus einer Menge elementarer Kernoperationen, die benötigt werden, um mit den Kernbausteinen umzugehen.

[5] Die bezeichnet einen vom Prozessor gesicherten (u.U. nicht zusammenhängenen) virtuellen Speicherbereich.

Alloc/Dealloc

Generische Operation zur Erzeugung bzw. Vernichtung von Kernbausteinen. Konkret werden durch den Aufruf von `Alloc()` neue Objektinkarnationen einer spezifischen Objektklasse zur Laufzeit angelegt. Die Funktion legt dabei lediglich den Speicherplatz[6] für das neuen Objekt fest und initialisiert die mit dem Objekt assoziierten Datenstrukturen. Das Objekt geht durch den Aufruf vom (virtuellen) Zustand EXPIRED in den Zustand ALLOCATED über. Bei der Deallokation mittels `Dealloc()` werden die Datenstrukturen aufgelöst und der reservierte Speicherplatz wieder freigegeben.

Map/Unmap

Mit Hilfe dieser Operationen lassen sich speicherbezogene Objekte (wie `Stack, VMPage, VMGroup, ...`) in den virtuellen Adreßraum des Prozessors ein- bzw. ausblenden. Damit kann auf die ansonsten innerhalb des Objektes gekapselten Ressourcen (virtuelle Speichersegmente) direkt zugegriffen werden. Bei Aufruf der `Map()` Operation muß ein bereits definierter ActivationContext angegeben werden, der als externer oder interner Speicherverwaltungsmanager für die transparente Bereitstellung des Seiteninhalts verantwortlich ist. Sowohl die `Map()` also auch die `Unmap()` Operation führen Änderungen an der stellenlokalen Seiten/Segmenttabelle durch.

Register/Unregister

Die elementare Operation `Register()` meldet ein zuvor allokiertes Kernobjekt beim kerninternen Event-Dispatcher an. Das Registrieren überführt das Objekt in den internen Zustand DISABLED und ist gleichzeitig Voraussetzung für die weitere Nutzung des Kernobjektes (insbesondere für die Bindung an Kontexte). Dafür muß der Benutzer eine Menge von Events angeben, die nach Signalisierung mit dem Objekt assoziiert werden sollen. Diese Information wird in einer objektbeschreibenden Zuordnungstabelle in der klassenspezifischen Objektstrukturliste (OSL) abgelegt. Der Dispatcher vergibt für jedes registrierte Kernobjekt einen (zeitlich und räumlich) eindeutigen 128-bit Bezeichner (*unique identifier*). Nach der Registrierung kann auf das Objekt nun unter Angabe dieses eindeutigen Bezeichners zugegriffen werden. Die `Unregister()` Operation löscht ein zuvor registriertes Kerndatenobjekt wieder aus der OSL. Danach ist das Objekt nicht mehr über die Kommunikationsmechanismen zugreifbar, liegt jedoch noch als Speicherfragment im Heap vor. Ein unregistriertes Objekt kann entweder mittels `Dealloc` gelöscht werden (falls nicht mehr benötigt) oder auf andere Stellen migriert und dort mit dem gleichen Zustand wieder registriert werden.

Enable/Disable

Die `Enable/Disable` Operation ist lediglich für Kernbausteine vorgesehen, die validiert bzw. invalidiert werden können (d.h. nur für Kontexte). Durch `Enable()` wird ein zuvor registriertes Kontextobjekt vom Zustand DISABLED in den Zustand READY überführt. Damit ist das

[6]Das Objekt kann dabei wahlweise im User-level oder im Kernel-level Bereich angelegt werden.

Kontextobjekt für eine spätere Validierung und Invalidierung durch die elementare Operation `SwitchContext()` vorbereitet. Mit `Disable()` läßt sich ein bereits aktiv verwendetes Kontextobjekt wieder bearbeiten. Im Zustand DISABLED können Kontexte durch dynamisches Binden konfiguriert und erweitert werden. Während des Bindungsprozesses ist es jedoch aus Konsistenzgründen nicht erlaubt, den Kontext zu validieren.

Attach/Detach

Mit `Attach()` lassen sich registrierte Kernobjekte (FOs/COs) dynamisch aneinander binden. Dies eröffnet die Möglichkeit mehrere Kontexte zu koppeln oder einzelne Kontexte mit FOs anzureichern. Gebundene Objekte werden als Einheit betrachtet (z.B. beim Kontextwechsel) und können intern auf ein anderes Kommunikationsverfahren (ACE-Aufrufe) zurückgreifen. Während Objektbindungen in den meisten Fällen einfach sind, lassen sich für einige Objekte auch Mehrfachbindungen realisieren. Weiterhin erlaubt `Attach()` die Definition von Kernablaufsequenzen `OpSeqs`. Die `Detach()` Operation löst eine bestehende Bindung zwischen zwei Kernobjekten wieder auf. Beide Operationen sind nur für Objekte im Zustand DISABLED zulässig.

SwitchContext

Die `SwitchContext()` Operation wird genutzt, um in einem ersten Schritt den aktuell gültigen Kontext zu invalidieren, und im nächsten Schritt einen neuen Kontext zu validieren. Kernintern wird dazu beim aktuellen Kontext die Methode `unload` und in Folge beim neuen Kontext die Methode `load` aufgerufen. Die dahinter verborgenen Kernabläufe sind durch (benutzerdefinierbare) `OpSeqs` festgelegt. Damit sind auch die Kosten für einen Kontextwechsel variabel, da diese von Anzahl und Bearbeitungsaufwand der Einträge innerhalb der `OpSeqs` abhängen. `SwitchContext()` ist nur für Kontexte definiert und zieht einen Zustandswechsel von READY auf VALID bzw. umgekehrt nach sich.

QueryInfo

`QueryInfo()` erlaubt den Lesezugriff auf Zustandsdaten, die innerhalb eines Kernobjektes gekapselt sind. Dies schließt die Bereitstellung von Information über den aktuellen Bindungszustand eines Objekts, die Belegung seiner `OpSeq` (falls vorhanden) oder die exportierte Schnittstelle ein. Die Operation ist nur für Objekte im Zustand DISABLED zulässig.

3.2.1 Zustandsübergangsdiagramm für Kernbausteine

Sowohl für FOs als auch für COs ist ein Zustandsübergangsdiagramm definiert, das alle möglichen Objektzustände charakterisiert und die Übergänge zwischen den Zuständen abängig von Kernabläufen und elementaren Operationen festlegt. In der nachfolgenden Abbildung 3.1 ist der Zustandsgraph für alle Klassen von Kernbausteinen dargestellt. Feingranulare Bausteine können jeweils nur die grau hinterlegten Zustände annehmen.

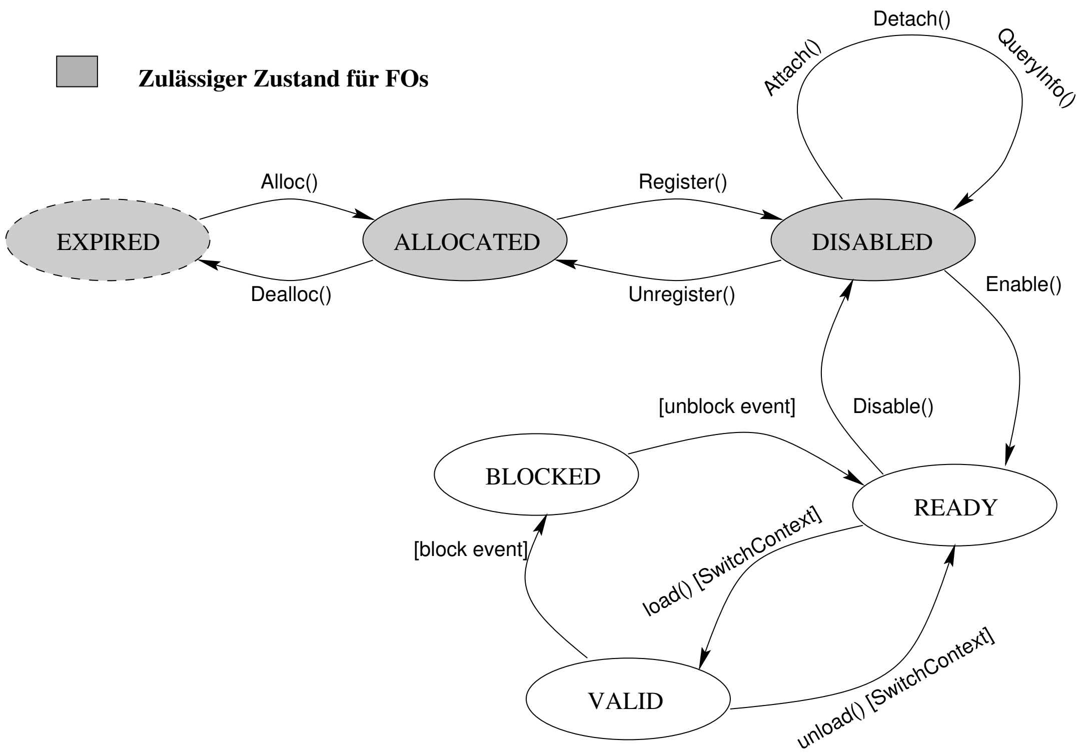

Abbildung 3.1: Zustandsübergangsdiagramm für FOs und COs

3.3 Kommunikationsmechanismen

Die elementaren Kommunikationsmechanismen des Dycos-Kerns werden zur Interobjektkommunikation und zur Interaktion zwischen Objekten und Anwendungen genutzt. Gleichzeitig dienen sie als Ausgangsbasis für die Konstruktion höherwertiger Kommunikationsmechanismen und -protokolle. Die elementaren Verfahren zur Interobjektkommunikation werden genau dann benötigt, wenn ein Kernbaustein aus einer (gültigen) Kontextumgebung einen Zugriff auf andere Objektmethoden im gleichen oder einem anderen Kontext durchführen möchte. Um dies zu erreichen, stehen generell zwei Kategorien von Kommunikationsverfahren zur Verfügung – die Event-basierte Kommunikation und die Kommunikation über ACE-Aufrufe.

3.3.1 Event-basierte Kommunikation

Die kernglobal nutzbare Event-basierte Kommunikation beruht auf einem unidirektionalen Signalisierungsverfahren und kann damit als nachrichtenorientiertes Verfahren charakterisiert werden. Die Einführung von Events als zentrales Verfahren zur kerninternen Interobjektkommunikation trägt der Tatsache Rechnung, daß die direkte Nutzung von Adreßbezeichnern in einem modular erweiterbaren System aus Zuverlässigkeitsgründen nicht tragbar ist. Direkte Methodenzugriffe über evtl. veraltete (d.h. ungültige) Objektreferenzen können im privilegierten Prozessormodus zu unauflösbaren, d.h. die Stabilität gefährdenden Zuständen führen. Ein Event-orientiertes Verfahren fungiert dagegen als Kontrollinstanz und ermöglicht weiterhin die homogene Integration der Hardware-generierten Ereignisse.

Als Rahmenbedingungen für die Anwendung der Event-basierten Kommunikation gelten folgende Grundregeln:

- ◇ Events sind selbst Kernbausteine und müssen vor der Nutzung beim Event-Dispatcher (im Kern) registriert worden sein (`Register()`).
- ◇ Events können mit Objektmethoden dynamisch assoziiert werden. Diese Assoziation erfolgt entweder initial beim Registrieren eines Objekts oder nachträglich durch dynamisches Binden zwischen einem Event und einer Objektmethode.
- ◇ Da Events nicht eindeutig *einem einzigen* Objekt zugeordnet sein können, müssen sie zusätzlich noch zieladressiert werden. Dazu wird der beim Registrieren vergebene *Unique Identifier* verwendet.
- ◇ Events unterscheiden sich konzeptionell von anderen Kernbausteinen in bezug auf die Eventnutzung und die dynamische Bindungsfähigkeit. Eventauslösungen werden real nicht über objektspezifischen Methodenaufrufe, sondern über Software-Traps realisiert. Dynamische Bindungen zwischen Events und anderen Kernbausteinen haben lediglich Einfluß auf die Datenbasis des Event-Dispatchers.

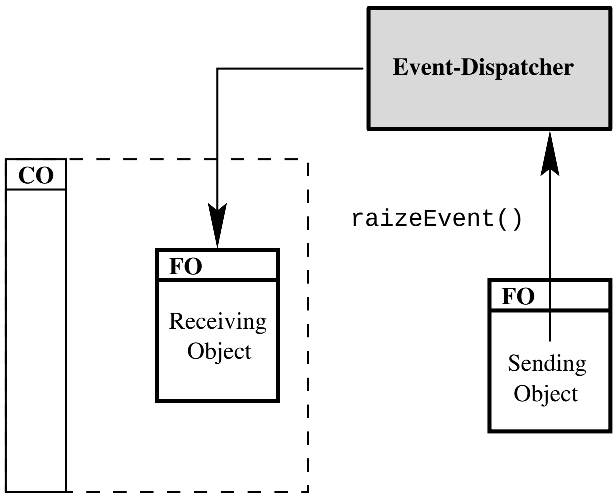

Abbildung 3.2: Event-basierte Kommunikation

Der Ablauf bei der Event-basierten Kommunikation ist aus Abbildung 3.2 ersichtlich. Der Initiator[7] löst explizit (aktiv in der Berechnungsphase) oder implizit (durch eine Operation, die eine vom Prozessor initiierte Ausnahmebehandlung zur Folge hat) ein bereits vordefiniertes (d.h. beim Kern registriertes) Event aus und referenziert dabei einen vorbelegten Event-Qualifier Vektor (`EQVec`). Der `EQVec` beinhaltet Information über die eindeutigen Bezeichner von Quell- und Zielobjekt, den genutzten Dispatcher Eingang (Gate) sowie die aktuellen Parameter, die

[7]Dies kann eine Anwendung, die Hardware oder ein Kernbaustein sein.

der Zielmethode übergeben werden sollen. Das Tupel {`Event, EQVec`} wird einem zentralen Nachrichtenvermittler, dem Event-Dispatcher zugeordnet. Dort wird anhand einer Tabelle eine Assoziation zwischen Event der Methode im Zielobjekt vorgenommen und die entsprechende Funktion zusammen mit dem `EQVec` aufgerufen. Dies erfolgt sequentiell relativ zum aktuellen Ausführungsfluß. Das Zielobjekt kann in einer beliebigen lokalen Kontextumgebung liegen. Ist die Kontextumgebung dagegen auf einem entfernten Knoten, so muß das Event nach dem Dispatcher noch eine weitere Vermittlungsinstanz (z.B. ein `EventHandler`) durchlaufen, die jedoch spezifisch konstruiert und auf Basis der Erweiterbarkeits- und Anpassbarkeitsverfahren in den Kern integriert werden muß (siehe Kap. 4.3). Die Kontrolle der Zugriffsberechtigung wird zunächst nur im Dispatcher durchgeführt, kann aber bei Bedarf im Zielobjekt selbst oder in evtl. zusätzlichen Vermittlungsinstanzen erweitert werden.

Event-Dispatcher

Der Event-Dispatcher fungiert als stellenlokaler Nachrichtenvermittler und als Zertifikationsinstanz. Für jede Klasse von Kernbausteinen wird eine Objektstrukturliste (OSL) verwaltet, die beim Registrieren bzw. Deregistrieren eines Objektes aktualisiert wird. In der klassenspezifischen OSL befinden sich für jedes registrierte Objekt folgende Einträge:

- Der *Unique Identifier* des Objektes,
- ein gültiger Objektzeiger (Adreßbezeichner),
- die exportierte Schnittstelle des Objektes (Array von Methodenzeigern),
- eine Event Referenz Liste mit einer Abbildung $[EventX \rightarrow MethodeY]$ für jedes mit dem Objekt assoziierte Event.

Beim Signalisieren eines Events prüft der Event-Dispatcher nun anhand der klassenspezifischen Objektstrukturliste (OSL), ob das adressierte Zielobjekt einen Methodenaufruf über das aktuelle Event zuläßt. Ist dies der Fall, so ruft der Dispatcher die Zielmethode direkt auf. Dies erfolgt sequentiell eingeordnet zum aktuellen Ausführungsfluß. Liegt jedoch in der OSL keine gültige Eintragung vor, so wird das ausgelöste Event verworfen.

Da die Event Kommunikation als homogenes Verfahren sowohl für die kerninterne Interaktion als auch für Kerneinsprünge von der Anwendungsebene aus genutzt wird, verfügt der Dispatcher über mehrere Eingänge (Gates). Diese gezielte Auswahl eines Gates ermöglicht es, die für die unterschiedlichen Rahmenbedingungen der Event-basierten Kommunikation notwendigen qualitativen Differenzierungen vorzunehmen. Näheres über die vorhandenen Gates und ihre Nutzung in Kap. 3.4.

3.3.2 ACE-basierte Kommunikation (ACE-Aufrufe)

Eine weitere Möglichkeit der Objektkommunikation wird durch sogenannte *Access Control Entry* (ACE-) Aufrufe (auch ACE-Calls) bereitgestellt. Dieses Verfahren ist grundsätzlich nur innerhalb von Kontexten oder gebundenen Kontexten zulässig und realisieren eine spezifische Art des indirekten Methodenaufrufs. ACE-Calls dienen zur Beschleunigung und Vereinfachung der Objektnutzung innerhalb eines wohldefinierten Kontextrahmens. Die potentielle Stabilitätsgefahr durch einen Methodenaufruf auf evtl. nicht-existente Objekte kann hierbei jedoch wirkungsvoll abgefangen oder zumindest in der Auswirkung auf den aktuellen Kontext beschränkt werden. Das Prinzip der ACE-Aufrufe basiert auf folgenden Rahmenbedingungen:

- ◇ Während des dynamischen Bindens zweier Kernbausteine tauschen diese ihre nach außen exportierte Schnittstelle aus.
- ◇ Die exportierte Schnittstelle wird auch als Access Control Entry (ACE-)Vektor bezeichnet. Der ACE-Vektor beinhaltet die Referenzen auf die nutzbaren Objektmethoden.
- ◇ Beim dynamischen Binden von FOs an einen Kontext führt der Kontext eine ACE Referenzliste aller gebundenen FOs und COs.

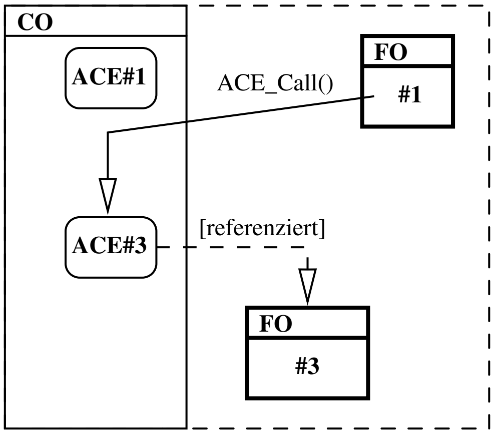

Abbildung 3.3: ACE-Aufrufe innerhalb eines Kontextes

Der ACE-Aufruf erlaubt nun den einfachen[8] Zugriff auf alle zur Verfügung stehenden Objekte bzw. Objektmethoden innerhalb einer gültigen Kontextumgebung. Damit kann nun neben dem allgemein nutzbaren rein nachrichtenorientierten Zugriffsverfahren noch ein prozedurales Verfahren zur Interobjektkommunikation herangezogen werden. Abbildung 3.3 zeigt dies exemplarisch.

3.3.3 Höherwertige Kommunikationsverfahren

Die Konstruktion höherwertiger Kommunikationsverfahren und -mechanismen läßt sich auf der Grundlage der vorhandenen elementaren Verfahren realisieren. In Kombination mit den entsprechenden Kernbausteinen `CStub, MStub, EventHandler` können sowohl synchrone als auch asynchrone Kommunikationskanäle etabliert werden. Abbildung 3.4 zeigt eine derartige Konstruktion mit Mailbox-Funktion (Port-ähnliche Kommunikation). Dabei werden vom Sender mittels Event-Signalisierung Daten in einen Empfangspuffer (`Mailbox`) eingetragen. Die zwischengespeicherten Daten werden im weiteren asynchron von einem (benutzerdefinierten) Protokoll EventHandler ausgelesen und interpretiert.

[8] Der Zugriff erfolgt unter Umgehung des Event-Dispatchers.

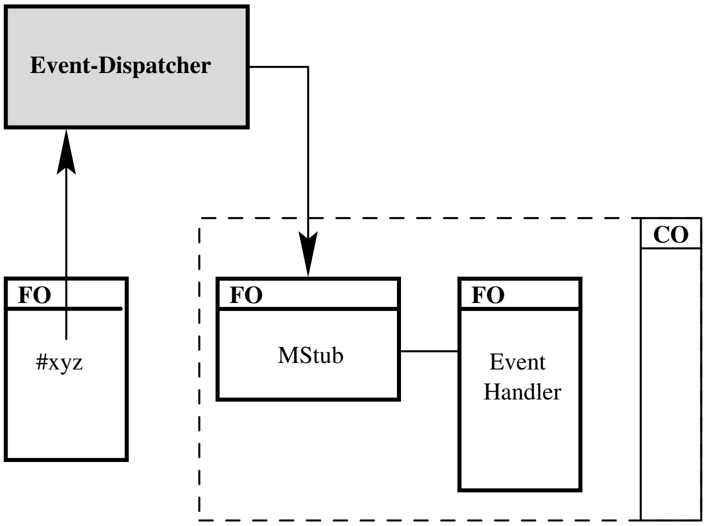

Abbildung 3.4: Mailbox-Kommunikation

Ein wesentlicher Vorteil der eben exemplarisch erläuterten freien Konstruktion höherwertiger Verfahren ist der, daß die Kommunikationsobjekte nach Bedarf dynamisch in den Anwendungskontext integriert werden können und die Interpretation der verschickten Nachrichten frei über die Protokoll EventHandler festgelegt werden kann.

3.4 Kernschnittstelle

Der anwendungsseitige Zugriff auf den Kern und seine Kernbausteine erfolgt über eine offene, erweiterbare Schnittstelle. Diese Schnittstelle wird konzeptionell über den Event-Dispatcher bereitgestellt. Für Systemaufrufe stehen dort generell drei verschiedene Eingänge (Gates) mit unterschiedlichen Eigenschaften zur Verfügung.

GateA
Einfacher User-level Kerneintrittspunkt. Erlaubt eine schnelle Ausführung von Kernoperationen im Anwendungskontext. Die Nutzung der Event-basierten Kommunikation wird nicht unterstützt. Gate A wird beim Aufruf der elementaren Operationen genutzt.

GateB
Reentranter Event-basierten Eintrittspunkt, der für die kerninterne Event-Signalisierung genutzt wird. Die aufrufende Instanz muß dabei ein `Event` und einen `EQVec` spezifizieren. Die unter Gate B ausgeführten Kernoperationen laufen im Kontext des Aufrufers.

GateC
Reentranter Standard-Kerneintrittspunkt für die Anwendung. Gate C stützt sich - analog zu Gate B - ebenfalls auf dem Event-basierten Kommunikationsverfahren ab, erzeugt jedoch beim Eintritt eine separate Rechenumgebung, die vom Kontext des Aufrufers strikt getrennt ist.

Die Eintritt in den Kern ist also auf unterschiedlichem Wege möglich. GateA unterstützt die konventionelle Parameterübergabe, während der Zugang zu GateB und GateC nur mit den Konventionen der Event-basierten Kommunikation möglich ist. An der Kernschnittstelle stehen damit folgende Operationen und Methoden zur Verfügung:

- Die elementaren Operationen des Kerns, vgl. Kap. 3.2.
- Alle benutzerdefinierten Events und die damit assoziierten objektspezifische Methoden (evtl. mit Einschränkungen, falls ein Objekt seine Methoden nach außen sperrt).

Da Events auch Kernbausteine sind, die zur Laufzeit erzeugt und gebunden werden können, kann die Kernschnittstelle dynamisch erweitert werden. Ein neu erzeugtes Event kann auf dieser Basis mit Kernablaufsequenzen `OpSeqs` assoziiert werden. Auf diese Weise lassen sich alle vorhandenen Objekte und ihre Methoden von der Anwendung aus nutzen. Die ist insbesondere bei dynamischen Erweiterungen der Kernfunktionalität von Interesse (Kap. 4.3).

Kapitel 4

Anwendung und Nutzung der Kernkonzepte

Im letzten Kapitel wurden die Konzepte und Abstraktionen des Dycos-Kerns vorgestellt. Das zur Verfügung stehende Instrumentarium läßt sich grundsätzlich in zwei Bereiche unterteilen – die grob- und feingranularen Kernbausteine und die elementaren Operationen (Toolset) zur Manipulation der Bausteine. Im folgenden wird nun erläutert, wie mit Hilfe des vorgestellten Instrumentariums Basismechanismen und Abläufe zur stellenlokalen Ressourcenverwaltung realisiert werden können. Die vorgestellten Verfahren sind in ihrem Wirkungsspektrum zunächst stellenbezogen, sie können jedoch weitergehend zur Konstruktion von stellenübergreifenden Managementaufgaben herangezogen werden.

In beiden Fällen kommt den Dycos Kontexten eine Schlüsselrolle zu. Sie eignen sich aufgrund der Kompositionalität und der anpaßungsfähigen Funktionalität dazu, geeignete Basismechanismen und präzise zugeschnittene Anwendungsumgebungen für die Nutzung durch höhere Ressourcenmanager zu kreieren. Für die Erzeugung und Komposition eines derartigen angepaßten Kontextes werden die elementaren Operationen des Kerns schrittweise angewandt. Dabei lassen sich grob drei Bearbeitungsphasen unterteilen, die durchlaufen werden müssen:

(1) Allokation Mit der `Alloc()` Operation werden zunächst Inkarnationen für alle benötigten Kernbausteine (FOs, COs) erzeugt.

(2) Registrierung Durch den Aufruf von `Register()` für jedes FO/CO erhalten diese eindeutige Bezeichner. Gleichzeitig werden hierbei die Standard-Events für den Zugriff auf die von den Bausteinen exportierte Schnittstelle angemeldet. Die Bausteine können nun über die Event-basierte Kommunikation genutzt werden.

(3) Kontextbildung Zur Bildung eines neuen *tagged* Kontextes müssen die dazu notwendigen FOs in den dafür vorgesehenen CO eingebunden werden. Dazu wird die Kernoperation `Attach()` verwendet. Nach dem Einbinden müssen die Kernablaufsequenzen des Kontextes, die sich hinter den exportierten `load()` und `unload()` Methoden verbergen, neu definiert werden. Dies wird ebenfalls durch den Einsatz der `Attach()` Operation erreicht (siehe Kap. 4.2).

Die nachfolgenden Abschnitte gehen näher auf die Verwaltung der Kontext-Abstraktion und die Kombinationsmöglichkeiten ein. Weiterhin werden die dynamischen Rekonfigurationseigenschaften auf Basis der `OpSeqs` verdeutlicht und ein Einblick in das Verfahren der dynamischen Kernerweiterung gegeben.

4.1 Kontexthandling

Der einfache Umgang mit Kontexten setzt zunächst voraus, daß anwendungsseitig die eben erläuterten Phasen durchlaufen wurden, d.h. daß die Kontexte bereits mit Bausteinen gefüllt und in bezug auf eine Anwendung funktional spezifiziert wurden. Typische Beispiele für derartige gestaltete Kontext wurden bereits im vorherigen Kapitel erwähnt und sind nachfolgend nochmals konkretisiert:

ActivationContext Realisiert die Rechenfähigkeit in einem System. Geeignete Bausteine dafür sind `ACB, Stack, PQueue` und ggf. weitere für die Realisierung von Kommunikationskanälen (wie `CStub, MStub, EventHandler`).

MemoryDomainContext Realisiert eine hardware-unterstützte Schutzdomäne um eine Anzahl von Speicherobjekten. Als Bausteine kommen dafür Vertreter der `VMPage, VMGroup` Klasse in Betracht. Ein zugebundener `TLBCache` hat in diesem Kontext eine gesonderte Rolle und dient als Softwarecache in der Hierarchie zwischen der PageTable und der physikalischen MMU/TLB des Prozessors.

CommunicationContext Realisiert eine separate Kommunikationsumgebung, die ausgehend von einem gebundenen ActivationContext genutzt werden kann. Hierfür lassen sich die Bausteine `CStub, MStub, EventHandler` für die Etablierung von synchronen und asynchronen Kommunikationskanälen einbinden. Der `EventHandler` übernimmt dabei die Abarbeitung des Nachrichtenprotokolls.

Diese Beispiele geben nur eine grobe Übersicht darüber, wie die Kontexte im Rahmen der Anwendungsberechnung eingesetzt und genutzt werden können. Falls sich die Randbedingungen der Anwendung während der Berechnungsphase ändern, so besteht natürlich die Möglichkeit, die standardmäßig konstruierten Kontexte dynamisch den neuen Gegebenheiten anzupassen.

4.1.1 Scheduling von Kontexten

Die Validierung eines ausgezeichneten Kontextes erfolgt durch die `SwitchContext()` Operation. Dagegen ist die dynamische Verwaltung der Kontexte (innerhalb einer Managementstrategie) durch das Dycos-Konzept nicht primär festgelegt.

Durch Nutzung von `SwitchContext()` innerhalb des Rechenflusses in einer Aktivität (Activation[1]) oder einem `EventHandler` lassen sich beliebige Schedulingverfahren und -strategien implementieren.

[1]Eine Activation bezeichnet den durch einen Kontext realisierten Ausführungsfaden mit zusätzlichen festlegbaren Eigenschaften

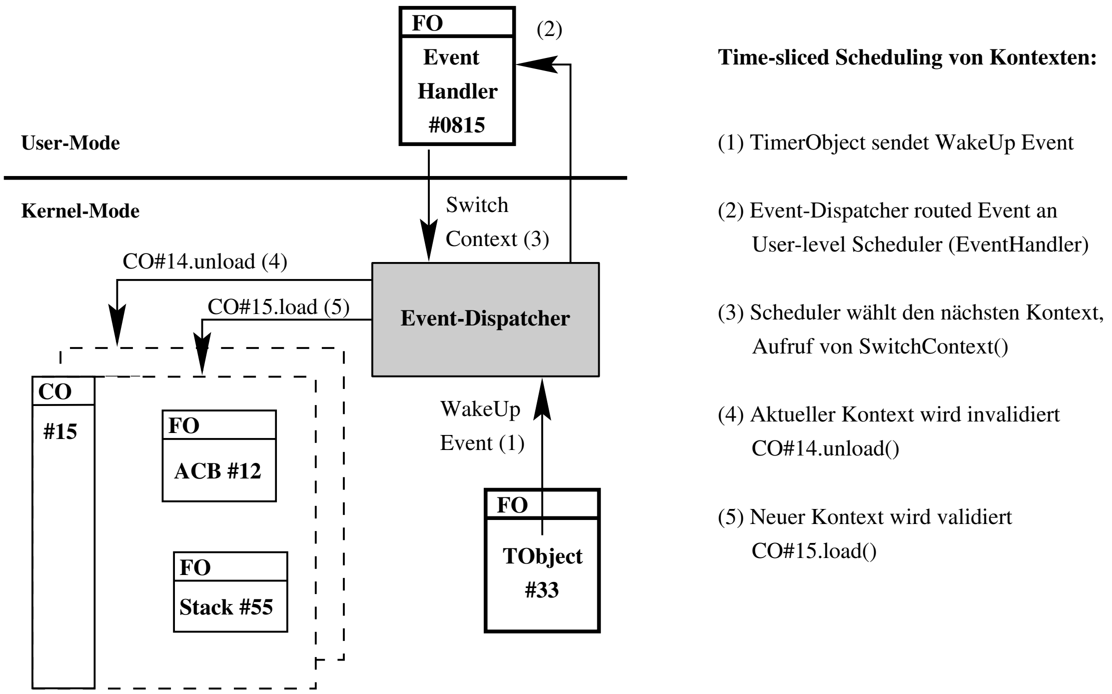

Abbildung 4.1: Implementierung von Schedulingverfahren

Abbildung 4.1 zeigt ein Beispiel, wie ein einfaches periodisches Schedulingverfahren mit einer benutzerdefinierten Schedulingstrategien auf Basis weniger Kernbausteine konstruiert werden kann. Das Bild zeigt zwei Activation-Kontexte (d.h. zwei Aktivitäten), die im priorisierten Prozessormodus (Kernel-Mode) verwaltet werden. Über den periodisch angesprochenen EventHandler, der den benutzerdefinierten Scheduler kapselt und logisch dem Benutzermodus (User-Mode) zugeordnet ist, werden nun die unterschiedlichen Activation-Kontexte für den Kontextwechsel ausgewählt. Dieses Beispiel verdeutlicht nochmal die Dycos charakteristische strikte Trennung zwischen Strategie und Mechanismus.

4.1.2 Realisierung von Anwendungsumgebungen

Für die Realisierung einer komplexen Anwendungsumgebung mit stark wechselnden Ressourcenanforderungen an die Basis ist es sinnvoll, separate Kontexte für diese Ressourcenbereiche anzulegen. Treten beispielsweise stark schwankende Speicher- und Schutzanforderungen, unterschiedliche Kommunikationsaufkommen oder wechselnde Schedulinganforderungen auf, so bringt eine Trennung der ressourcenspezifischen Kernbausteine in unterschiedliche Kontexte erhöhte Flexibilität und deutliche Effizienzgewinne beim Kontextwechsel mit sich. Die Kontexte können bei Bedarf zur Laufzeit gebunden und in bezug auf den Kontextwechsel als Einheit betrachtet werden. Ändern sich Anforderungen während der Berechnung, so ist es möglich, einzelne Kontexte aus dem Verbund herauszunehmen und ggf. durch andere zu ersetzen. So lassen sich beispielsweise, die in INSEL-Systemen auftretenden Akteursphären[2] durch eigene MemoryDomain-

[2]Ein Akteursphäre bezeichnet die Menge der einer abstrakten INSEL-Aktivität (Akteur) konzeptionell zugeordneten passiven Komponenten.

Kontexte realisieren, die abhängig von den Speicheranforderungen des Akteurs unterschiedliche Caching-Strategien für die MMU/TLB Verwaltung beinhalten. Eine weitere Möglichkeit besteht darin, Akteure mit ihrem zugeordneten passiven Manager in eigenen Activation-Kontexten zusammenzufassen. Durch Nutzung der `Register()/Unregister()` Operation lassen sich dann diese Aktivitäten stellenlokal abmelden und (im Rahmen einer Lastverschiebung bzw. eines entfernten Prozeduraufrufs) zu beliebigen entfernten Stellen migrieren.

Durch die Möglichkeit dynamische Kontext-Kontext Bindungen durchzuführen ergibt sich zusammen mit der funktionalen Gestaltung der einzelnen Kontextabläufe ein Spektrum von Realisierungsmöglichkeiten für *differenziert-gewichtete* Kontexte. Abbildung 4.2 zeigt, wie sich dieses Prinzip dazu einsetzen läßt, um Kontextwechselzeiten zu beeinflussen und in bezug auf die von der Anwendung genutzten Ressourcen zu optimieren.

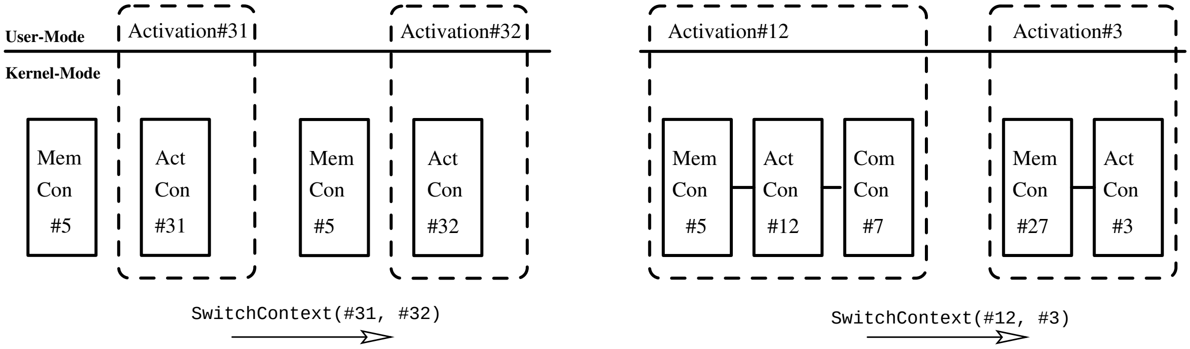

Abbildung 4.2: Leicht- und Schwergewichtiger Kontextwechsel

Für einen schnellen Wechsel von einfachen Aktivitäten innerhalb einer Schutzdomäne eignet sich die Konstruktion links im Bild. Ein Kontextwechsel für einen Kontextverbund aus Activation-, MemoryDomain- und Communication-Kontext, wie rechts zu sehen, ist dagegen deutlich teurer und entspricht in etwa einem klassischen UNIX-Prozeßwechsel.

4.2 Konfiguration von Kernabläufen

Feingranular unterteilte Kernabläufe lassen sich auf Basis der Kontext und EventHandler Objekte definieren. Die Operationen Sequenz Listen `OpSeq` spielen dabei eine Hauptrolle. Jede `OpSeq` kann durch `Attach()` mit einem Event assoziiert werden. Nach dem Empfang dieses Events vom Event-Dispatcher, wird die damit assoziierte `OpSeq` ausgeführt. Auf diese Weise lassen sich durch interne oder externe Event-Signalisierung beliebige Kernablaufsequenzen anstoßen.

Die Einträge in einer `OpSeq` sind wiederum konfigurierbar. Sie werden genutzt, um in der aktuellen Kontextumgebung auf Objektmethoden zuzugreifen und definierte, feinkörnige Berechnungen durchzuführen. Die möglichen Einträge in einer `OpSeq` sind:

- ACE-Aufrufe
- Event-Signalisierungen

- Elementare Operationen
- Modulaufrufe[3] für EventHandler

Eine `OpSeq` wird durch Spezifikation einer spezifischen Datenstruktur festgelegt, die im folgenden durch bei einem `Attach()` Aufruf mit dem Zielobjekt referenziert wird. Auf dieser Grundlage werden z.B. die Kernablaufsequenzen für den Kontextwechsel individuell in jedem Kontext festgelegt. Ein `EventHandler` kann auf dieser Basis eine Event-getriggerte Folge von Aktionen, z.B. zur Bearbeitung eines Protokolls oder einer Verwaltungsstrategie durchführen.

Das (informell aufbereitete) Beispiel einer Operationen Sequenz Liste für die Invalidierung eines einzelnen Activation-Kontextes mit zugeordneten `ACB, Stack` und `PQueue` Objekten ist nachfolgend aufgeführt:

OpSeq [unload]

1. `ACE-Call( ACB.invalidate ); -- Maschinenregistersatz auf Stack sichern`
2. `RaizeEvent( CalcWorkload, PerfAnalyser, ... ); -- Workload berechnen`
3. `ACE-Call( PQueue.dequeue ); -- Kontext in die Ready-Queue einfügen`

Die obige Beispiel `OpSeq` wurde um einen Eintrag erweitert, der nach jeder Kontextinvalidierung einen EventHandler anstößt, der den Prozeß-spezifischen Workload berechnet.

4.3 Erweiterbarkeit und Anpaßbarkeit

Neben den Konfigurationsmöglichkeiten durch die Definition von `OpSeqs` besteht im weiteren noch die Möglichkeit, funktionale Erweiterungen des Kerns durchzuführen. Damit lassen sich weitere, individuelle Anpassungen vollziehen.

Um eine funktionale Erweiterung durchzuführen muß auf einen Vertreter der EventHandler Klasse zurückgegriffen werden. Jeder neu inkarnierte EventHandler ist im initialen Zustand leer und erhält seine Funktionalität erst durch das nachträgliche 'einhängen' von Codemodulen in seine Objekthülle. Die Codemodule selbst sind danach in der Lage, über die allgemeinen Kommunikationsverfahren des Kerns mit der Umgebung zu interagieren, können aber auch von anderen Objekten oder der Anwendung genutzt werden. Dies geschieht grundsätzlich indirekt über `OpSeqs` mit Einträgen vom Typ *Modulaufruf*.

Als Codemodule kommen einerseits diejenigen in Frage, die beim statischen Binden des Kerns bereits zur Verfügung stehen. Andererseits können auf diese Weise auch Codemodule eingebunden werden, die nachträglich, d.h. zur Laufzeit ins System gelangen und als statisch gebundene Codefragmente im Speicher liegen.

Die Codemodule müssen generell folgende Anforderungen erfüllen.

[3] siehe Kap. 4.3

- ⋄ Module müssen statisch-gelinkt sein.
- ⋄ Segmente des Moduls sind an einer festgelegten Adresse in den virtuellen Adreßraum gemappt.
- ⋄ Exportierter Moduleintrittspunkt (Einsprungsadresse).
- ⋄ Vorhandene Prologphase (benutzerspezifisch) in der bei Bedarf die aktuelle Kontextumgebung mittels `QueryInfo()` abgefragt wird.

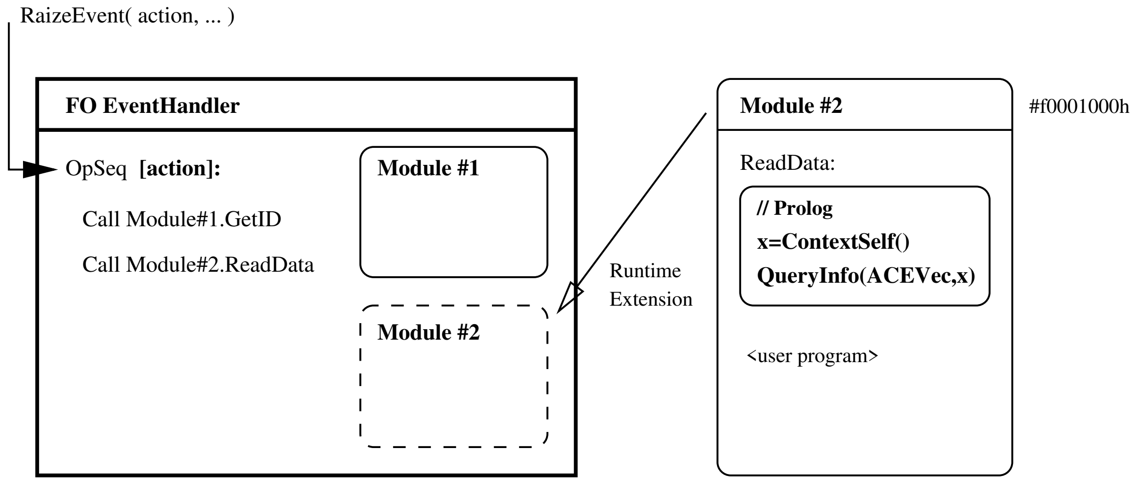

Abbildung 4.3: Erweiterung der Kernfunktionalität

Ein Beispiel für eine Kernerweiterung ist in Abbildung 4.3 veranschaulicht. Das *Module#1* steht seit der Initialisierung des Kerns bereit und wurde im Laufe der Berechnungen dynamisch dem EventHandler zugewiesen. *Module#2* wurde nachträglich übersetzt, statisch gelinkt und an eine feste virtuelle Adresse in den Speicher kopiert. Über die Adresse des Eintrittspunktes `ReadData` kann die Funktion nun beim EventHandler angemeldet werden. Die Nutzung von `ReadData` ist über eine `OpSeq` des Handlers möglich, die mit dem Event `action` assoziiert ist. Die Funktion selbst kann nach Aufruf durch den Handler ihre aktuelle Kontextumgebung abfragen (Prologphase) und auf Basis dieser Daten mit anderen Kernbausteinen interagieren.

Kapitel 5

Softwarearchitektur des Kerns

Die Softwarearchitektur des Kerns ist modular aufgebaut und gliedert sich in einen maschinenabhängigen und einen maschinenunabhängige Bereich. Diese Bereiche sind wiederum weiter unterteilt, wie dies Abbildung 5.1 veranschaulicht.

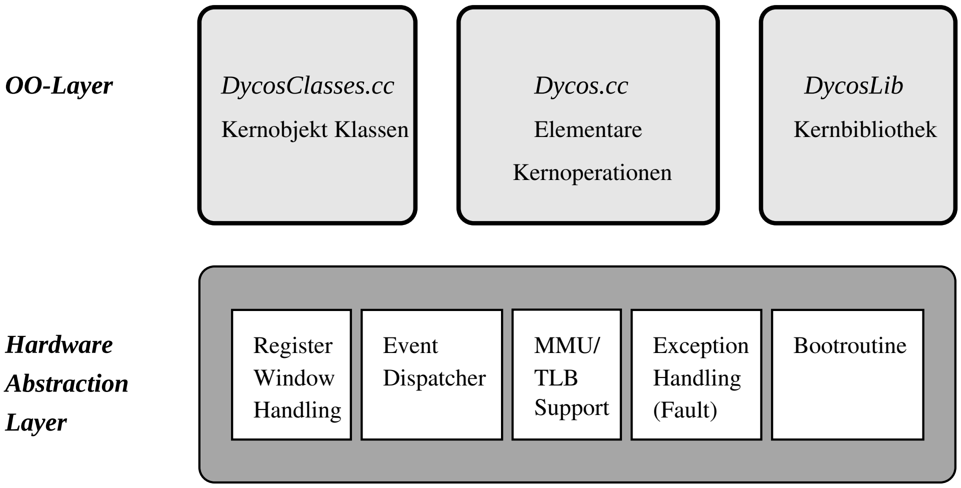

Abbildung 5.1: Dycos Softwarearchitektur im Überblick

Der maschinenunabhängige Teil besteht aus drei Bereiche:

- Hauptmodul (*Dycos.CC*): Hier sind die elementaren Kernoperationen enthalten.
- Kernbaustein-Bibliothek (*DycosClasses.cc*): Hier sind die Objektklassen der FOs und COs definiert.
- Kernfunktions-Bibliothek (*DycosLib*): Hier sind Kernabläufe, Konfigurationsmakros und Programmodule für Aktivitäten und EventHandler definiert, die zur Konstruktion von Basismechanismen dienen.

Die obigen Bereiche bilden die Objekt-orientierte Schicht (OO-Layer) der Architektur. Sie ist weitestgehend in der Sprache C++ implementiert. Die Objektklassen der COs und der FOs wurden als *C++ Klassenbibliothek* realisiert. Damit ist möglich bei Bedarf weitere Kernbaustein-Klassen (off-line) nach dem gleichen Schema zu entwickeln.

Der maschinenabhängige Teil, die Hardware-Abstraktionsschicht (*locore*), kapselt die prozessorspezifischen und zeitkritischen Module des Kerns. Darin sind folgende low-level Funktionen enthalten:

- V9 Register-Window Management
- Event-Dispatcher und Event-Signalisierung
- Software Trap Service Routinen für die allgemeine Ausnahmebehandlung
- MMU/TLB Management
- Bootroutine

Der *locore* ist aus Leistungsgründen größtenteils in Assembler implementiert. Als Zielplattform der Implementierung ist die SPARC V9 Architektur [WG94] von Sun-Microsystems vorgesehen.

Durch die gegebene Software-Modularität und die in Kap. 4.3 dargestellten Erweiterbarkeitskonzepte lassen sich sowohl off-line als auch runtime Erweiterungen der Softwarearchitektur durchführen.

Kapitel 6

Entwicklungsstand

Zur Evaluation der Dycos Kernkonzepte wurde eine prototypische Teilemulation auf Basis des UNIX Betriebssystems Solaris 2.5.1 implementiert [GK97]. DycosEmu umfaßt alle maschinenunabhängigen Objekte bzw. Objektklassen, die zur Konstruktion von Activation-Kontexten benötigt werden. Weiterhin sind Kommunikations- und Synchonisationsobjekte realisiert, so daß bereits eigene Schedulingverfahren und -strategien mit Hilfe der Emulation ausgetestet werden konnten. Auch die Spezifikation neuer Kernabläufe durch Definition von eigenen `OpSeqs` konnte so realisiert werden. Die Emulation kann auf zwei unterschiedliche Arten genutzt werden. Zum einen kann die Emulationsbibliothek direkt zu einem Anwendungsprogramm (in C, C++) gebunden werden. Auf diese Weise lassen sich spezifische Dycos-Programme entwickeln, die Kernfunktionen der Emulation aufrufen und nutzen. Zum anderen wurde eine graphische Fensteroberfläche entworfen, die es ermöglicht, interaktiv mit den elementaren Kernoperationen zu arbeiten. Diese Simulator-ähnliche Oberfläche gestattet es, Kernbausteine 'von Hand' zu komponieren und Kontexte zu erstellen. Abb. 6.1 zeigt einen Bildschirmausschnitt der DycosEmu Menüschnittstelle.

Auf Grundlage der Emulation lassen sich nun in weiteren Arbeitsschritten die für V-PK-Systeme notwendigen angepaßten Basismechanismen und -funktionen implementieren. Gleichzeitig bietet DycosEmu ein gute Ausgangsbasis für Weiterentwicklungen an der Kernarchitektur, da hierbei eine Benutzermodus-realisierte Testumgebung besteht und Änderungen mit relativ geringem Aufwand auf die langfristig geplante native UltraSPARC Portierung übertragen werden können.

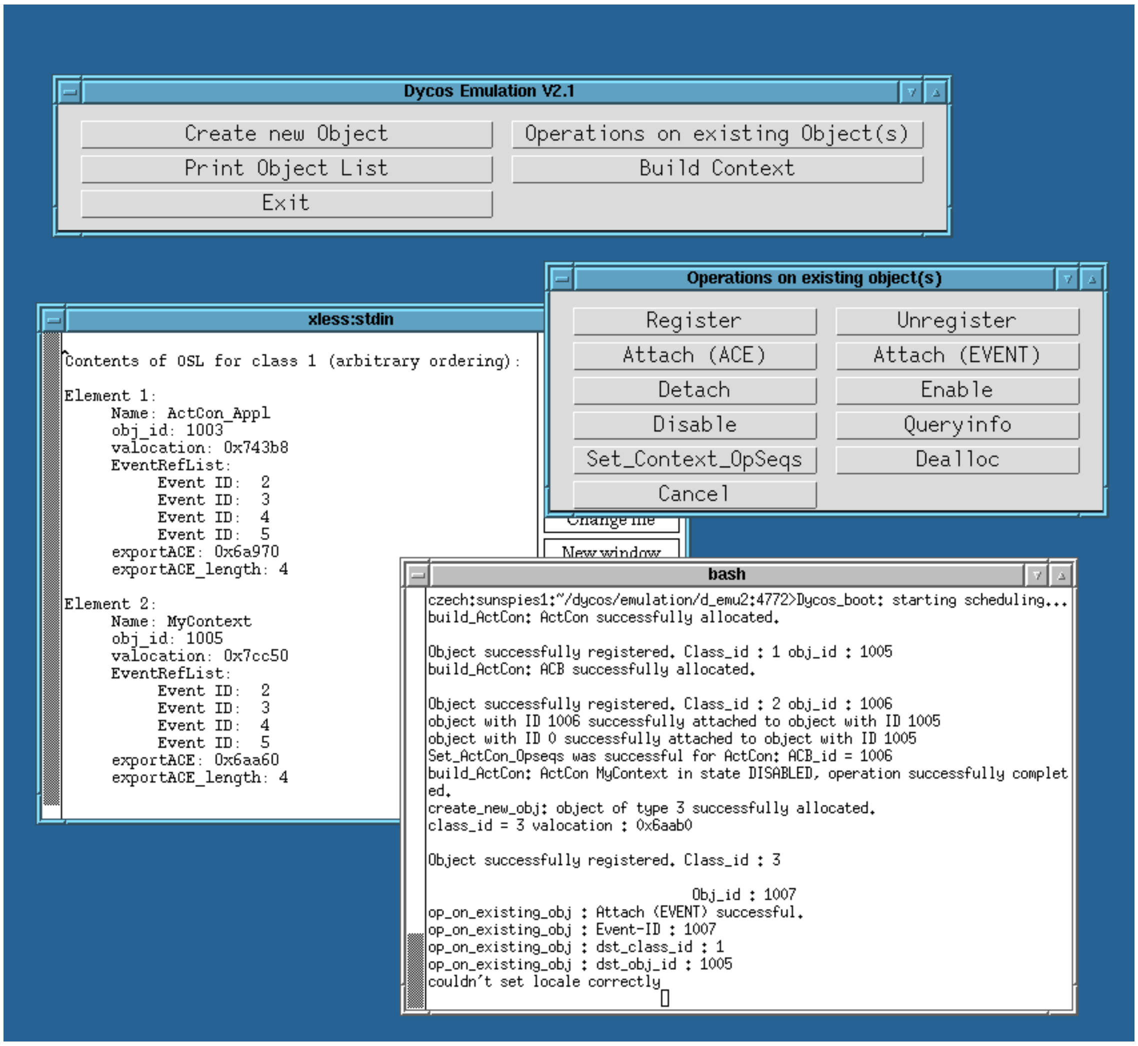

Abbildung 6.1: Graphische Oberfläche von DycosEmu

Literaturverzeichnis

[ABLL92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Transactions on Computer Systems*, pages 53–79, February 1992.

[B+92] D. Black et al. Microkernel operating system architectures and MACH. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architctures*, pages 11–30, April 1992.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safty and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, Colorado, December 1995.

[CB93] J. B. Chen and B. N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 120–133, Asheville, NC, 1993.

[CD94] D. R. Cheriton and 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.

[Cho92] The Chorus Team. Overview of the Chorus distributed operating system. In *USENIX Workshop on Microkernels and other Kernel-Architectures*, Seattle, WA, 1992.

[CLBL92] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA, March 1992.

[EKO95] D. R. Engler, M. F. Kaashoek, and 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, December 1995.

[F+96] B. Ford et al. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the OSDI 1996*, October 1996.

[FHL94] B. Ford, M. Hibler, and J. Lepreau. Evolving Mach 3.0 to a Migrating Threads Model. In *Proceedings of the USENIX Winter Conference*, 1994.

[GK97] M. Gwinner and K. Krüger-Barvels. Emulation des Dycos–Kerns. Fortgeschrittenenpraktikum, TU-München, Januar 1997.

[HK93] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the 1993 Summer Usenix Conference*, June 1993.

[Lie95] J. Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, Colorado, December 1995.

[Loe91] K. Loepere. MACH3 Kernel Principles. Technical Report, Open Software Foundation and Carnegie Mellon University, 1991.

[OS92] B. Ozden and A. Silberschatz. A taxonomy of shared address space systems. Technical Report TR–92–33, University of Texas at Austin, July 1992.

[Ous90] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware. In *Usenix Summer Conference*, pages 247–256, Anaheim, California, June 1990.

[SESS94] M. Seltzer, Y. Endo, C. Small, and K. Smith. An Introduction to the Architecture of the VINO-kernel. Technical Report TR-34-94, Computer Science, Harvard University, 1994.

[T^+90] A.S. Tanenbaum et al. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33:46–63, Dec 1990.

[TRC95] S.-M. Tan, D. K. Raila, and R. H. Campbell. A Case for Nano-Kernels. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[vDHT95] L. van Doorn, P. Homburg, and A. S. Tanenbaum. Paramecium: An extensible object-based kernel. In *Proceedings of the 5th Hot Topics on Operating Systems (HotOS) workshop*, Orcas Island, WA, May 1995.

[WG94] D. L. Weaver and T. Germond. *The SPARC Architecture Manual, Version 9.* SPARC International, Mountain View, CA, 2nd edition, 1994.

[WLAG93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th SOSP*, Asheville, NC, December 1993.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

342/1/90 A	Robert Gold, Walter Vogler: Quality Criteria for Partial Order Semantics of Place/Transition-Nets, Januar 1990
342/2/90 A	Reinhard Fößmeier: Die Rolle der Lastverteilung bei der numerischen Parallelprogrammierung, Februar 1990
342/3/90 A	Klaus-Jörn Lange, Peter Rossmanith: Two Results on Unambiguous Circuits, Februar 1990
342/4/90 A	Michael Griebel: Zur Lösung von Finite-Differenzen- und Finite-Element-Gleichungen mittels der Hierarchischen Transformations-Mehrgitter-Methode
342/5/90 A	Reinhold Letz, Johann Schumann, Stephan Bayerl, Wolfgang Bibel: SETHEO: A High-Performance Theorem Prover
342/6/90 A	Johann Schumann, Reinhold Letz: PARTHEO: A High Performance Parallel Theorem Prover
342/7/90 A	Johann Schumann, Norbert Trapp, Martin van der Koelen: SETHEO/PARTHEO Users Manual
342/8/90 A	Christian Suttner, Wolfgang Ertel: Using Connectionist Networks for Guiding the Search of a Theorem Prover
342/9/90 A	Hans-Jörg Beier, Thomas Bemmerl, Arndt Bode, Hubert Ertl, Olav Hansen, Josef Haunerdinger, Paul Hofstetter, Jaroslav Kremenek, Robert Lindhof, Thomas Ludwig, Peter Luksch, Thomas Treml: TOPSYS, Tools for Parallel Systems (Artikelsammlung)
342/10/90 A	Walter Vogler: Bisimulation and Action Refinement
342/11/90 A	Jörg Desel, Javier Esparza: Reachability in Reversible Free- Choice Systems
342/12/90 A	Rob van Glabbeek, Ursula Goltz: Equivalences and Refinement
342/13/90 A	Rob van Glabbeek: The Linear Time - Branching Time Spectrum
342/14/90 A	Johannes Bauer, Thomas Bemmerl, Thomas Treml: Leistungsanalyse von verteilten Beobachtungs- und Bewertungswerkzeugen
342/15/90 A	Peter Rossmanith: The Owner Concept for PRAMs
342/16/90 A	G. Böckle, S. Trosch: A Simulator for VLIW-Architectures
342/17/90 A	P. Slavkovsky, U. Rüde: Schnellere Berechnung klassischer Matrix-Multiplikationen

Reihe A

342/18/90 A Christoph Zenger: SPARSE GRIDS
342/19/90 A Michael Griebel, Michael Schneider, Christoph Zenger: A combination technique for the solution of sparse grid problems
342/20/90 A Michael Griebel: A Parallelizable and Vectorizable Multi- Level-Algorithm on Sparse Grids
342/21/90 A V. Diekert, E. Ochmanski, K. Reinhardt: On confluent semi-commutations-decidability and complexity results
342/22/90 A Manfred Broy, Claus Dendorfer: Functional Modelling of Operating System Structures by Timed Higher Order Stream Processing Functions
342/23/90 A Rob van Glabbeek, Ursula Goltz: A Deadlock-sensitive Congruence for Action Refinement
342/24/90 A Manfred Broy: On the Design and Verification of a Simple Distributed Spanning Tree Algorithm
342/25/90 A Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Peter Luksch, Roland Wismüller: TOPSYS - Tools for Parallel Systems (User's Overview and User's Manuals)
342/26/90 A Thomas Bemmerl, Arndt Bode, Thomas Ludwig, Stefan Tritscher: MMK - Multiprocessor Multitasking Kernel (User's Guide and User's Reference Manual)
342/27/90 A Wolfgang Ertel: Random Competition: A Simple, but Efficient Method for Parallelizing Inference Systems
342/28/90 A Rob van Glabbeek, Frits Vaandrager: Modular Specification of Process Algebras
342/29/90 A Rob van Glabbeek, Peter Weijland: Branching Time and Abstraction in Bisimulation Semantics
342/30/90 A Michael Griebel: Parallel Multigrid Methods on Sparse Grids
342/31/90 A Rolf Niedermeier, Peter Rossmanith: Unambiguous Simulations of Auxiliary Pushdown Automata and Circuits
342/32/90 A Inga Niepel, Peter Rossmanith: Uniform Circuits and Exclusive Read PRAMs
342/33/90 A Dr. Hermann Hellwagner: A Survey of Virtually Shared Memory Schemes
342/1/91 A Walter Vogler: Is Partial Order Semantics Necessary for Action Refinement?
342/2/91 A Manfred Broy, Frank Dederichs, Claus Dendorfer, Rainer Weber: Characterizing the Behaviour of Reactive Systems by Trace Sets
342/3/91 A Ulrich Furbach, Christian Suttner, Bertram Fronhöfer: Massively Parallel Inference Systems
342/4/91 A Rudolf Bayer: Non-deterministic Computing, Transactions and Recursive Atomicity

Reihe A

342/5/91 A	Robert Gold: Dataflow semantics for Petri nets
342/6/91 A	A. Heise; C. Dimitrovici: Transformation und Komposition von P/T-Netzen unter Erhaltung wesentlicher Eigenschaften
342/7/91 A	Walter Vogler: Asynchronous Communication of Petri Nets and the Refinement of Transitions
342/8/91 A	Walter Vogler: Generalized OM-Bisimulation
342/9/91 A	Christoph Zenger, Klaus Hallatschek: Fouriertransformation auf dünnen Gittern mit hierarchischen Basen
342/10/91 A	Erwin Loibl, Hans Obermaier, Markus Pawlowski: Towards Parallelism in a Relational Database System
342/11/91 A	Michael Werner: Implementierung von Algorithmen zur Kompaktifizierung von Programmen für VLIW-Architekturen
342/12/91 A	Reiner Müller: Implementierung von Algorithmen zur Optimierung von Schleifen mit Hilfe von Software-Pipelining Techniken
342/13/91 A	Sally Baker, Hans-Jörg Beier, Thomas Bemmerl, Arndt Bode, Hubert Ertl, Udo Graf, Olav Hansen, Josef Haunerdinger, Paul Hofstetter, Rainer Knödlseder, Jaroslav Kremenek, Siegfried Langenbuch, Robert Lindhof, Thomas Ludwig, Peter Luksch, Roy Milner, Bernhard Ries, Thomas Treml: TOPSYS - Tools for Parallel Systems (Artikelsammlung); 2., erweiterte Auflage
342/14/91 A	Michael Griebel: The combination technique for the sparse grid solution of PDE's on multiprocessor machines
342/15/91 A	Thomas F. Gritzner, Manfred Broy: A Link Between Process Algebras and Abstract Relation Algebras?
342/16/91 A	Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Thomas Treml, Roland Wismüller: The Design and Implementation of TOPSYS
342/17/91 A	Ulrich Furbach: Answers for disjunctive logic programs
342/18/91 A	Ulrich Furbach: Splitting as a source of parallelism in disjunctive logic programs
342/19/91 A	Gerhard W. Zumbusch: Adaptive parallele Multilevel-Methoden zur Lösung elliptischer Randwertprobleme
342/20/91 A	M. Jobmann, J. Schumann: Modelling and Performance Analysis of a Parallel Theorem Prover
342/21/91 A	Hans-Joachim Bungartz: An Adaptive Poisson Solver Using Hierarchical Bases and Sparse Grids
342/22/91 A	Wolfgang Ertel, Theodor Gemenis, Johann M. Ph. Schumann, Christian B. Suttner, Rainer Weber, Zongyan Qiu: Formalisms and Languages for Specifying Parallel Inference Systems
342/23/91 A	Astrid Kiehn: Local and Global Causes

Reihe A

342/24/91 A	Johann M.Ph. Schumann: Parallelization of Inference Systems by using an Abstract Machine
342/25/91 A	Eike Jessen: Speedup Analysis by Hierarchical Load Decomposition
342/26/91 A	Thomas F. Gritzner: A Simple Toy Example of a Distributed System: On the Design of a Connecting Switch
342/27/91 A	Thomas Schnekenburger, Andreas Weininger, Michael Friedrich: Introduction to the Parallel and Distributed Programming Language ParMod-C
342/28/91 A	Claus Dendorfer: Funktionale Modellierung eines Postsystems
342/29/91 A	Michael Griebel: Multilevel algorithms considered as iterative methods on indefinite systems
342/30/91 A	W. Reisig: Parallel Composition of Liveness
342/31/91 A	Thomas Bemmerl, Christian Kasperbauer, Martin Mairandres, Bernhard Ries: Programming Tools for Distributed Multiprocessor Computing Environments
342/32/91 A	Frank Leßke: On constructive specifications of abstract data types using temporal logic
342/1/92 A	L. Kanal, C.B. Suttner (Editors): Informal Proceedings of the Workshop on Parallel Processing for AI
342/2/92 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: The Design of Distributed Systems - An Introduction to FOCUS
342/2-2/92 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: The Design of Distributed Systems - An Introduction to FOCUS - Revised Version (erschienen im Januar 1993)
342/3/92 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems
342/4/92 A	Claus Dendorfer, Rainer Weber: Development and Implementation of a Communication Protocol - An Exercise in FOCUS
342/5/92 A	Michael Friedrich: Sprachmittel und Werkzeuge zur Unterstüt- zung paralleler und verteilter Programmierung
342/6/92 A	Thomas F. Gritzner: The Action Graph Model as a Link between Abstract Relation Algebras and Process-Algebraic Specifications
342/7/92 A	Sergei Gorlatch: Parallel Program Development for a Recursive Numerical Algorithm: a Case Study
342/8/92 A	Henning Spruth, Georg Sigl, Frank Johannes: Parallel Algorithms for Slicing Based Final Placement
342/9/92 A	Herbert Bauer, Christian Sporrer, Thomas Krodel: On Distributed Logic Simulation Using Time Warp

Reihe A

342/10/92 A H. Bungartz, M. Griebel, U. Rüde: Extrapolation, Combination and Sparse Grid Techniques for Elliptic Boundary Value Problems

342/11/92 A M. Griebel, W. Huber, U. Rüde, T. Störtkuhl: The Combination Technique for Parallel Sparse-Grid-Preconditioning and -Solution of PDEs on Multiprocessor Machines and Workstation Networks

342/12/92 A Rolf Niedermeier, Peter Rossmanith: Optimal Parallel Algorithms for Computing Recursively Defined Functions

342/13/92 A Rainer Weber: Eine Methodik für die formale Anforderungsspezifkation verteilter Systeme

342/14/92 A Michael Griebel: Grid– and point–oriented multilevel algorithms

342/15/92 A M. Griebel, C. Zenger, S. Zimmer: Improved multilevel algorithms for full and sparse grid problems

342/16/92 A J. Desel, D. Gomm, E. Kindler, B. Paech, R. Walter: Bausteine eines kompositionalen Beweiskalküls für netzmodellierte Systeme

342/17/92 A Frank Dederichs: Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen

342/18/92 A Andreas Listl, Markus Pawlowski: Parallel Cache Management of a RDBMS

342/19/92 A Erwin Loibl, Markus Pawlowski, Christian Roth: PART: A Parallel Relational Toolbox as Basis for the Optimization and Interpretation of Parallel Queries

342/20/92 A Jörg Desel, Wolfgang Reisig: The Synthesis Problem of Petri Nets

342/21/92 A Robert Balder, Christoph Zenger: The d-dimensional Helmholtz equation on sparse Grids

342/22/92 A Ilko Michler: Neuronale Netzwerk-Paradigmen zum Erlernen von Heuristiken

342/23/92 A Wolfgang Reisig: Elements of a Temporal Logic. Coping with Concurrency

342/24/92 A T. Störtkuhl, Chr. Zenger, S. Zimmer: An asymptotic solution for the singularity at the angular point of the lid driven cavity

342/25/92 A Ekkart Kindler: Invariants, Compositionality and Substitution

342/26/92 A Thomas Bonk, Ulrich Rüde: Performance Analysis and Optimization of Numerically Intensive Programs

342/1/93 A M. Griebel, V. Thurner: The Efficient Solution of Fluid Dynamics Problems by the Combination Technique

342/2/93 A Ketil Stølen, Frank Dederichs, Rainer Weber: Assumption / Commitment Rules for Networks of Asynchronously Communicating Agents

342/3/93 A Thomas Schnekenburger: A Definition of Efficiency of Parallel Programs in Multi-Tasking Environments

342/4/93 A Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, Christoph Zenger: A Proof of Convergence for the Combination Technique for the Laplace Equation Using Tools of Symbolic Computation

Reihe A

342/5/93 A	Manfred Kunde, Rolf Niedermeier, Peter Rossmanith: Faster Sorting and Routing on Grids with Diagonals
342/6/93 A	Michael Griebel, Peter Oswald: Remarks on the Abstract Theory of Additive and Multiplicative Schwarz Algorithms
342/7/93 A	Christian Sporrer, Herbert Bauer: Corolla Partitioning for Distributed Logic Simulation of VLSI Circuits
342/8/93 A	Herbert Bauer, Christian Sporrer: Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving
342/9/93 A	Peter Slavkovsky: The Visibility Problem for Single-Valued Surface (z = f(x,y)): The Analysis and the Parallelization of Algorithms
342/10/93 A	Ulrich Rüde: Multilevel, Extrapolation, and Sparse Grid Methods
342/11/93 A	Hans Regler, Ulrich Rüde: Layout Optimization with Algebraic Multigrid Methods
342/12/93 A	Dieter Barnard, Angelika Mader: Model Checking for the Modal Mu-Calculus using Gauß Elimination
342/13/93 A	Christoph Pflaum, Ulrich Rüde: Gauß' Adaptive Relaxation for the Multilevel Solution of Partial Differential Equations on Sparse Grids
342/14/93 A	Christoph Pflaum: Convergence of the Combination Technique for the Finite Element Solution of Poisson's Equation
342/15/93 A	Michael Luby, Wolfgang Ertel: Optimal Parallelization of Las Vegas Algorithms
342/16/93 A	Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, Christoph Zenger: Pointwise Convergence of the Combination Technique for Laplace's Equation
342/17/93 A	Georg Stellner, Matthias Schumann, Stefan Lamberts, Thomas Ludwig, Arndt Bode, Martin Kiehl und Rainer Mehlhorn: Developing Multicomputer Applications on Networks of Workstations Using NXLib
342/18/93 A	Max Fuchs, Ketil Stølen: Development of a Distributed Min/Max Component
342/19/93 A	Johann K. Obermaier: Recovery and Transaction Management in Write-optimized Database Systems
342/20/93 A	Sergej Gorlatch: Deriving Efficient Parallel Programs by Systemating Coarsing Specification Parallelism
342/01/94 A	Reiner Hüttl, Michael Schneider: Parallel Adaptive Numerical Simulation
342/02/94 A	Henning Spruth, Frank Johannes: Parallel Routing of VLSI Circuits Based on Net Independency
342/03/94 A	Henning Spruth, Frank Johannes, Kurt Antreich: PHIroute: A Parallel Hierarchical Sea-of-Gates Router

Reihe A

342/04/94 A Martin Kiehl, Rainer Mehlhorn, Matthias Schumann: Parallel Multiple Shooting for Optimal Control Problems Under NX/2

342/05/94 A Christian Suttner, Christoph Goller, Peter Krauss, Klaus-Jörn Lange, Ludwig Thomas, Thomas Schnekenburger: Heuristic Optimization of Parallel Computations

342/06/94 A Andreas Listl: Using Subpages for Cache Coherency Control in Parallel Database Systems

342/07/94 A Manfred Broy, Ketil Stølen: Specification and Refinement of Finite Dataflow Networks - a Relational Approach

342/08/94 A Katharina Spies: Funktionale Spezifikation eines Kommunikationsprotokolls

342/09/94 A Peter A. Krauss: Applying a New Search Space Partitioning Method to Parallel Test Generation for Sequential Circuits

342/10/94 A Manfred Broy: A Functional Rephrasing of the Assumption/Commitment Specification Style

342/11/94 A Eckhardt Holz, Ketil Stølen: An Attempt to Embed a Restricted Version of SDL as a Target Language in Focus

342/12/94 A Christoph Pflaum: A Multi-Level-Algorithm for the Finite-Element-Solution of General Second Order Elliptic Differential Equations on Adaptive Sparse Grids

342/13/94 A Manfred Broy, Max Fuchs, Thomas F. Gritzner, Bernhard Schätz, Katharina Spies, Ketil Stølen: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems

342/14/94 A Maximilian Fuchs: Technologieabhängigkeit von Spezifikationen digitaler Hardware

342/15/94 A M. Griebel, P. Oswald: Tensor Product Type Subspace Splittings And Multilevel Iterative Methods For Anisotropic Problems

342/16/94 A Gheorghe Ştefănescu: Algebra of Flownomials

342/17/94 A Ketil Stølen: A Refinement Relation Supporting the Transition from Unbounded to Bounded Communication Buffers

342/18/94 A Michael Griebel, Tilman Neuhoeffer: A Domain-Oriented Multilevel Algorithm-Implementation and Parallelization

342/19/94 A Michael Griebel, Walter Huber: Turbulence Simulation on Sparse Grids Using the Combination Method

342/20/94 A Johann Schumann: Using the Theorem Prover SETHEO for verifying the development of a Communication Protocol in FOCUS - A Case Study -

342/01/95 A Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids

342/02/95 A Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law

342/03/95 A Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space

Reihe A

342/04/95 A Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes

342/05/95 A Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems

342/06/95 A Maximilian Fuchs: Formal Design of a Model-N Counter

342/07/95 A Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology

342/08/95 A Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies

342/09/95 A Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness

342/10/95 A Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design

342/11/95 A Thomas Schnekenburger: The ALDY Load Distribution System

342/12/95 A Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm

342/13/95 A Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming

342/14/95 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks

342/15/95 A Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs

342/16/95 A Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik

342/17/95 A Georg Stellner: Using CoCheck on a Network of Workstations

342/18/95 A Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications

342/19/95 A Thomas Schnekenburger: Integration of Load Distribution into ParMod-C

342/20/95 A Ketil Stølen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication

342/21/95 A Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control

342/22/95 A Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes

342/23/95 A Petr Jančar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation

342/24/95 A M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems

Reihe A

342/01/96 A Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries

342/02/96 A Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs

342/03/96 A Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes

342/04/96 A Thomas Huckle: Efficient Computation of Sparse Approximate Inverses

342/05/96 A Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS — On-line Monitoring Interface Specification

342/06/96 A Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components

342/07/96 A Richard Mayr: Some Results on Basic Parallel Processes

342/08/96 A Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht

342/09/96 A P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme

342/10/96 A Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PFS-Lib – A File System for Parallel Programming Environments

342/11/96 A Manfred Broy, Gheorghe Ştefănescu: The Algebra of Stream Processing Functions

342/12/96 A Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete

342/13/96 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Many-to-Many Data-flow Networks

342/14/96 A Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project

342/15/96 A Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time μ -Calculus

342/16/96 A Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen

342/17/96 A Richard Mayr: Model Checking PA-Processes

342/18/96 A Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States

342/01/97 A Tobias Müller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsfähigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken

Reihe A

342/02/97 A Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations

342/03/97 A Bernhard Mitschang: Technologie für Parallele Datenbanken - Bericht zum Workshop

342/04/97 A nicht erschienen

342/05/97 A Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik

342/06/97 A Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump

342/07/97 A Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming

342/08/97 A Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement

342/09/97 A Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic

342/10/97 A Manfred Broy, Franz Regensburger, Bernhard Schätz, Katharina Spies: The Steamboiler Specification - A Case Study in Focus

342/11/97 A Christine Röckl: How to Make Substitution Preserve Strong Bisimilarity

342/12/97 A Christian B. Czech: Architektur und Konzept des Dycos-Kerns

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B	Wolfgang Reisig: Petri Nets and Algebraic Specifications
342/2/90 B	Jörg Desel: On Abstraction of Nets
342/3/90 B	Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems
342/4/90 B	Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme
342/1/91 B	Barbara Paech1: Concurrency as a Modality
342/2/91 B	Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox - Anwenderbeschreibung
342/3/91 B	Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen
342/4/91 B	Werner Pohlmann: A Limitation of Distributed Simulation Methods
342/5/91 B	Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis
342/6/91 B	Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme
342/7/91 B	W. Reisig: Concurrent Temporal Logic
342/1/92 B	Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support
342/2/92 B	Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen
342/1/93 B	Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung
342/2/93 B	Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick
342/1/94 B	Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS