

## Das PM Projekt

F. Hauck, T. Eirich, M. Fäustle,  
J. Kleinöder, R. Pruy, P. Schlenk

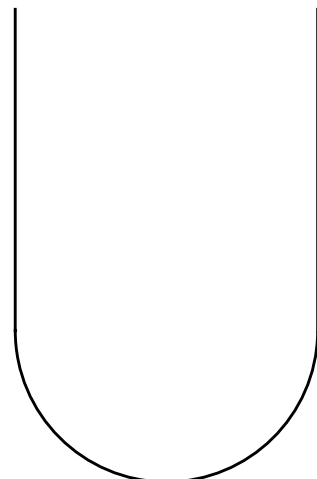
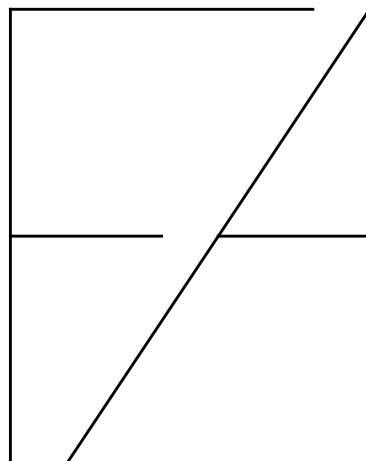
November 1991

TR-I4-6-91

## Interner Bericht

Institut für  
Mathematische Maschinen  
und Datenverarbeitung  
der  
Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Lehrstuhl für Informatik IV  
(Betriebssysteme)



---

# Das PM Projekt

*Franz J. Hauck, Thomas Eirich, Michael Fäustle, Jürgen Kleinöder, Rainer Pruy, Peter Schlenk*

*IMMD 4, Universität Erlangen-Nürnberg  
Martensstraße 1, D-W 8520 Erlangen*

*{Hauck, Eirich, Faeustle, Kleinoeder, Pruy, Schlenk}@immd4.informatik.uni-erlangen.de*

## 1. Einführung

Das PM Projekt wurde im Jahre 1988 von M. Kaiserswerth und P. Schlenk als Konzept zur Strukturierung paralleler und verteilter Systeme gegründet [Kaiserswerth88], [Kaiserswerth89]. PM stand damals für Parallele Module oder auch für die Initialen der Gründer. Das Projekt wurde im Rahmen des Sonderforschungsbereichs 182<sup>1</sup> ins Leben gerufen. Dessen Titel lautet "Multiprozessor- und Netzwerkkonfigurationen". Der Lehrstuhl IMMD 4 ist mit dem Teilprojekt B2 am SFB beteiligt. Das Thema des Teilprojekts lautet "Entwurf und Implementierung eines an Hardwarearchitektur und Aufgabenklassen adaptierbaren Multiprozessorbetriebssystems".

Dieser Artikel soll die ursprünglichen Ideen von PM aufzeigen und deren Entwicklung bis heute nachzeichnen. Weiterhin sollen die bisherigen Erkenntnisse zusammen mit den notwendigen Forschungsschwerpunkten für die nächste Zeit dargelegt werden.

Das Kapitel 1 wird PM einführen. Kapitel 2 wird die Arbeit an einer prototypischen Implementierung aufzeigen und Kapitel 3 unsere Erkenntnisse zusammenfassen und künftige Forschungsschwerpunkte skizzieren. Kapitel 4 schließt mit einer Zusammenfassung der Entwicklung von PM.

### 1.1 PM, ein verteiltes System

Die Notwendigkeit für echte verteilte Systeme scheint immer mehr zu wachsen. Die Anzahl der Workstations und Computersysteme im Allgemeinen steigt ständig. Meist sind diese Systeme miteinander verbunden, z.B. in einem lokalen Netzwerk. Oft wird diese Verbindung aber nicht maximal genutzt. Die Maschinen sind nur zu Bruchteilen ausgelastet und die Benutzer haben wenig oder gar keine Fehlertoleranzeigenschaft, wenn eine Workstation abstürzt, usw.

Ein verteiltes System würde die Nutzung von miteinander verbundenen Maschinen erheblich verbessern. Die größten Vorteile, die man sich von einem verteilten System erhofft, sind die folgenden:

- *Gemeinsames Nutzen von Ressourcen und Lastverteilung*  
Das bedeutet bessere Auslastung und hoffentlich weniger Kosten.

---

<sup>1</sup> Ein Sonderforschungsbereich, abgekürzt SFB, ist ein Forschungsprogramm, das von der *Deutschen Forschungsgemeinschaft* DFG gefördert wird.

- *Verfügbarkeit und Fehlertoleranz*  
Das System könnte in der Lage sein, mehr Fehler in Hard- und Software zu tolerieren als heutzutage. Zunächst wird durch die Verteilung die Ausfallwahrscheinlichkeit zwar erhöht, doch durch entsprechende softwaregestützte Fehlertoleranzmechanismen ist die Ausfallwahrscheinlichkeit erheblich reduzierbar.
- *Skalierbarkeit und Graceful Degradation*  
Es ist leicht das System zu erweitern und andererseits fällt das System nicht im Gesamten aus, sondern es läuft mit weniger Kapazität weiter.

PM soll ein verteiltes System sein, das die aufgezeigten Vorteile der verteilten Programmierung unterstützt. Dabei steht keine besondere Anwendung des verteilten Systems im Vordergrund. Vielmehr ist das System zunächst für einen Vielseitigbetrieb gedacht, wie er heutzutage z.B. durch das Betriebssystem UNIX abgedeckt wird. Erklärtes Ziel ist aber, das System so aufzubauen und zu strukturieren, daß es sich leicht an verschiedene Aufgabenklassen adaptieren läßt, soweit dies überhaupt möglich ist.

Verteilte Systeme sind ganz automatisch parallele Systeme. PM wird also für Parallelität, Multiuser- und Multiprogrammbetrieb ausgelegt sein müssen.

Ein verteiltes System kann aus verschiedenen Architekturen bestehen. Es kann Multiprozessoren neben Einfachprozessoren geben und die Maschinen selbst können verschiedene Prozessortypen enthalten. Das bedeutet, daß verteilte Systeme mit dem Problem der Heterogenität fertig werden müssen.

Die Kommunikation zwischen den Komponenten eines verteilten Systems stellt normalerweise einen Flaschenhals dar, weil innerhalb einer Komponente erheblich effizienter kommuniziert werden kann. Die Komponenten der Software müssen daher so aufgebaut sein, daß sie eine gewisse Lokalitätseigenschaft besitzen, die nur geringe Interaktionen zwischen den Komponenten zur Folge hat. Diese können dann leichter auf die physikalische Struktur des Systems abgebildet werden.

## 1.2 PM, ein objektorientiertes System

Das objektorientierte Programmierparadigma scheint für verteiltes Programmieren besonders geeignet zu sein, stellen doch objektorientiert strukturierte Programme genau die Softwarekomponenten dar, die die verlangte Lokalitätseigenschaft besitzen. Objekte interagieren miteinander über Schnittstellen durch Nachrichtenaustausch. Die Objekte und die Schnittstellen sind bei gutem Design so gewählt, daß die Interaktion entsprechend gering ist.

Objektorientierte Programme stellen damit ein bereits logisch verteiltes Programm dar, das lediglich auf das physikalisch verteilte System abgebildet werden muß.

Objektorientiertes Programmieren bringt auch hinreichend bekannte Vorteile für das Softwareengineering im Allgemeinen. So stellen Objekte eine Kapselheit

dar, die nur durch die nach außen bekannte Schnittstelle durchbrochen werden kann. Diese Schnittstelle abstrahiert zunächst von den im Objekt gespeicherten Daten (*Datenabstraktion*) und von dessen Verhalten (*Funktionsabstraktion*).

Das Vorhandensein einer Schnittstelle erlaubt es, Objekte auszutauschen ohne das Gesamtsystem zu beeinträchtigen, vorausgesetzt die Schnittstelle wird mit gleichem nach außen sichtbarem Verhalten von dem neuen Objekt angeboten. Dies kommt der Wartung eines verteilten Systems und auch der Adaptierbarkeit des Systems an verschiedene Umgebungen zu gute.

Viele Vorteile erhofft man sich durch das Wiederverwenden von Objekten in einem objektorientierten System, z.B. schnellere Entwicklungszeiten und kleinerer Codeumfang, doch dies muß sich erst in der Praxis erweisen.

### 1.3 PM, ein großes System

Ein verteiltes System kann sehr große Ausmaße annehmen. Es ist nicht daran gedacht PM nur für eine kleine physikalische Struktur zu konzipieren, wie sie z.B. ein lokales Netzwerk darstellt.

Je größer ein verteiltes System ist, desto mehr Vorteile können daraus gezogen werden. So kann man sich vorstellen, Ressourcen aus dem amerikanischen Raum transparent in Europa benutzen zu können oder als Anwender aus Deutschland in Hawaii transparent auf seine Daten zuzugreifen, d.h. in gleicher gewohnter Weise.

Eine solche Größe macht Wartungsarbeiten ohne Abschaltung des Gesamtsystems zu einem absoluten Muß. Wie bereits erwähnt könnte die objektorientierte Programmierung dafür eine große Hilfe sein. Wartungsarbeiten können in einem solchen System nicht zentral erfolgen. Das bedeutet, daß die Verwaltung autonom an verschiedenen lokalen Teilsystemen erfolgen muß.

Viel komplexer wird das Problem der Benennung von Objekten in einem großen System. Es ist unmöglich alle Objekte zentral zu verwalten. Das geht schon aus Fehlertoleranzgründen nicht. Aber es ist auch sehr schwierig verteilte Nameserver dafür einzurichten, wenn Objekte migrieren können, d.h. von einem Rechner zum anderen wandern.

### 1.4 PM

Wir können die Einführung in die Idee von PM damit schließen, daß PM ein großes objektorientiertes verteiltes System für unterschiedlichste allgemeine Anwendungen sein soll. Das System soll leicht an spezielle Aufgabenklassen adaptierbar bleiben.

Das PM Projekt besteht aus zwei Teilen. Der eine Teil stellt das verteilte objektorientierte Programmiermodell dar. In dem Programmiermodell können Anwendungen beschrieben werden, die auf ortstransparente Kommunikation und ähnli-

che Abstraktionen aufbauen. In diesem Teil werden verteilte Anwendungen und verteilte Bestandteile des Betriebssystems realisiert. Zu letzteren könnte z.B. Benutzerverwaltung oder Druckerspools zählen.

Der zweite Teil stellt das eigentliche Betriebs- oder Laufzeitsystem für den ersten Teil dar. Dieses System hat die Aufgabe, die Abstraktionen des ersten Teils zu realisieren und die dort formulierten Anwendungen zur Ausführung zu bringen.

## 2. Erster Schritt

Zunächst wurden einige Überlegungen zum objektorientierten Programmieren und dessen Kombination mit verteiltem Programmieren angestellt. Das resultierende Objektmodell wurde in einer prototypischen Implementierung getestet.

Die Objekte einer verteilten Anwendung konnten in einer C++ ähnlichen Sprache formuliert werden. Die Objekte wurden in verschiedene UNIX Prozesse gelegt, die dann über ein Netzwerk von Workstations verteilt werden konnten.

### 2.1 Objektmodell

#### 2.1.1 Objekte

Das Objektmodell des Prototyps versucht zunächst eine Synthese bzw. Erweiterung der Ansätze von Argus [Liskov85] und Emerald [Hutchinson87a]: die Trennung in *lokale* und *globale* Objekte [Schlenk89a]. *Lokale* Objekte sind *privat* und *abhängig verteilt*, *globale* sind *sharable* und *unabhängig verteilt*. Ein *lokales* Objekt hat nur genau ein Objekt, von dem es aufgerufen werden kann. Diese Eigenschaft bleibt über die Lebensdauer des Objekts erhalten. *Globale* Objekte dagegen können von mehr als einem anderen Objekt aufgerufen werden.

Im Unterschied zu Argus sollte das Attribut *lokal/global* (des Typs) einer Instanzvariable frei wählbar bzw. konfigurierbar sein. In Argus wird diese Festlegung untrennbar an den Typ geknüpft, die Parameterübergabe von Objekten mit *lokalen* Typen ist nur *by-value*, mit *globalen* Typen *by-reference* gestattet. Die Überprüfung erfolgt statisch. Es kann also keine *lokalen* und *globalen* Objekte gleichen Typs geben.

Emerald schlägt den umgekehrten Weg ein: eine Datenflußanalyse des Compilers versucht die *Lokalität*, genauer *Privatheit*, von Typen zu einem Typ abzuleiten, alle übrigen Typen gelten als *global*.

Im Gegensatz zu Emerald wird in PM die *lokal-* bzw. *global-*Eigenschaft nicht vom Compiler erkannt, sondern vom Programmierer angegeben. Dabei kann ein Objekttyp lokale und globale Instanzen besitzen, wie in Emerald und im Gegensatz zu Argus.

Lokale Objekte werden in der Implementierung mit einer *call-by-copy* Semantik als Parameter übergeben, globale Objekte dagegen mit *call-by-reference*. *Call-by-*

*copy* kopiert übergibt eine Kopie des Objekts und entspricht damit *call-by-value* in anderen Sprachen. Damit die Übergabe der Parameter transparent für den Anwender bleibt, muß der Programmierer zusichern, daß ein lokales Objekt nach der Übergabe nicht mehr angesprochen (*in*-Parameter) oder beim aufgerufenen Objekt nicht zurückbehalten wird (*out*-Parameter)<sup>2</sup>.

In welcher Art ein Parameter übergeben wird, hängt vom konkret übergebenen Objekt ab und wird also zur Laufzeit entschieden, soweit sich dies nicht durch statische Prüfungen ermitteln läßt.

Lokale Objekte werden immer zusammen mit dem einzigen Objekt, dem sie bekannt sind, in einer Verteilungseinheit abgelegt. Durch die Parameterübergabe verändert sich die Zuordnung eines lokalen Objekts zu seinem möglichen Aufrufer. Dadurch wird das lokale Objekt verlagert. Die Parameterübergabe ähnelt den Emerald Semantiken *call-by-move* (*in* oder *out*) oder *call-by-visit* (*inout*).

Eine Verlagerung für globale Objekte kann mit dem Schlüsselwort *move* angestoßen werden. Es dient aber wie in Emerald nur als Hinweis an die Implementierung. Bei lokalen Objekten bleibt das Schlüsselwort unausgewertet, da dort immer eine Verlagerung stattfindet.

In einer Verteilungseinheit befindet sich immer nur ein globales Objekt und dessen lokale Objekte. Da diese Anordnung zu viele Verteilungseinheiten, sprich UNIX Prozesse, erzeugt hätte, wurde im Prototyp zugelassen, mehrere globale Objekte unwiderruflich in eine Verteilungseinheit zu legen.

### 2.1.2 Interaktion und Koordinierung

Die Interaktion zwischen Objekten erfolgt als synchroner Aufruf. Dabei wird im aufgerufenen Objekt dynamisch ein Aktivitätsträger gestartet. Eine dynamische Bindung des Aktivitätsträger und eine synchrone Kommunikation wird als besonders einfach zu implementierende und als besonders leicht verständlich für den Programmierer angesehen [Schlenk89c].

Konzeptionell bedeutet der synchrone Aufruf ein Wandern des Aktivitätsträgers durch die Objekte. Realisiert wird dies genau so im lokalen Fall, d.h. beim Aufruf eines lokalen Objekts. Dieses befindet sich in der gleichen Verteilungseinheit, also im gleichen Adreßraum. Dabei findet ein gewöhnlicher Prozeduraufruf statt.

Im globalen Fall, d.h. beim Aufruf eines globalen Objekts, wird ein RPC verwendet. Es wird wirklich ein neuer Aktivitätsträger gestartet und der aufrufende Aktivitätsträger bleibt blockiert, bis das Ergebnis des Aufrufs vorliegt. Als RPC Semantik wird *at-most-once* verwendet [Monge89b].

In einem Objekt können somit mehrere Aktivitätsträger gleichzeitig aktiv sein, was eine entsprechende Koordinierung<sup>3</sup> nötig macht. Da Aktivitätsträger sich durch die Kommunikation konzeptionell nicht vermehren können, gibt es initiale

---

<sup>2</sup> Eine Kombination von *in*- und *out*-Parametern (*inout*) muß dann entsprechend beide Bedingungen einhalten.

Aktivitätsträger, die mit der Erzeugung eines Objekts als zyklischer Prozeß entstehen. Weiterhin ist es möglich dynamisch Aktivitätsträger zu erzeugen und auf deren Beendigung zu warten (*fork-join*).

Ein Objekt stellt damit eine Prozeßabstraktion zur Verfügung. Durch das Beschreiben der Koordinierung bei der Objektbeschreibung ist zusätzlich eine Asynchronitätsabstraktion gegeben.

Die Koordinierung selbst sollte ursprünglich durch Nichtblockierungsbedingungen über den Zustand des Objekts erfolgen. Dazu wurden die in [Mackert83] beschriebenen Koordinierungsmethode vorgesehen. Die Methode wurde dann verändert und erweitert [Hofmann90]. Die Arbeit gibt außerdem Hinweise für eine möglichst parallele Implementierung.

Diese Methode ist auf sehr hohem Abstraktionsniveau angesiedelt, so daß man sich Optimierungsmöglichkeiten durch den Compiler und eine verbesserte Verifikationsmöglichkeit erhofft.

### 2.1.3 Konfiguration

Für die Konstruktion von komplexen Systemen aus einfachen Objekten ist es aus Gründen der Wiederverwendbarkeit und Flexibilität und auch aus Gründen des Software Engineerings von Vorteil die Programmierung kleinerer Objekte und deren Komposition zu großen Systemen zu trennen [DeRemer76]. In verschiedenen Projekten wurde daher eine eigene Konfigurationssprache entwickelt, z.B. CONIC [Kramer85].

Aufgabe der Konfiguration ist es die logische Struktur der Objektinstanzen zu beschreiben. Dabei gibt eine Programmiersprache, in der die Objekte beschrieben werden, vor, welche Objektinstanzen konfigurierbar sind.

In einer zweiten Phase wird die logische Struktur auf die physikalische Struktur des verteilten Systems abgebildet.

Diese statische Konfiguration wird ergänzt durch Mechanismen zur dynamischen Rekonfiguration.

## 2.2 Implementierung

Die Implementierung wurde in mehrere Schichten aufgeteilt:

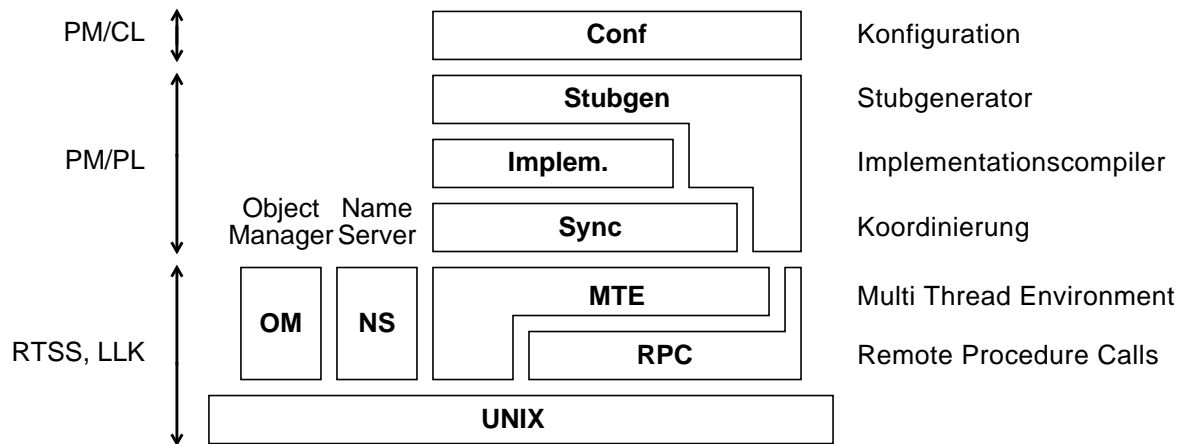
PM/LLK (Low Level Kernel) Ein Kern, der Basisabstraktionen bereitstellt, prinzipiell aber auch für andere Programmiermodelle außer PM geeignet sein soll.

---

<sup>3</sup> Wir sprechen hier von Koordinierung im Gegensatz zum oft verwendeten Begriff der Synchronisation. Unter Synchronisation wollen wir die Gleichzeitigkeit von Ereignissen verstehen, während es bei Koordinierung vielmehr um die Ungleichzeitigkeit von Ereignissen geht.

PM/RTSS	(Runtime Support System) Laufzeitsystem, das die Verbindung zwischen dem PM Programmiermodell und dem Kern herstellt.
PM/PL	(PM Programming Language) Programmiersprache zur Beschreibung von Objektklassen
PM/CL	(PM Configuration Language) Konfigurationssprache zur Beschreibung der Konfiguration

Die Plattform für die Implementierung bildet das UNIX bzw. SunOS Betriebssystem der verwendeten Workstations.



Die Objekte werden in UNIX Prozesse abgelegt, wobei ein UNIX Prozeß eine Verteilungseinheit darstellt. Zur besseren Gruppierung können mehrere globale Objekte in eine Verteilungseinheit zusammengefaßt werden.

In einem solchen UNIX Prozeß gibt es ein Laufzeitssystem, das die PM Objekte unterstützt. Dieses Laufzeitssystem ist in C++ geschrieben. Es enthält Objekte, die die ortstransparente Kommunikation zwischen den Verteilungseinheiten mittels RPC realisieren (RPC Schicht).

Die MTE Schicht (Multi Thread Environment) erlaubt es mehrere unabhängige Ausführungspfade innerhalb eines UNIX Prozesses zu verwenden (sogenannte Light Weighted Processes, LWP oder auch Threads). Dies ist nötig, weil in einem PM Objekt eine beliebige Anzahl von Aktivitätsträgern gleichzeitig agieren kann. Implementierungen von MTE werden in [Gorr89], [Götz91], [Prisille91] und [Helm91] beschrieben.

MTE und RPC bilden zusammen mit UNIX das, was man in der Schichtung LLK und RTSS genannt hat. Eine genaue Trennung ist nicht vorgenommen worden, da man hier nicht direkt auf die Hardware eines Systems aufsetzen mußte.

Die Schichtung des Sprachanteils PM/PL teilt sich in einen Stubgenerator und zwei Phasen eines Compilers. Der Stubgenerator erzeugt aus der C++ ähnlichen Sprache PM/PL eine Menge von C++ Objekten, von denen einige Stubfunktion haben. Diese werden als Stellvertreterobjekte in UNIX Prozessen instantiiert [Hauck89]. Sie konvertieren die Parameter in eine maschinenunabhängige Darstellung, XDR [XDR87].

Die Stubs sind eng verbunden mit den dazugehörigen RPC Objekten. Diese enthalten die physikalische Adresse des anzusprechenden Objekts. Die Adresse wird von einem Nameserverprozeß ausgegeben, der alle im System befindlichen Objekte anhand eines symbolischen Namens kennt [Monge89a]. Der Nameserver ist einmal im System vorhanden. Es wurden auch Versuche gemacht, diesen repliziert auszuführen [Schröer89]. Eine Integration in den Prototypen wurde jedoch nicht vorgenommen.

Ein Objektmanagerprozeß auf jedem beteiligten Rechner sorgt für die Instantiierung der Objekt-Prozesse. Er kann mit einem Kontrollprogramm von einer beliebigen Workstation aus ferngesteuert werden.

Zwei Phasen eines PM/PL Compilers sind im Schema eingezeichnet. Die eine Phase übersetzt die leicht abgewandelte Syntax von PM/PL in C++ Syntax. Die andere Phase wertet die Nichtblockierungsbedingungen der Koordinierung aus und wandelt diese in Koordinierungscode um. Diese beiden Phasen wurden nicht implementiert. Im Prototyp wurde die Syntax per Hand angepaßt und die Koordinierung durch direkte Kontrolle über die MTE-Schicht realisiert.

Es wurden aber Arbeiten durchgeführt, die sich mit der Realisierung der Nichtblockierungsbedingungen in C++ Code beschäftigten [Pruy89], [Küpfer90].

Die Konfigurationssprache PM/CL erlaubt Anwendungen aus den Klassen zusammenzustellen, die in PM/PL geschrieben sind. In PM/PL werden die konfigurierbaren Objektverbindungen (*knows*-Beziehungen) festgelegt. Diese Verbindungen müssen in der Konfigurationsphase mit konkreten Objekten versehen werden. Sie können entweder lokal oder global sein, im gleichen oder in einem anderen UNIX Prozeß liegen (letzteres nur wenn global). Die Konfiguration nimmt damit auch die Aufteilung der Objekte auf die Verteilungseinheiten vor. Die Verteilung dieser Einheiten auf bestimmte Maschinen kann nur rudimentär festgelegt werden. Sie erfolgt letztlich über das Instantiieren durch die Objektmanager.

In der prototypischen Implementierung erzeugt ein PM/CL Compiler eine Menge von C++ Dateien und ein Makefile, das die konfigurierten Prozesse zusammenbindet [Aufderheide90]. Die Prozesse müssen mit einem Kontrollprogramm über die Objektmanagerprozesse instantiiert werden.

## 2.3 Erfahrungen und Erkenntnisse

Der Zustand eines Objekts (im Sinne des Clonens eines Objekts) in PM/PL ist die transitive Hülle der *lokalen* Instanzvariablen. *Lokal* definiert also *part-of* (ist Teil von), *global* definiert *use* (benutzt). Die Festlegung der *lokal/global* Eigenschaften ist damit trotz der angegebenen Einschränkungen auf der Parameterübergabe nicht semantisch transparent für das Anwendungssystem: Der Umfang des Zustandes und damit die Kopie eines Objekts werden verändert. Die Zulässigkeit der Konfiguration muß vom Programmierer garantiert werden. Konfigurationsbeschreibung und Programmiersprache erlauben keine geeigneten Überprüfungen der Konfiguration durch den Compiler.

Instanzvariablen werden eingeteilt in konfigurierbare und nicht-konfigurierbare. Nicht-konfigurierbare Objekte sind *lokal* und werden implizit instantiiert und gebunden, konfigurierbare *lokale* und *globale* Objekte müssen explizit instantiiert und gebunden werden, beliebiges Sharen von Instanzen ist damit, wenn auch aufwendig, beschreibbar.

Die Implementierung unterstützt keinerlei dynamische Rekonfiguration, abgesehen von der Verlagerung *lokaler* Objekte durch die Übergabe als Parameter.

Das Objektmodell unterstützt keine Vererbung und keine wirkliche Schnittstelle, d.h. es kann ein Objekt nicht gegen ein anderes ausgetauscht werden, da ein neues Objekt nicht die gleiche Schnittstelle anbieten kann. Dies liegt daran, daß in C++ zwei Klassen nicht den gleichen Typ haben können.

Aus diesen Erkenntnissen wurde folgendes Fazit gezogen:

- Trennung der orthogonalen Eigenschaften:  
privat / sharable, abhängig / unabhängig verteilt, part-of / use
- Vereinheitlichung der Parameterübergabesemantik
- Einführung von Typen als Schnittstelle

### 3. Die nächste Schritte

Das objektorientierte Programmiermodell für das verteilte System bedarf nach den Erfahrungen aus dem Prototypen einer Verbesserung und Erweiterung. Es wird abkürzend PM/OM genannt (PM Objektmodell).

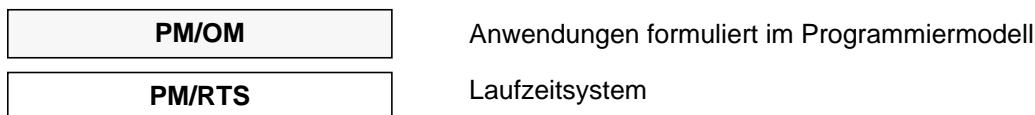
Für die Realisierung der Abstraktionen von PM/OM ist ein Laufzeitsystem erforderlich, künftig PM/RTS genannt (PM Runtime System). Der Begriff LLK wird fallengelassen. Zum einen erscheint es wenig erfolgversprechend einen Minimal-kern zu bauen, der für verschiedene Programmiermodelle geeignet ist, da sich die Anforderungen des Programmiermodells bis auf die grundlegenden Abstraktionen ausdehnen. Zum anderen wurden grundlegend neue Strukturierungsvorstellungen für das Laufzeitsystem entwickelt.

Das Programmiermodell PM/OM soll zur Entwicklung des Laufzeitsystems herangezogen werden. Es wird keine eigene Implementierungssprache benötigt, man verwendet bekannte Mechanismen und hat gleichzeitig eine Anwendung für bestimmte Mechanismen im Programmiermodell. Die Verwendung von PM/OM im Laufzeitsystem bedeutet aber auch, daß nicht in allen Bereichen mit allen Abstraktionen von PM/OM gearbeitet werden kann, da diese dadurch ja erst implementiert werden sollen. So muß man z.B. ab einer bestimmten Ebene auf ortstransparente Kommunikation verzichten, wenn man diese implementieren möchte.

Das Laufzeitsystem selbst besteht also aus Objekten, die neben den Objekten der verteilten Anwendungen im System existieren. Die Laufzeitsystem-Objekte werden Metaobjekte genannt, da sie zur Realisierung der Anwendungsobjekte dienen.

### 3.1 Das Objektmodell

Das Objekt- und Programmiermodell PM wird vom Laufzeitsystem realisiert. Es dient als Grundlage für die Programmierung verteilter Applikationen. Diese An-



wendungen kann man sich in einer eigenen Implementierungsschicht über dem Laufzeitsystem vorstellen.

Die folgenden Unterkapitel beleuchten Aspekte des überarbeiteten Objektmodells.

#### 3.1.1 Das Referenzenmodell

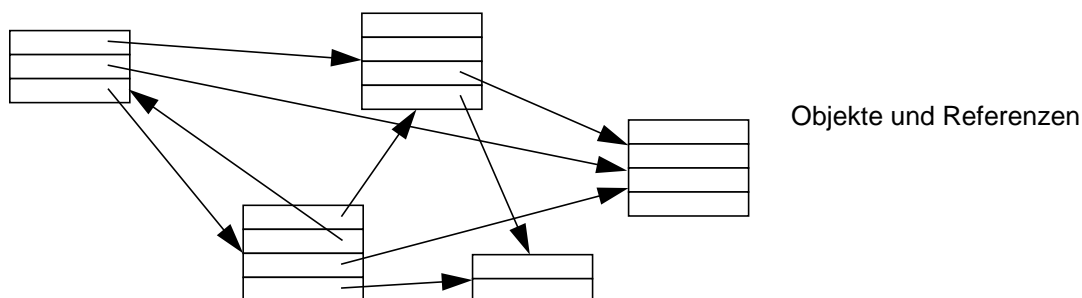
Ein Objekt im PM/OM hat folgende Eigenschaften. Sie wurden teilweise aus [Sakkinen89] übernommen:

- Identität - Sie ist eindeutig im Objektsystem.
- Integrität - Sie erhält die Kapselung des Objekts nach außen.
- Erzeugbarkeit, Zerstörbarkeit
- Typ
- Attribute

Unter Attributen versteht man Instanzvariablen, Operationen und Prozesse (Aktivitätsträger). Das Referenzenmodell stellt eine genauere Definition der Instanzvariablen dar.

Alle Instanzvariablen im Modell sind Objekte. Es gibt also keine primitiven Datentypen wie in C++, die im Modell teilweise einer anderen Semantik unterliegen, z.B. bei der Parameterübergabe.

Ein Objekt hat auf alle Objekte (Instanzvariablen), die es ansprechen kann, Verweise, im folgenden Referenzen genannt. Diese Referenzen verweisen auf das entsprechende Objekt. Sie können z.B. durch die eindeutige Identität repräsentiert werden. Der *direkte Zustand* eines Objekt sind die Referenzen, die es besitzt. Der



direkte Zustand ist nicht verteilbar und bildet die kleinste Verteilungseinheit. Instanzvariablen dagegen können verteilt sein (*indirekter Zustand*).

Über eine Referenz sind die Operationen eines Objekts aufrufbar. Die Referenz ist getypt, d.h. sie bestimmt welche Operationen aufrufbar sind. Dieser Typ muß mit dem Typ des Objekts verträglich sein.

Zur Initialisierung des Systems gibt es eine Reihe von Objekten, die im System von vorneherein bereitgestellt werden, sich selbst aber von anderen Objekten nur durch Verhalten und Identität unterscheiden. Solche Objekte könnten minimal die "Bits" 0 und 1 sein. Praktischerweise wird man Objekte ähnlich denen in anderen Programmiersprachen anbieten: Integer, Real, Character, usw.

- In welcher Form diese Objekte von vorneherein im System vorhanden sind ist zu untersuchen. Stichworte sind hier Werte und Objekte, sowie unveränderliche Objekte (*immutable objects*).

### 3.1.2 Typ, Klasse und Vererbung

Das Objektmodell ist klassenbasiert, d.h. die Beschreibung von Objekten erfolgt in Klassen. Die Klasse dient als Vorlage für die Instantiierung von Objekten. Die von einer Klasse erzeugten Objekte sind in ihrer Struktur identisch. Sie haben potentiell das gleiche Verhaltensmuster und unterscheiden sich nur durch ihre eigene Identität und eventuell durch die Identität der Objekte, auf die sie Referenzen besitzen. In anderen Systemen werden diese Referenzen auch Zustand genannt.

Ein Objekt kann von außen nur über den Aufruf einer Operation beeinflußt werden. Dies stellt die Integrität des Objekts sicher. Die Operationen werden auch Methoden genannt. Die Signatur und das sichtbare Verhalten der von außen erreichbaren Methoden stellt den Typ eines Objekts dar.

In vielen Sprachen wird der Typ einer Klasse und deren Objekte direkt abgeleitet, so daß jede Klasse ihren eigenen Typ besitzt. Typkonformität, Polymorphismen und Austauschbarkeit von Objekten kommen nur durch eine sogenannte Vererbungsbeziehung zwischen Klassen zustande, z.B. C++ [Ellis90] und Eiffel [Meyer88]. Wir fordern eine Trennung der Begriffe Typ und Klasse.

Typen sind eigenständige Einheiten, die Signatur und sichtbares abstraktes Verhalten von Klassen beschreiben. Letzteres ist zur Zeit nur informell beschreibbar durch Kommentare oder durch geschickte Namenswahl der Methoden. Es gibt allerdings Ansätze das Verhalten konkreter zu fassen, z.B. durch *Properties* in POOL-I [America90].

Klassen sind reine Implementierungseinheiten, die konkrete Struktur und konkretes Verhalten beschreiben. Sie müssen mit einem Typ assoziiert werden, indem Signaturen aus dem Typ mit denen der Klasse verbunden werden. Alle Objekte der Klasse besitzen dann diesen Typ. Dieses Verfahren erlaubt mehreren Klassen gemeinsame Typen zu implementieren und andererseits einer Klasse mehrere verschiedene Typen anzubieten.

Vererbung im Sinne der objektorientierten Programmierung muß getrennt für Typen und Klassen definiert werden. Vererbung zwischen Typen kann Konformität ausdrücken. Dies muß nicht eine Konformität in Richtung Spezialisierung alleine sein (entsteht durch Hinzufügen von Methoden). Vielmehr kann durch Streichung von Methoden eine Konformität in die andere Richtung (Generalisierung) erreicht werden. Möglich erscheint auch die Vererbung ohne Konformität.

Es scheint weiterhin nützlich zu sein, Konformität zweier Typen ohne Vererbungsbeziehung ausdrücken zu können.

Überlegungen zur Vererbung von Klassen wurden in [Eirich91] angestellt. Die Idee ist dabei eine Schnittstelle zur Vererbung zu fordern, da ohne Schnittstelle der Austausch von Klassen in der Vererbungshierarchie unmöglich ist ohne die Kapselung der Objekte zu verletzen [Snyder87]. Der verfolgte Ansatz ist dabei eine Art statische Delegation, die entsprechend syntaktisch unterstützt wird. Der Typ einer Klasse kann dann als Schnittstelle zur Vererbung betrachtet werden. In Vererbung stehende Klassen oder deren Objekte sind mit einer gewöhnlichen Referenz verbunden. Die Vererbung benötigt daher keine spezielle Behandlung im Referenzenmodell.

- Es ist zu untersuchen wie handhabbar die Trennung von Typ und Klasse bei objektorientierter Programmierung ist. Speziell bei der Konfiguration scheint die Möglichkeit an jeder Stelle hinter einem Typ zwischen mehreren Implementierungen wählen zu können eine solche Fülle von Einflußmöglichkeiten zu eröffnen, daß die Handhabbarkeit in Frage stehen könnte, wenn es nicht gelingt entsprechende Vereinfachungen und Defaulteinstellungen zu definieren.
- Die Untersuchung des Vererbungsansatzes durch statische Delegation ist noch nicht abgeschlossen. Hier ist ebenfalls die praktische Anwendung zu überprüfen.

### 3.1.3 Konfiguration

Ein zentrales Anliegen objektorientierter Ansätze ist die Wiederverwendbarkeit der Resultate des Software-Designprozesses einer Anwendung. Auf der anderen Seite erfordert ein großes verteiltes Anwendungssystem, die Möglichkeit auf die Verteilung Einfluß zu nehmen. Dabei sind die komplexen globalen Abhängigkeiten und die dynamischen Abläufe im System zu berücksichtigen. Die Beschreibung der Verteilung kann in der Regel nicht lokal zu einem Objekt unabhängig von dem Kontext, in dem es benutzt wird, erfolgen. Dynamische Abläufe, die Änderungen an der Last, die ein Objekt erzeugt, oder Änderungen an den Beziehungen von Objekten, die die Interaktion beeinflussen, hervorrufen, müssen faßbar gemacht werden.

Der Ansatz einer Beschreibung der Verteilung in einer entsprechend erweiterten Programmiersprache hat zur Folge, daß

- die objekt-lokale Sicht und damit die strikte Kapselung aufgegeben werden muß, um auch nicht objekt-lokale Zusammenhänge beschreibbar zu machen.

- die Wiederverwendbarkeit drastisch eingeschränkt wird oder über weitreichende Parametrierungen der Verteilung für alle Benutzungssituationen erkauft wird. Neue unvorhergesehene Anforderungen können nicht berücksichtigt werden oder führen dann zu einer Erweiterung oder sogar Neu-Programmierung.

Zwei zentrale Eigenschaften der objektorientierten Programmierung wären damit kompromittiert.

Sinnvoller erscheint der Ansatz, die Beschreibung der Verteilung getrennt von der Programmierung auf den Instanzen in ihrem konkreten Verwendungskontext vorzunehmen. Wir bezeichnen das als Verteilungskonfiguration. Die Verteilungskonfiguration beschreibt ein *Verteilungssystem* über dem Anwendungssystem: der Zustand des Verteilungssystems beschreibt die erlaubten Orte der Objekte des Anwendungssystems, das Verhalten Änderungen des Zustands als Reaktion auf externe Ereignisse. Die initiale Verteilung (initialer Zustand) wird als statische, die Anpassung der Verteilung als Reaktion auf Ereignisse (Verhalten) als dynamische Konfiguration bezeichnet.

Ereignisse, die bei der dynamischen Konfiguration nicht berücksichtigt wurden, können einen Eingriff von außen erfordern: eine Rekonfiguration. Bei der langen, ununterbrochenen Laufzeit verteilter Anwendungen, ermöglicht durch geeignete Fehlertoleranzmechanismen, verbietet sich eine Unterbrechung: Die Rekonfiguration muß dynamisch, zur Laufzeit, erfolgen.

Nicht nur die dynamischen Abläufe des Anwendungssystems können eine Rekonfiguration erfordern, auch die verteilte Hardware durch Ausfall, Austausch und Erweiterung sowie der Benutzer durch geänderte Anforderungen an das Anwendungssystem. Die Rekonfiguration kann dabei nicht auf das Verteilungssystem beschränkt bleiben, sondern muß auch Änderungen am Anwendungssystem erlauben: Entfernen, Austauschen und Hinzufügen von Komponenten. Diese Art von Rekonfiguration ist zwangsläufig nicht vollständig transparent für das Anwendungssystem, die angebotenen Mechanismen müssen auf die Sprache abgestimmt und mit dem Ablauf koordiniert werden.

Ziel der Konfiguration ist die Beschreibung der Struktur und der Änderungen an der Struktur sowie die Beschreibung der Verteilung als auch der Änderungen an der Verteilung. Struktur- wie Verteilungskonfiguration definieren Objekteigenschaften und -beziehungen. Die Abbildung der Strukturbeschreibung erfolgt durch den Compiler auf die entsprechenden Abstraktionen des Laufzeitsystem. Die Verteilungsbeschreibung wird durch das Konfigurationssystem und die sogenannte Ablaufsteuerung auf das Laufzeitsystem abgebildet. Die Ablaufsteuerung stellt eine Art Laufzeitsystem für das Verteilungssystem dar. Ihre Aufgabe ist es also auf externe Ereignisse nach dem Verhalten des Verteilungssystem zu reagieren und den Zustand des Verteilungssystems und damit die Verteilung des Anwendersystems zu verändern.

Beschreibt das "Programm" die Struktur und die vorgesehenen Änderungen der Struktur, so beschreibt die Verteilungskonfiguration die Verteilung und die vorgesehenen Änderungen an der Verteilung. Nicht vorgesehene Änderungen an

Struktur oder Verteilung bzw. deren "Transformationsvorschriften" werden durch die Rekonfiguration vorgenommen.

"Programm" und "Verteilungskonfiguration" lassen sich damit beide als Paar von Zustand und Verhalten beschreiben, die Rekonfiguration als Änderung an einem der Paare.

Generelles Ziel der Konfiguration ist die Anpaßbarkeit, d.h. Anpassung ohne Beschränkung der Wiederverwendbarkeit. Die Konfiguration stellt dazu "Programmier- bzw. Berechnungsmodelle" zur Beschreibung von weiteren orthogonalen Eigenschaften des Anwendungssystems.

Die Verweise auf Objekte bilden die Grundlage der Verteilungskonfiguration. Deren initialer Zustand und die mögliche dynamische Veränderungen müssen adäquat durch Mechanismen der Programmiersprache erfaßbar sein:

- Beschreibung von Lokalität und Sharing

Lokalität bzw. Sharing kann explizit durch entsprechende Deklaration von Typen und Instanzvariablen erfolgen. Die deklarierte Eigenschaft kann vom Compiler und Binder überprüft werden.

Durch Einschränkung der Parameterübergabesemantiken kann die Privatheit implizit sichergestellt werden, z.B. *call-by-copy*, *call-by-temporary-reference* (nur während des Aufrufs bleibt Referenz gültig).

- Parameterübergabesemantik

Die allgemeine Übergabesemantik ist *call-by-object-reference*. Sie wird durch entsprechende Einschränkungen verändert, um Lokalität und Optimierungen in der Implementierungen zu ermöglichen.

- Zustand von Objekten

Der Zustand wird als Teilmenge der Instanzvariablen definiert. Sie stehen in einer Objektbeziehung *part-of*, alle anderen Instanzvariablen werden benutzt aber sind nicht Teil des Zustands (*use*-Beziehung).

Ein Vorschlag für eine Veränderung von C++ zu einer Sprache, die die beschriebenen Mechanismen besitzt, und für eine Beschreibung des Verteilungssystems wird in [Fäustle92] gemacht werden.

- Der Ansatz ein zweites System mit Zustand und Verhalten neben das Anwendersystem zu stellen, um die Verteilung zu beschreiben, könnte eventuell auf andere Eigenschaften des Programmiermodells angewandt werden, z.B. für die Koordinierung. Dafür müssen jedoch zunächst geeignete Grundlagen in der Programmierung geschaffen werden.

Neben dem durch die Konfiguration verfolgten Ansatz, gilt es weitere zu finden. Programmiermodell und Architektur sollten daraufhin weiterentwickelt werden.

- In welcher Art das Verteilungssystem auf externe Ereignisse reagieren soll und kann, sowie die Strukturierung der Ablaufsteuerung stellen künftige Untersuchungsschwerpunkte dar.

### 3.1.4 Interaktion

Das bisherige Interaktionsschema war synchron und verteilt als RPC ausgeführt. Es handelt sich dabei um eine Request-Reply Kommunikation. An einer solchen Kommunikationsart soll festgehalten werden. Als Alternative käme ein One-Way-Message-Passing in Frage, das aber ohne Aufwand<sup>4</sup> nur die Übergabe monotoner Prädikate erlaubt, da keine Prädikate invalidiert werden können. Als Spezialfall kann der Verzicht auf eine Antwort gehandhabt werden, so daß die einfachere Implementierung des One-Way-Message-Passing durch eine Optimierung von Request-Reply Kommunikation erreicht werden kann, wenn festgestellt wird, daß die Antwort für den Sender uninteressant sein wird.

Aktivitätsträger konnten bisher dynamisch mit einer *fork-join* Semantik erzeugt werden. Diese *fork-join* Semantik stellt nichts anderes als eine primitive asynchrone Interaktion dar. Bei einer asynchronen Interaktion kann gezielt eine Operation angegeben werden, die auszuführen ist, während man bei *fork-join* den aktuellen Aktivitätsträger teilt und dessen lokalen Zustand verdoppeln muß, obwohl dies in den meisten Fällen gar nicht nötig wäre. Es wird daher zusätzlich zum synchronen Interaktionsschema ein asynchrones Schema vorgeschlagen und *fork-join* als dynamische Aktivitätsträgererzeugung verworfen. Die Kommunikation besteht dann aus zwei Teilen:

- *SendRequest* - mit Übergabe von Parametern
- *GetReply* - mit Übernahme der Ergebnisse

Zwischen der Ausführung beider Teile besteht abstrakt echte Parallelität. *GetReply* wartet auf die Ergebnisse. *SendRequest* direkt gefolgt von *GetReply* stellt einen synchronen Aufruf dar.

Die beiden Kommunikationsprimitive bieten eine gute Basis für Gruppenkommunikation. So kann bei *SendRequest* eine Menge von Empfängern angegeben werden. Bei *GetReply* kann eine Teilmenge der möglichen Antworten als Ergebnis übernommen werden.

Einkommende Aufträge (*Requests*) erzeugen immer eigene Aktivitätsträger, die beim Senden der Antwort zerstört werden, d.h. im Objektmodell gibt es nur ein konzeptionelles Wandern von Aktivitätsträgern im synchronen Fall der Interaktion.

Die Abbildung des Interaktionsmodells auf eine Implementierung kann ähnlich wie im bisherigen Modell erfolgen. Für den synchronen Fall gibt es zwei Implementierungsmöglichkeiten. Im Falle eines knotenlokalen Aufrufs wird der aufrufende Aktivitätsträger der Implementierung für die Bearbeitung des Auftrags wiederverwendet. Ein RPC Aufruf wird im verteilten Falle durchgeführt. Dabei wird ein neuer Aktivitätsträger gestartet.

---

<sup>4</sup> Es muß eine Antwort zurückgeschickt werden, um Prädikate invalidieren zu können. Der Aufwand entsteht in der nötigen Zuordnung von Antworten zu Aufträgen, die sonst das System übernimmt.

Bei asynchroner Kommunikation kann im lokalen Fall ein Compiler feststellen, daß die zu erzielende Parallelität zu gering ist und das *SendRequest* nur als Hinweis auffassen. *GetReply* wird dann wie ein synchroner Aufruf behandelt. Andernfalls wird ein neuer lokaler Aktivitätsträger gestartet, mit dem sich der Sender bei *GetReply* synchronisieren muß.

Im verteilten Falle muß beim Empfänger sowieso ein eigener Aktivitätsträger gestartet werden, so daß sich keine Änderung der Implementierung ergibt. Bei *SendRequest* wird eine entsprechende RPC Nachricht verschickt. Die Implementierung kann jetzt parallel die Nachrichten für eine RPC Protokoll abwickeln oder dies erst bei *GetReply* anstoßen. Einkommende Replynachrichten müssen intern gespeichert werden. Dazu stellt der Empfänger den Speicher bereit.

Die bekannten Techniken und Implementierungen für RPC Protokolle können also auch bei asynchroner Interaktion verwendet werden.

- In [Monge92] wird ein Überblick über verschiedene Kommunikationsformen in verteilten objektorientierten Systemen gegeben werden. Dabei wird untersucht wie Aktivitätsträger an der Objektinteraktion teilhaben können, z.B. Rendezvous, asynchrone und synchrone Interaktion etc. Einen anderen Schwerpunkt bildet die Semantik der Nachrichtenübertragung, z.B. *at-most-once*.

Interaktion und Semantiken werden auch unter dem Aspekt der Kommunikation zwischen Objektgruppen betrachtet (*Multicasts*). Gruppenkommunikation kann z.B. für softwaregesteuerte Fehlertoleranz eingesetzt werden, erfordert aber eine höhere Semantik für die Nachrichtenübertragung (*atomic multicasts*).

### 3.1.5 Koordinierung

Koordinierungsmaßnahmen sind Teil des abstrakten Verhaltens eines Objekts. Es wird daher gefordert diese in der Schnittstelle sichtbar zu machen. Dabei wird nach Verfahren gesucht, die einer Implementierung der Schnittstelle genügend Freiraum geben, einem Nutzer der Schnittstelle aber genügend Information über die Verwendbarkeit und das Verhalten des Objekts vermitteln.

Die Forderung nach Wiederverwendbarkeit scheint nur erfüllbar zu sein, wenn die Formulierung der Methoden und die Formulierung der in der Konsistenz und dem gewünschten Verhalten eines Objekts begründeten Koordinierungsbedingungen voneinander getrennt erfolgen.

In diesem Zusammenhang und zusammen mit der Einführung asynchroner Kommunikationsformen wurde quasiparallele Nebenläufigkeit innerhalb eines Objekts diskutiert. Die Dualität von Systemen mit synchronen Aufrufen und dynamischer Prozeßstruktur und Systemen mit asynchronen Aufrufen und statischer Prozeßstruktur wurde in [Liskov86] nahegelegt. Es ist daher möglich mit asynchronen Aufrufen quasiparallele Objekte zu formulieren ohne an Ausdrucksfähigkeit zu verlieren. Insbesondere die in [Liskov86] gezeigten Probleme des *local* und *remote delay* sind blockierungsfrei ausdrückbar.

Koordinierungszusammenhänge müssen aber nicht nur lokal für einzelne Objekte formuliert werden, sondern auch für Gruppen von Objekten. Dies ist zur Beschreibung der Koordinierung in verteilten Strukturen notwendig. Hier wird eine zentralisierte Formulierung der Zusammenhänge angestrebt. Einem Programmierer soll dabei das gleiche Denkmodell angeboten werden, das ihm von der Koordinierung einzelner Objekte her bekannt ist.

Die Mackert'schen Koordinierungsmechanismen [Mackert83] haben sich in diesem Kontext als nicht geeignet erwiesen. Die Betriebssystemmaschine, über die die Semantik der Koordinierungskonstrukte definiert ist, birgt einen hohen Kommunikationsaufwand zwischen den beteiligten Datenstrukturen. Zwar erlaubt die Betriebssystemmaschine hinreichend Nebenläufigkeit [Hofmann90], diese kommt aber nur dann zum Tragen, wenn die Kosten für die Kommunikation gering sind. Im Fall einer physikalischen Verteilung sind sicherlich höhere Kosten anzunehmen.

Die Forderung Kommunikation und Koordinierung getrennt zu betrachten scheint unmöglich, da eine Request-Reply Kommunikation immer eine implizite Synchronisation hervorruft.

Die begrenzte Länge von Auftragswarteschlangen, die bei asynchronen Systemen mit sequentieller oder quasiparalleler Abarbeitung zu führen sind, wurde bisher als Argument gegen solche Systeme verwandt. Bei einer dynamischen Prozeßzeugung ist die Anzahl der möglichen parallelen Aktivitätsträger jedoch ebenfalls begrenzt, so daß bei übermäßiger Kommunikation und schlechter Koordinierung die gleichen Anomalien auftreten können.

- Es ist ein Koordinierungsmechanismus zu finden, der Wiederverwendbarkeit und Kapselung von Objekten nicht entgegensteht. Dabei ist ein quasiparalleler Ansatz nicht von vorneherein auszuschließen.

Die Beschreibung der Koordinierung sollte möglichst deklarativ erfolgen, da dies eine abstraktere Beschreibung darstellt, die an sich erstrebenswert ist, andererseits mehr Möglichkeiten für Optimierungen bietet und außerdem eine größere Wiederverwendbarkeit verspricht.

- Die Koordination von Aufrufen eines koordinierten Objekts bei anderen Objekten können eventuell optimiert werden, so daß im Idealfall kein Koordinierungscode für das aufgerufene Objekt zu erzeugen ist, wenn sichergestellt werden kann, daß die Aufrufe nur in einer verträglichen Weise erfolgen werden. Es ist zu untersuchen, welche Bedingungen erfüllt sein müssen, damit in dieser Weise optimiert werden kann.
- Die Frage der Koordinierung zwischen mehreren Objekten, d.h. die Beschreibung von Protokollen, ist noch unerforschter, als die Koordinierung innerhalb eines Objekts. Sie gewinnt noch mehr Bedeutung, wenn statt Vererbung die statische Delegation als Kompositionsmittel eingesetzt wird [Eirich91]. Es ist zu erwarten, daß eine Lösung dieser Fragestellung ganz allgemein das Problem der Verträglichkeit von Nebenläufigkeit und Vererbung lösen würde.

- ❑ Das Denkmodell der Koordinierung, wie es einem Programmierer von MacKert angeboten, scheint in Hinblick auf Verständlichkeit und Expressivität zur Formulierung von Koordinierungszusammenhängen geeignet. Die operationell definierte Semantik jedoch führt zu Problemen. Es müssen alternative Definitionen der Semantik gefunden werden, die in Bezug auf Verteilung günstigere Eigenschaften aufweisen.

### 3.1.6 Andere Abstraktionen

Das abstrakte Objektmodell beachtet bisher noch keine höheren Abstraktionen:

- ❑ Es ist zu untersuchen, wie sich Persistenz und Fehlertoleranzeigenschaften in das Objektmodell integrieren lassen. Als fehlertolerante Mechanismen sind Replikationen und Transaktionen zu untersuchen. Interessant ist dabei der Zusammenhang zwischen Koordinierung und Transaktion, zwischen Replikation und Kommunikation.
- ❑ Das bisherige Modell betrachtet keinerlei Zugriffsberechtigungen. Ein Betriebssystem kommt ohne Rechte nicht aus. Verschiedene objektorientierte Systeme implementierten Zugriffsrechte über Capabilities, die im PM Objektmodell einer Objektreferenzen mit Zugriffsrechten gleichkommen. Es ist zu untersuchen, ob Zugriffsberechtigungen Eingang in das Objektmodell und dessen Fehlerbehandlung finden sollen.

## 3.2 Das PM Laufzeitsystem

Die Aufgaben des Laufzeitsystem können in mehrere Aufgabengebiete unterteilt werden, wie Prozeßverwaltung, Speicherverwaltung, Kommunikationsabwicklung etc. Die einzelnen Aufgaben stellen jeweils mehr oder weniger orthogonale Abstraktionen des PM Objektmodells dar. Ihre Realisierung kann in mehreren Schichten erfolgen, z.B. bei der Kommunikation von der ortstransparenten, über die Interadreibraum, zur Intraadreibraumkommunikation.

In allen Schichten sollen die Abstraktionen des PM Objektmodells mehr oder weniger zum tragen kommen. Dabei verspricht man sich durch die Strukturierung des PM/RTS einen Rückschluß auf das Programmiermodell durch Integration von Anforderungen aus der praktischen Anwendung.

### 3.2.1 PM/SA, Systemarchitektur

Die PM Systemarchitektur stellt Überlegungen dar, wie das Laufzeitsystem zu strukturieren ist, um das Objektmodell zu realisieren und gleichzeitig adaptierbar zu sein.

Grundlage dabei ist, daß das Laufzeitsystem im PM Objektmodell aufgebaut und strukturiert wird. Dabei können nicht alle Abstraktionen des Objektmodells an allen Stellen verwendet werden, da diese ja durch das Laufzeitsystem erst implementiert werden sollen.

Das Laufzeitsystem kann in seine Aufgabenklassen vertikal zerlegt werden. In den Aufgabenklassen kann man verschiedene Schichten erkennen (horizontal). So ist es denkbar, die ortstransparente Kommunikation mit Objekten zu implementieren, die mit einem Interadreibraumaufwurf interagieren. Der Interadreibraumaufwurf wird mit Objekten implementiert, die immer im gleichen Adreibraum liegen. So erhält man ein geschichtetes Laufzeitsystem, das von Schicht zu Schicht Abstraktionen des Objektmodells aufgibt, um diese zu implementieren. Zusätzlich kommen Objekte hinzu, die eine Abstraktion der Hardware darstellen und die nötig sind, um die Implementierung zu realisieren. In diesem Beispiel ist es nötig Objekte zu schaffen, die vom "Adreibraumwechsel durch das Nutzen einer MMU" abstrahieren.

Rechensysteme, die andere Rechensysteme implementieren, werden *Metasysteme* genannt [Maes88]. Das PM/RTS kann damit als Metasystem des verteilten PM Systems angesehen werden. Objektorientierte Metasysteme bestehen aus *Metaobjekten*. Die Schichtung eines Metasystems in Ebenen, die Abstraktionen auf der Basis niedrigerer Abstraktionen aufbauen, ist eine Metaschichtung. Metaebenen bauen auf Abstraktionen von Meta-Metaebenen auf oder anders ausgedrückt Meta-Metaebenen realisieren Metaebenen usw. Die Abstraktionen, die eine Metaebene realisiert, werden immer kleiner in Richtung Metaebene.

Die Schichtung stellt eine Rekursion dar, wenn eine Metaebene Abstraktionen benutzt, die sie selbst oder eine höhere Ebene bereitstellt. Die Rekursion wird nur aufgelöst, wenn die Metaebene sicherstellt, daß rekursive Aufrufe niedrigere Dienste realisieren und damit irgendwann die eigene Abstraktion nicht mehr benutzt wird.

Es ist schwierig, Metaebenen von Schichten zu unterscheiden, die durch normales Strukturieren von Software entstehen. Ob ein Objekt Metaobjekt ist, hängt vom Standpunkt des Betrachters ab. Allgemein kann man sagen, daß ein Objekt, das eine Abstraktion oder Teile einer Abstraktion des Objektmodells implementiert<sup>5</sup>, aus Sicht des Nutzers dieser Abstraktion als Metaobjekt bezeichnet wird.

Das PM System besteht nun aus einer Menge von Objekten, die in verschiedenen Metasystemen oder im Anwendersystem liegen. Diese enthält die Anwender- oder Betriebssystemobjekte, die im vollständigen und verteilten Objektmodell beschrieben wurden.

Da diesen Objekten zum Großteil gemeinsame Abstraktionen zugrunde liegen, ist es leicht realisierbar, daß sie miteinander kommunizieren. Kommunikation zwischen Anwender- und Metasystem ist z.B. nötig, um neue Objekte im Anwendersystem zu instantiiieren. Dabei bringt das Metasystem Dienste für das Anwendersystem.

Eine andere Art dieser Kommunikation ist Reflexion. Wenn z.B. ein Unixprozeß seine Priorität durch einen Kernaufwurf ändert, so beeinflußt er sein Verhalten auf

---

<sup>5</sup> Diese Definition schränkt den Begriff Metaobjekt auf das Objektmodell ein, da im RTS nur dessen Implementierung im Vordergrund steht. Metaobjekte können von der Idee her auch in Anwendungen eingesetzt werden, wo sie Basisabstraktionen einer Anwendung definieren. Ein gutes Beispiel ist das Silica Window System [Rao91].

eine Weise, die er in seinem Programmiermodell nicht sehen kann. So bleibt dem Prozeß sein Schedulingverhalten weitgehend verborgen.

Metainteraktion liegt vor, wenn das Anwendersystem mit dem Metasystem kommuniziert. Reflexion wird als Metainteraktion definiert, wobei die Wirkung eines Operationsaufrufs im Metasystem eine Verhaltensänderung des aufrufenden Objekts bewirkt [Maes87a]. Diese Verhaltensänderung betrifft Objekte des Anwendersystems und ist gerade nicht durch Operationsaufrufe in diesen Objekten zustande gekommen. Reflexion verletzt damit möglicherweise die Kapselung von Objekten. Reflexion stellt aber die einzige Möglichkeit dar, Verhalten zu beeinflussen, das im Objektmodell des Anwendersystems gar nicht definiert ist. So wird im Anwendermodell eventuell keine Aussage über eine Pagingstrategie gemacht. Mit Reflexion kann die Strategie beeinflußt werden.

- ❑ Es ist zu untersuchen, wie sich die Aufgaben des RTS vertikal zergliedern lassen und wie sich diese in Metaebenen schichten. Dabei ist besonders die Verflechtung der Ebenen interessant, d.h. wie orthogonal die Aufgaben letztlich realisierbar sind.
- ❑ Interesse gilt den direkten Abstraktionen der Hardware. Wie können bekannte und verbreitete Leistungsmerkmale der Hardware in einer Abstraktion vereinigt werden, die sich in einer Realisierung leicht portieren läßt, z.B. die Abstraktion einer MMU für den Adreßraum und den Adreßraumwechsel.
- ❑ Der Einsatz und Nutzen von Reflexion ist näher zu betrachten. Reflexion ist kein neuartiger Mechanismus, sondern automatisch vorhanden, sobald Anwender- und Metasystem miteinander kommunizieren können und das müssen sie in der Regel. Es geht somit nicht mehr um die Frage, ob man in solch einem System Reflexion haben möchte oder nicht, sondern nur darum, Reflexion mit all ihren Vor- und Nachteilen zu erkennen, nutzbringend einzusetzen und mißbräuchliche Benutzung zu vermeiden.

### 3.2.2 Schutzräume und PM/RTS

Betriebssysteme enthalten üblicherweise einen Kern, der in einer geschützten Umgebung ablaufen muß. Dabei ist die Art der Kommunikation sehr verschieden, je nachdem, ob Anwendungen mit dem Kern, der Kern mit der Anwendung oder zwei Anwendungen bzw. zwei Teilbereiche im Kern miteinander kommunizieren [Hofmann91]. Es besteht die Tendenz, Teile aus dem Kern des Betriebssystems auslagern zu wollen, um sie wie Anwendungsprogramme auszutauschen und zu programmieren.

Da PM aus einer Menge von Objekten und Metaobjekten besteht, die transparent kommunizieren können, ist nicht ersichtlich, warum das Laufzeitsystem in einen geschützten Bereich gelegt werden muß. Der Begriff des Kerns wird daher fallengelassen. Es stellt sich somit auch nicht die Frage, Teile aus dem Kern auslagern zu wollen.

Das Einziehen von Schutzgrenzen, wie sie Adreßräume bilden, kann im PM-System nach Aufgaben und Notwendigkeit stattfinden, da Objekte mit Metaobjekten sowohl über Adreßraumgrenzen als auch im gleichen Adreßraum transparent kommunizieren können<sup>6</sup>. Das Ziehen von Schutzraumgrenzen muß nicht horizontal erfolgen, sondern kann auch vertikal nach Aufgabenklassen erfolgen.

Schutzgrenzen sind auch durch das Programmiermodell selbst gegeben, da die Sprache eine Kapselverletzung eines Objekts verbietet. Bewegt man sich also nur innerhalb von PM/OM, so kann ein Compiler bereits gewissen Schutz garantieren, so daß nicht so viele teure Schutzräume nötig sind. Je nach Anwendungs- und Sicherheitsbedürfnissen kann dies individuell organisiert (konfiguriert) werden, was dem Ziel der Adaptierbarkeit des Systems zu Gute kommt.

- Inwieweit Adreßraumgrenzen orthogonal zu allen anderen Abstraktionen von PM/OM sind, muß untersucht werden. So ist sicherlich nicht jede Partition von Objekten des Laufzeitsystems eine gültige Aufteilung in Adreßräume.

### 3.2.3 Objekt- und Speicherverwaltung

Ein wesentlicher Bestandteil des PM Laufzeitsystems ist die Bereitstellung von Mechanismen zur Verwaltung der Objekte von Anwendungen in PM/OM und der Zuordnung von Speicherplatz (Haupt- und Hintergrundspeicher) zu diesen Objekten.

Für die Konzeption der Objekt- und Speicherverwaltung werden folgende Grundsätze festgelegt:

- Die Struktur von Objekt-Verwaltung und -Speicherung sollte sich aus der Definition des Objektmodells und der Anwendungen ergeben, sie sollte aber nicht von der Hard- und Firmware-Architektur diktiert sein, auf der das System implementiert wird.
- Die Umgebung, in der die Ausführung einer Anwendung erfolgt, ist dagegen zwangsläufig durch die Architektur der verwendeten Hardware vorgegeben. Abhängig von der Art der Anwendung wird aber die Möglichkeit offen bleiben müssen, diese Umgebung unterschiedlich zu organisieren.
- Die Umgebung, in der die längerfristige Speicherung von Objekten erfolgt, ist ebenfalls durch die Art der verwendeten Hintergrundspeicher bestimmt. Abhängig von den Anforderungen einer Anwendung kann aber eine unterschiedliche Organisation des Hintergrundspeichers sinnvoll sein.

---

<sup>6</sup> Dies gilt sicher nicht für alle Objekte, da es welche geben muß, die diese Abstraktion erst schaffen. Dennoch ist eine viel gezieltere Interaktion zwischen Anwenderebene und Laufzeitsystem möglich, als dies zwischen einem Unix-Prozeß und dem Unix-Kern möglich wäre.

Aus diesen Grundsätzen ergibt sich etwa folgendes Bild:

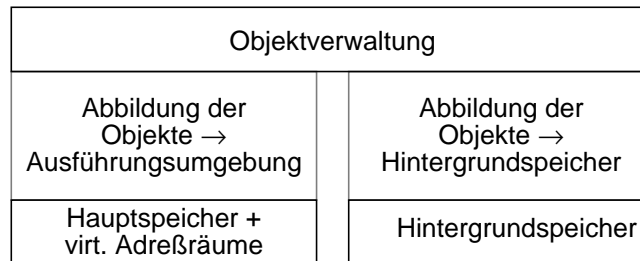
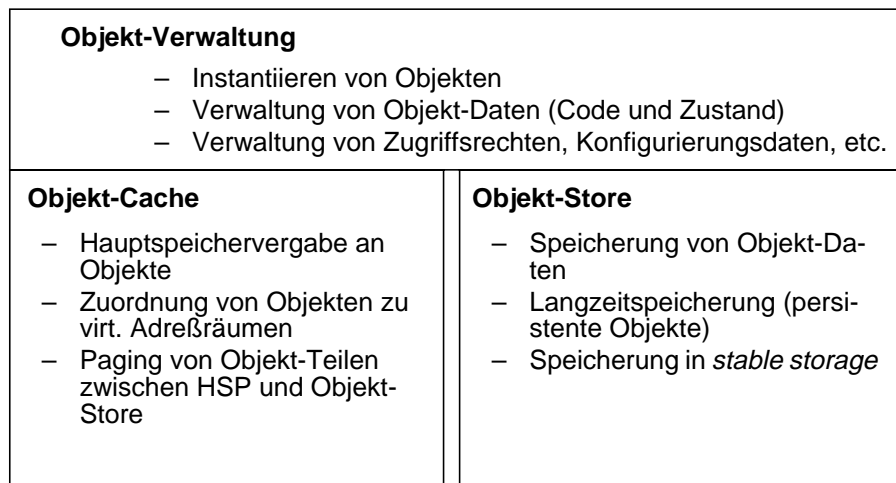


Abbildung der Objekte auf Ausführungsumgebung und Hintergrundspeicher

Während früher in Betriebssystemen die Sichtweise verbreitet war, Anwendungen im Hauptspeicher zu verwalten und im Fall von Engpässen auf Hintergrundspeicher auszulagern (z. B. die ursprüngliche UNIX-Sicht), hat sich in den letzten Jahren die umgekehrte Betrachtungsweise mehr und mehr durchgesetzt. Auf dem Hintergrundspeicher werden die Daten der Anwendung organisiert und gespeichert, der Hauptspeicher wird lediglich als Cache für die zur Ausführung benötigten Teile genutzt.

Analog zu dieser Betrachtungsweise können die Mechanismen zur Abbildung von Objekten in eine Ausführungsumgebung als Cache für die Ausführung von Operationen auf Objekten betrachtet werden, die in der Objektverwaltung bzw. auf Hintergrundspeicher liegen.

Für PM/RTS ergibt sich damit folgende Gliederung der Objekt- und Speicherverwaltung:



Gliederung in *Object-Management*, *Object-Cache* und *Object-Store* und Beispiele für Funktionen der Komponenten

Die Mechanismen der beschriebenen Komponenten sind in PM/RTS nicht fest vorgegeben. Es ist prinzipiell möglich durch Instantiierung neuer Metaobjekte beispielsweise einen speziellen Objekt-Cache mit anderen Paging-Strategien zu erzeugen, der dann für eine spezielle Anwendung genutzt wird, während die anderen Anwendungen unabhängig davon mit dem Standard-Objekt-Cache arbeiten.

Analog könnte z. B. für eine Datenbankanwendung ein modifizierter Objekt-Store mit angepaßten Pufferungsalgorithmen instantiiert werden.

In einem verteilten System sind die Schnittstellen zwischen Metaobjekten weitgehend ortstransparent. Die Komponenten der Objekt- und Speicherverwaltung können somit problemlos an verschiedenen Orten im verteilten System platziert sein. Daraus ergeben sich interessante Perspektiven für das Modell — einige Beispiele:

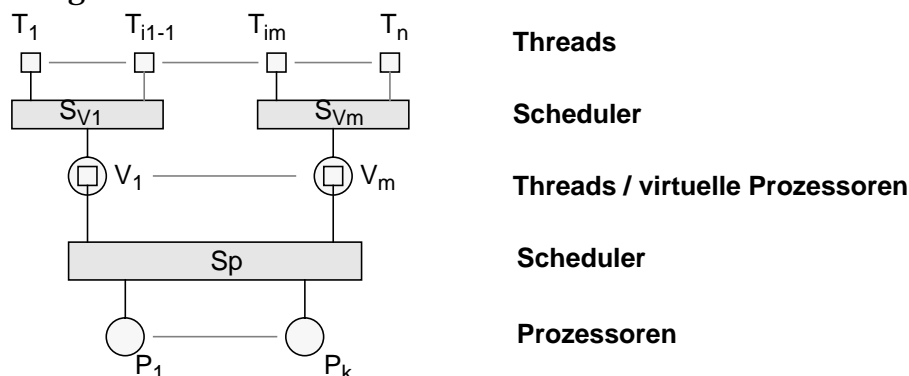
- Eine Objekt-Migration läßt sich auf das Cachen des Objekts auf einem anderen Knoten zurückführen.
- Die Platzierung eines Objekts in mehreren Stores kann als Basis für replizierte Speicherung dienen.
- Die Abbildung eines Objekts in mehrere Objekt-Caches auf verschiedenen Knoten ermöglicht ein *Sharing* von Objekten zwischen Knoten (analog zu *distributed shared memory*)

Ein Vorschlag für die Objekt- und Speicherverwaltung des PM Laufzeitsystems wird in [Kleinöder92] gegeben werden. Die Arbeit wird begleitet durch eine prototypische Implementierung auf einem Mach-System, die die Funktionsfähigkeit des Konzepts an sich zeigen soll.

### 3.2.4 Aktivitätsträgerverwaltung

In der PM Prototypimplementierung (siehe Kapitel 2) wurde ein das PM/MTE System entwickelt. Es bietet eine hervorragende Grundlage für die Strukturierung der Aktivitätsträgerverwaltung im PM Laufzeitsystem.

Bei der Modellierung geht es darum zu einer rekursiven Struktur zu kommen, die es erlaubt eine Hierarchie von virtuellen Ausführungspfaden zu beschreiben, die möglicherweise verschiedenen Schedulingstrategien unterworfen sind. Damit soll die Abstraktion von Aktivitätsträgern, wie sie auf unterster Ebene reale Prozessoren bilden, in mehreren Schichten bis zu den Aktivitätsträgern des PM Objektmodells vollzogen werden.



Kernstück des Systems bilden wiederverwendbare Komponenten wie Prozessorobjekte und Schedulingobjekte. Sie erzeugen zusammen eine neue Abstraktion von Aktivitätsträgerobjekten (Threads) auf einer höheren Ebene. Ein Thread selbst kann wiederum als (virtueller) Prozessor aufgefaßt werden. Einer Menge

solcher virtueller Prozessoren kann wieder ein Scheduler zugeordnet werden, der Threads einer (noch) höheren Abstraktion erzeugt.

Die Prozessorobjekte können auf unterster Ebene Hardwareprozessoren sein. In der MTE Implementierung standen sie für UNIX Prozesse. Sie könnten aber auch Interpreter für bestimmte Sprachen darstellen.

- Die Überlegungen in diesem Teilgebiet sollen zu einer Struktur des PM/RTS führen, die das Multiplexen von Aktivitätsträgern implementiert. Interessante Gesichtspunkte sind hier die Wiederverwendbarkeit von Strategieobjekten auf den verschiedenen Metaschichten, wie z.B. Scheduler, und die Anpaßbarkeit des Systems an verschiedene Anforderungen.

### 3.2.5 Kommunikationsprotokolle

Die objektorientierte Strukturierung einer Implementierung für verschiedene Kommunikationsprotokolle wird in [Wippich91] und [Geys91] beispielhaft vorgenommen. Die Implementierungen stützen sich auf den PM Prototypen (siehe Kapitel 2) können aber als Beitrag zur Strukturierung des PM/RTS gesehen werden. Ein Überblick über die Implementierungen enthält [Monge92].

### 3.2.6 Künftige Schwerpunkte

Das Laufzeitsystem muß künftig auch den Neuerungen im Programmiermodell Rechnung tragen.

- Die Integration von Fehlertoleranz durch Transaktionen und Replikationen sind zu überlegen.  
Wie Fehlertoleranz durch einen Votiermechanismus in replizierten Objekten erreicht wird, wurde in [Kramer91] gezeigt. Die Arbeit orientiert sich marginal an dem PM Objektmodell, so daß eine Integration der Erkenntnisse nötig ist.
- In welcher Form Rechte in einem objektorientierten verteilten System vergeben werden können, sowie deren Auswirkungen auf die verschiedenen Schichten des Laufzeitsystems sind zu untersuchen.

## 4. Zusammenfassung

PM ist ein Konzept für ein verteiltes objektorientiertes System. Wir haben gezeigt wie sich die Überlegungen zu einem Programmier- oder Objektmodell für verteiltes Programmieren aus den Erfahrungen und Erkenntnissen einer prototypischen Implementierung verändert haben.

Das PM Projekt besteht aus zwei Forschungsschwerpunkten, wobei der erste Schwerpunkt sich mit verteilten objektorientierten Programmiermodellen beschäftigt. Es wurden bereits Hinweise gegeben wie sich klassische objektorientier-

te Programmiermodelle erweitern lassen, um der Verteiltheit gerecht zu werden, z.B. Einschränkungen auf den Parameterübergabesemantiken, Beschreibung von Privatheit und Zustand von Objekten.

Der zweite Schwerpunkt liegt in der Architektur eines Laufzeitsystems für das verteilte Programmiermodell. Es realisiert eine Umgebung zum Ablauf verteilter Anwendungen, wie sie in dem Programmiermodell beschrieben werden können. Die Strukturierung des Laufzeitsystems erfolgt im gleichen Objektmodell, wobei schrittweise Abstraktionen des Modells aufgegeben werden, um diese zu realisieren. Das Laufzeitsystem besteht damit aus einer Menge von Metasystemen bzw. Metaobjekten.

Die Strukturierung des Laufzeitsystems erfolgt so, daß das System an verschiedene Aufgaben anpaßbar ist. Diese Anpassbarkeit wird durch geeigneten Austausch von Einzelteilen des Laufzeitsystems erreicht. Die Austauschbarkeit von Einheiten aus einem Objektsystem wird wiederum durch Mechanismen des Programmiermodells unterstützt, wie z.B. Zusicherung von Privatheit.

Die Erkenntnisse aus der Strukturierung des Laufzeitsystems führen zu Forderungen an das Programmiermodell. Änderungen des Modells erfordern Änderungen in dem implementierenden System. Die beiden Schwerpunkte beeinflussen und befruchten sich also gegenseitig.

## 5. Danksagung

Das PM Projekt hatte im Laufe seiner Entwicklung viele direkte und indirekte Mitarbeiter, die durch ihre wissenschaftlichen Arbeiten wie Dissertationen, Studien- oder Diplomarbeiten das Projekt beeinflußt haben und denen wir hier danken möchten.

Namentlich bedanken wollen wir uns bei den ständigen Mitgliedern des PM Arbeitskreises, ohne die wesentliche Teile dieser Arbeit nicht zustandegekommen wäre. Erwähnen möchten wir hier Prof. Fridolin Hofmann, Raúl Monge und Babette Rotzoll.

## 6. Literaturverzeichnis

- [Aufderheide90] Aufderheide, Stefan; "Entwurf und Implementierung einer Konfigurations-sprache für PM"; Studienarbeit 25/90, IMMD 4, Universität Erlangen-Nürnberg; Juli 1990
- [DeRemer76] DeRemer, Frank; Kron, Hans; "Programming-in-the-Large versus Programming-in-the-Small"; IEEE Transactions on Software Engineering; June 1976;pp. 321-327
- [Eirich91] Eirich, Thomas; Hauck, Franz J.; "Inheritance by Aggregation"; submitted for publication; January 1991
- [Ellis90] Ellis, Margaret A.; Stroustrup, Bjarne; "The Annotated C++ Reference Manual"; Addison-Wesley Publishing Company; Reading Massachusetts; 1990

- [Fäustle92] Fäustle, Michael; "Konfiguration in verteilten objektorientierten Systemen"; Dissertation (in Vorbereitung); IMMD 4, Universität Erlangen-Nürnberg; April 1992
- [Götz91] Götz, Peter; "Entwicklung und Implementierung einer Multithreadumgebung in einem oder mehreren UNIX Prozessen"; Studienarbeit 50/90, IMMD 4, Universität Erlangen-Nürnberg; Januar 1991
- [Gorr89] Gorr, Manfred; "Spezifikation und Implementierung eines Laufzeitsystems für PM Objekte unter UNIX"; Diplomarbeit; IMMD 4; Universität Erlangen-Nürnberg; 1989
- [Hauck89] Hauck, Franz J.; "Implementierung eines Stubgenerators als Phase des PM/PL Compilers"; Diplomarbeit 13/89; IMMD 4, Universität Erlangen-Nürnberg; Juli 1989
- [Helm91] Helm, Bernd; "Entwurf und Implementierung eines objekt-orientierten Betriebssystemkerns für Multiprozessorsysteme"; Diplomarbeit 21/91; IMMD 4, Universität Erlangen-Nürnberg; August 1991
- [Hofmann90] Hofmann, Wilhelm; "Die Koordinierung in Betriebssystemen für Multiprozessoren"; Dissertation, IMMD 4, Universität Erlangen-Nürnberg; IMMD Band 23(10); Juli 1990
- [Hofmann91] Hofmann, Fridolin; Schlenk, Peter; Eirich, Thomas; "Encapsulation and Interaction in Future Operating Systems"; Workshop "Operating systems of the 90s and beyond"; to appear, 1991
- [Hutchinson87a] Hutchinson, Norman C.; Raj, Rajendra K.; Black, Andrew P.; Levy, Henry M.; Jul, Eric; "The Emerald Programming Language"; Technical Report 87-10-07; University of Washington, Seattle; October 1987 (revised August 1988)
- [Kaiserswerth88] Kaiserswerth, M.; Schlenk P.; "PM: A new foundation for distributed multiprocessor operating systems"; internal report; IMMD 4, University of Erlangen-Nürnberg; July 1988
- [Kaiserswerth89] Kaiserswerth, Matthias; Schlenk, Peter; "PM: Ein Betriebssystemkonzept für verteilte Multiprozessorsysteme", Bericht 89/3 des SFB 182 Multiprozessor- und Netzwerkkonfigurationen, Teilbereich B2; Universität Erlangen; IMMD Band 22(4); Februar 1989; pp. 1-20
- [Kleinöder92] Kleinöder, Jürgen; "Objekt- und Speicherverwaltung in einer offenen objekt-orientierten Betriebssystemarchitektur"; Dissertation (in Vorbereitung); IMMD 4, Universität Erlangen-Nürnberg; April 1992
- [Kramer85] Kramer, J.; Magee, J.; "Dynamic Configuration for Distributed Systems"; IEEE Transactions on Software Engineering; Vol. SE-11, No. 4; April 1985; pp. 424-436
- [Kramer91] Kramer, Martin; "Software-implementierte Fehlertoleranzmechanismen für objektorientierte verteilte Systeme"; Dissertation; IMMD 4, Universität Erlangen-Nürnberg; Oktober 1991
- [Küpfer90] Küpfer, Uwe; "Koordinierung paralleler Objekte in PM/PL"; Diplomarbeit 19/90; IMMD 4, Universität Erlangen-Nürnberg; Mai 1990
- [Mackert83] Mackert, Lothar; "Modellierung, Spezifikation und korrekte Realisierung von asynchronen Systemen"; Dissertation, IMMD 4, Universität Erlangen-Nürnberg; IMMD Band 16(7); Juli 1983
- [Maes87a] Maes, Pattie; "Computational Reflection"; Technical Report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel; 1987
- [Maes88] Maes, Pattie [Ed.; Nardi, Daniele [Ed.; "Meta-level Architecture and Reflection"; Workshop Papers held in Alghero, Italy, October 1986; North Holland, 1988

- [Meyer88] Meyer, Bertrand; "Object-Oriented Software Construction"; Prentice Hall International; Hemel Hempstead, Hertfordshire; 1988
- [Monge89a] Monge, Raúl; "Communication, Management and Naming of PM Global Objects", Bericht 89/3 des SFB 182 Multiprozessor- und Netzwerkkonfigurationen, Teilbereich B2; Universität Erlangen; IMMD Band 22(4); Februar 1989; pp. 60-71
- [Monge89b] Monge, Raúl; "PM-RPC: An Inter-object Communication Mechanism for Global Objects", Bericht 89/3 des SFB 182 Multiprozessor- und Netzwerkkonfigurationen, Teilbereich B2; Universität Erlangen; IMMD Band 22(4); Februar 1989; pp.72-87
- [Monge92] Monge, Raúl; "Kommunikation in verteilten objektorientierten Systemen"; Dissertation (in Vorbereitung); IMMD 4, Universität Erlangen-Nürnberg; März 1992
- [Prisille91] Prisille, Kay; "Implementierung eines Debuggers für eine Multithreadumgebung"; Studienarbeit 8/91, IMMD 4, Universität Erlangen-Nürnberg; März 1991
- [Pruy89] Pruy, Rainer; "Ein Ansatz zur Integration von abstrakten Koordinierungsangaben in eine Programmiersprache", Diplomarbeit, IMMD IV; Universität Erlangen, 1989
- [Rao91] Rao, Ramana; "Implementational Reflection in Silica"; ECOOP '91 Proceedings; July 1991; pp. 251-267
- [Sakkinen89] Sakkinen, Marku; "Disciplined Inheritance"; ECOOP 89 Proceedings; Cambridge University Press; Cambridge UK; 1989
- [Schlenk89a] Schlenk, Peter; "PM: A new foundation for distributed multiprocessor operating systems"; Bericht 89/3 des SFB 182 Multiprozessor- und Netzwerkkonfigurationen, Teilbereich B2; Universität Erlangen; IMMD Band 22(4); Februar 1989
- [Schlenk89c] Schlenk, Peter; "Kommunikationsausprägungen in verteilten Programmen"; Bericht 89/5 des SFB 182 Multiprozessor- und Netzwerkkonfigurationen; Universität Erlangen; IMMD Band 22(13); Oktober 1989; pp. 29-50
- [Schröer89] Schröer, Joachim; "Spezifikation und Implementierung eines verteilten Name Servers für PM"; Diplomarbeit 21/89; IMMD 4, Universität Erlangen-Nürnberg; Dezember 1989
- [Snyder87] Snyder, Alan; "Inheritance and the Development of Encapsulated Software Components"; in Research Directions in Object-Oriented Programming; Shriver B., Wegner P. (Eds.); MIT Press 1987; pp. 165-188
- [XDR87] Sun Technical Notes; "The eXternal Data Representation"; Stanford, 1987