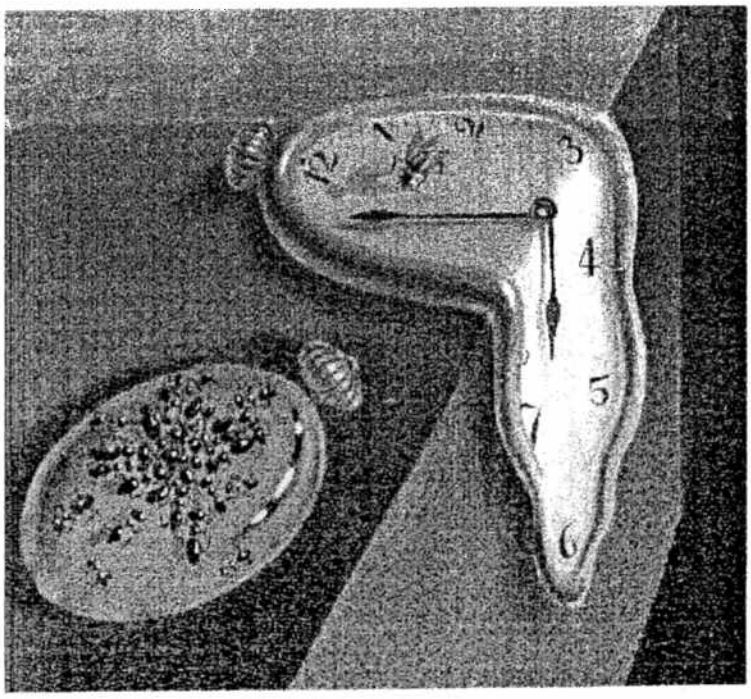
Institut für Automation
Abt. für Automatisierungssysteme

Technische
Universität
Wien

Projektbericht Nr. 183/1-40
Dezember 1993

DOBOS – Konzept eines Distributed Object-Based Operating Systems

U. Schmid, W. Kastner



Ausschnitt aus: Salvador Dali, "Die Beständigkeit der Erinnerung"

DOBOS

Konzept eines Distributed Object-Based Operating Systems (Preliminary Version)

U. Schmid, W. Kastner

22. März 1994

Zusammenfassung

Am Institut für Automation der TU Wien wird zur Zeit ein flexibles Monitoring-System Versatile Timing Analyzer (VTA)[1] *zur Erfassung und Auswertung des zeitlichen Verhaltens von verteilten (Echtzeit-)Systemen entwickelt. Bei der Entwicklung eines derart komplexen heterogenen verteilten Systems müssen geeignete Methoden für den Entwurf der Software angewandt werden, etwa jene, wie sie das objektorientierte Paradigma bietet. Aufgrund der Tatsache, daß die für uns verfügbaren objektorientierte Systeme primär für sequentielle Umgebungen gedacht sind, wurde im Rahmen des Projekts mit der Konzeption und Realisierung eines leistungsfähigen Distributed Object-Based Operating Systems* DOBOS *als Basis für die Implementierung des VTAs begonnen. Die für* DOBOS *entwickelten Konzepte erlauben eine homogene Vereinigung der beiden „Welten" Objektorientierung und Parallelität, ohne dabei aber die Offenheit gegenüber „normaler" Software und darüberhinaus die einfache und effiziente Implementierbarkeit preiszugeben.*

1 Einleitung

Objektorientierung ist zweifellos eines der Paradigmen der Informatik, dessen Bedeutung in Forschung und Entwicklung – und mittlerweile auch in der Praxis – unumstritten ist. Eine ähnlich Aussage trifft auch für parallele bzw. verteilte Systeme[2] zu. Im Gegensatz dazu ist die naheliegende Integration dieser beiden „Welten", Objektorientierung und Parallelität, zwar Gegenstand aktiver Forschung (siehe z.B. [1], [4], [6], [9], [10]), in der Praxis verwendbare Systeme sind jedoch rar (siehe z.B. [4], [6]).

Diese Tatsache wurde uns sehr rasch klar, als wir im Rahmen eines größeren Forschungsprojekts vor dem Problem standen, eine nach objektorientierten Prinzipien entworfene Software für ein heterogenes verteiltes System (bestehend aus einer Sun-Workstation

[1] gefördert vom *Österreichischen Fonds zur Förderung der wissenschaftlichen Forschung*, Projektnummer P8390-TEC.

[2] Wir verwenden den Terminus verteilte Systeme für ein kollaborierendes System lose gekoppelter autonomer Rechner; der Oberbegriff der parallelen Systeme umfaßt zusätzlich auch die eng gekoppelten Multiprozessor- bzw. Multitaskingsysteme.

unter SunOS und mehreren 68030-basierenden VME-Mikroprozessorsystemen unter dem Echtzeitbetriebssystem pSOS) realisieren zu müssen; konkretes Ziel dieses Projektes ist die Entwicklung eines (verteilten) Monitoring-Systems *Versatile Timing Analyzer VTA* für die experimentelle Timing-Analyse in verteilten Echtzeitsystemen. Inadäquate Modellierungsmittel, fehlende sprachliche Mechanismen und nicht zuletzt Probleme mit der Verteilung auf eine heterogene Hardware-Architektur ließen existierende „kommerzielle" Systeme (C++, Smalltalk) sehr bald ausscheiden. Die in diesem Zuge durchgeführte, grobe Requirements-Analyse ergab etwa folgendes Anforderungsprofil für eine „VTA-taugliche" Kombination Betriebssystem + Implementierungssprache:

- Standard-Konzepte der Objektorientierung (Inheritance, Polymorphismus, usw.),
- dynamische Klassen,
- statisches Type-Checking,
- persistente Objekte,
- Unterstützung von Parallelität,
- Object Location Transparency trotz verteiltem (heterogenen) System,
- C-ähnliche Syntax der Implementierungssprache,
- einfache Anbindung/Integration existierender (C-)Software-Systeme,
- minimaler System-Overhead,
- einfache Abbildung auf SunOS und pSOS.

Eine kurze Evaluation existierender Research-Prototypensysteme (siehe z.B. [4]) zeigte ebenfalls sehr rasch, daß sie dem Anforderungsprofil nicht entsprachen: Neben allfälligen Portierungsproblemen auf die heterogene Hardware lassen derartige Systeme die Anbindung/Integration von Standard-Softwarekomponenten und Tools (z.B. UNIX Interface-Builder, Software zur Visualisierung) nicht zu. Deshalb wurde nach einem (pragmatischen) Konzept eines *Distributed Object Based Operating Systems* DOBOS[3] gesucht, das (von uns) im Rahmen des Projekts VTA auch vollständig zu realisieren ist. Als vorläufiges Ergebnis dieser Anstrengungen liegt das Konzept für DOBOS vor; an der Definition der Syntax der Implementierungssprache bzw. dem darauf aufbauenden Precompiler für C und der Detailkonzeption bzw. Realisierung des eigentlichen Betriebssystems wird zur Zeit noch gearbeitet.

Natürlich ist die Konzeption eines Systems wie DOBOS ohne die intensive Mitarbeit aller im Projekt VTA arbeitenden Kollegen undenkbar. Besonderer Dank gebührt hier *Johann Klasek*, der im Kontext seiner Arbeit über das Primitive Event Management des VTAs (siehe [5]) die wichtigsten objektorientierten Konzepte für DOBOS entwickelte, sowie den Kollegen *Stefan Stöckler* und *Harald Haberstroh* für ihr Engagement in den unersetzlichen Diskussionsrunden.

[3]Die Berechtigung dieser Abkürzung in Hinblick auf die bekannte Dobos-Torte, die aus mehreren parallelen, durch Cremelagen getrennten dünnen Teigschichten besteht, wird im Abschnitt 4 offensichtlich.

Der Rest der vorliegende Arbeit ist wie folgt gegliedert: Im Abschnitt 2 wird kurz das „klassische“ objektorientierte Paradigma skizziert und dessen Vorteile in Entwurf und Implementierung von Software-Systemen dargelegt. Der daran anschließende Abschnitt 3 bietet eine Einführung in das verteilte Monitoring-System Versatile Timing Analyzer und verdeutlicht anhand der Design-Ziele, warum sich die erläuterten objektorientierten Betrachtungsweisen und Entwicklungsmethoden so gut für den Entwurf der VTA System-Software eignen. Die Konzepte von DOBOS selbst werden im darauffolgenden Abschnitt 4 vorgestellt, wobei der Schwerpunkt auf der Modellierung und nicht auf Aspekten der Implementierung liegt.

2 Objektorientierung

Die objektorientierte Betrachtungsweise bietet die konsequente Fortsetzung der bis heute in der Informatik entwickelten Abstraktionsmechanismen und Strukturierungsprinzipien. Ihre Bedeutung läßt sich am besten aufzeigen, wenn man die geschichtliche Entwicklung der verschiedenen Programmierkonzepte analysiert, was aber den Rahmen dieser kurzen Einführung bei weitem sprengen würde. So sei nur darauf hingewiesen, daß der eigentliche Durchbruch wohl durch das Modulkonzept gelang. Dieses gestattet, komplexe Software-Systeme in relativ unabhängige Komponenten zu zerlegen, die untereinander ausschließlich über definierte Schnittstellen kommunizieren. Jedes *Modul* setzt sich aus einer „sichtbaren“ Schnittstelle, welche die Deklaration von Typen, Konstanten, Variablen und Prozeduren enthält, und einer „unsichtbaren“ Implementation, die im wesentlichen aus Deklarationen von Typen, Konstanten und Variablen sowie der eigentlichen Implementierung der Prozeduren nebst Initialisierungsteil für das Modul selbst besteht, zusammen. Alle in der Schnittstelle des Moduls vorkommenden Variablen haben permanenten Charakter, sie behalten somit ihren Wert auch nach Verlassen des Moduls. Auf diese Art kann die Implementierung von Datenstrukturen und Prozeduren verborgen, ja sogar nach Belieben ausgetauscht und modifiziert werden, solange die Schnittstelle nicht geändert wird (*information hiding*). Module bilden statische Objekte, die nur einmal definiert und ebenso nur ein einziges mal instanziiert werden dürfen.

Die logische Erweiterung dieses Konzepts führt zu den *abstrakten Datentypen* der objektbasierenden Welt. Mit ihrer Hilfe kann eine Menge von Daten und die darauf anwendbaren Operationen, wobei wiederum das Verhalten der Operationen von der Repräsentation der Daten unabhängig ist, bestimmt werden. Abstrakte Datentypen dienen als Schablonen, aus denen beliebig viele *Objekte* erzeugt werden können. Objekte verfügen über einen inneren Zustand, der vor der Außenwelt geschützt und verborgen bleibt und nur den Operationen innerhalb des Objekts selbst zugänglich gemacht wird. Daten, die den inneren Zustand des Objekts wiedergeben, werden *Instanzvariablen* genannt und gegen willkürlichen Zugriff abgekapselt. Information über sie und Modifikation an ihnen wird nur über wohldefinierte Operationen, die *Methoden* genannt werden, erlaubt (*data encapsulation*). Die Menge aller Methoden bestimmt die Schnittstelle und somit letztlich auch das Verhalten des Objekts.

Die objektorientierte Programmierung bietet nun weitere Mechanismen an, die es gestatten, einen bereits bestehenden abstrakten Datentyp auf einfache und flexible Art

und Weise zu ergänzen oder zu verändern, ohne das Prinzip des Information Hidings zu verletzen. Die Pendants der abstrakten Datentypen der objektbasierenden Welt werden in der objektorientierten Welt *Klassen* genannt. Um eine existierende Klasse zu erweitern oder zu spezialisieren, ist es möglich, aus einer angegebenen Klasse eine neue Klasse abzuleiten, wobei die Instanzvariablen und Methoden vererbt werden (*inheritance*). Die entstandene Unterklasse unterscheidet sich von ihrer Oberklasse durch

- **Erweiterung**. Neue Instanzvariablen und Methoden wurden hinzugefügt.
- **Variation**. Methoden der Oberklasse wurden neudefiniert (oder unterbunden).
- **Spezialisierung**. Erweiterung und Variation wurden kombiniert.

Eine gültige Methode, die nicht in der aktuellen Klasse spezifiziert ist, muß natürlich in einer der Oberklassen zu finden sein. Klarerweise ist die Klasse und somit die konkrete Implementierung einer Operation (Methode) eines Objekts von dem Objekt abhängig, auf das die jeweilige Operation (zur Laufzeit) angewendet wird. Die Fortführung dieses Gedankens auf die Laufzeit des Programmes führt schließlich zum wichtigen Konzept des *Polymorphismus*: Eine polymorphe Operation (Methode) ist eine solche, die auf Objekte verschiedener Klassen angewendet werden kann; sie hat zwar denselben Namen und dieselbe Schnittstelle, muß aber natürlich in jeder Klasse spezifisch implementiert werden. Die Entscheidung, welche Methode nun tatsächlich angewendet werden muß, wird erst zur Laufzeit, abhängig von der Klasse des konkreten Objekts, getroffen. Die Einführung des Konzepts einer polymorphen Variablen, der Referenzen auf Objekte verschiedener Klassen zugewiesen werden dürfen, vervollständigt diesen äußerst leistungsfähigen Mechanismus.

Die realisierungstechnische Basis dafür bildet das dynamischen Binden[4]. In herkömmlichen Programmiersprachen wird ein Prozeduraufruf in der Regel statisch (schon zur Übersetzungszeit) in die entsprechende Verzweigung im Programmcode umgesetzt; beim dynamischen Binden wird erst zur Laufzeit über das Sprungziel entschieden. Mit Hilfe dieses Mechanismus ist auch die Realisierung dynamischer Klassen(-Referenzen) möglich: Hier ist bei einer Referenz auf eine Klasse (bei der Instanziierung eines Objekts) aus dem statischen Programmtext (zur Übersetzungszeit) nicht ersichtlich, um welche Klasse es sich nun tatsächlich handelt. Dynamisch referenzierte Klassen (und gegebenenfalls auch einige Oberklassen) müssen daher zur Laufzeit, bei der Exekution der Instanziierung, geladen werden. Der Vorteil dieses Konzepts liegt in der Tatsache begründet, daß im Falle des Hinzufügens einer neuen Klassendefinition der *verwendende* Programmtext *nicht* geändert werden muß!

Abschließend sollen noch kurz die – für uns primär relevanten – Vorteile und Verbesserungen der Anwendung eines objektorientierten Entwurfs im Vergleich zur klassischen prozeduralen Programmierung stichwortartig aufgezählt werden:

- **Verbesserte Modellierung**. Die Modellierung von Systemen der realen Welt wird durch Verkleinerung der konzeptionellen Lücke zwischen den Objekten der Modellwelt und den Objekten der realen Welt ganz wesentlich verbessert. Daraus resultiert unter anderem auch

[4] d.h., die Zuordnung Objekt → Klasse.

- Erleichterung des Projektmanagements durch konsequente (hierarchische) Modularisierung.
- Möglichkeit des Rapid Prototypings.
- Deutlich verkürzte Anpassungszeit bei späterer („vernünftiger", das Real- und demzufolge das Modellsystem nicht überall betreffender) Änderung der Benutzeranforderung an ein Programm.

- **Wiederverwendbarkeit von (Standard-)Softwarebausteinen.** Durch die Vererbung ist es möglich, bei der Modellierung/Implementierung neuer Klassen auf bestehende zurückzugreifen. Auch dieses Argument stützt einige der vorher erwähnten Punkte, garantiert aber zusätzlich noch
  - Erhöhte Zuverlässigkeit und Korrektheit von Programmen.
  - Bessere Wartbarkeit und geringeren Wartungsaufwand.

3 Der Versatile Timing Analyzer VTA

Der gegenwärtig am Institut für Automation/TU-Wien in Entwicklung befindliche Versatile Timing Analyzer VTA ist ein flexibles *Monitoring-System* zur experimentellen Beobachtung des (Zeit-)Verhaltens eines verteilten Echtzeitsystems (*Targets*); eine kurze Übersicht ist in [7] zu finden. Das folgende Bild zeigt die prinzipielle Struktur des Research-Prototypen, der (aus pragmatischen Gründen) zunächst für VMEbus-basierende Target-Systeme konzipiert wird.

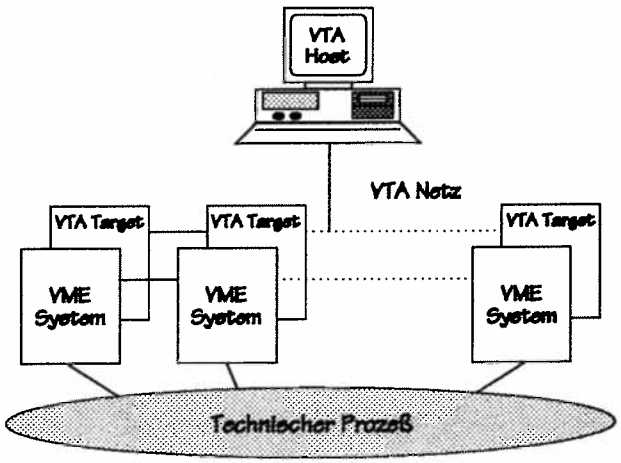


Abbildung 1: VTA-Architektur

Vom Prinzip her ist der VTA ein ereignisbasierendes Monitoring-System, welches als Reaktion auf bestimmte Zustandsübergänge im Target (*Ereignisse*) bestimmte Monitoring-Aktivitäten ausführt. Sowohl die zu erfassenden Ereignisse als auch die auszuführenden Aktionen können dabei vom Benutzer frei definiert bzw. programmiert werden.

Die primäre Überwachung des Targets ist Aufgabe der *VTA-Targets*, die über ein eigenes *VTA-Netz* untereinander und mit dem *VTA-Host* gekoppelt sind. Bei den VTA-Targets handelt es sich um spezielle CPU-Karten, die in freie Slots der relevanten Knoten des Targets gesteckt werden und die von dort stammenden „primitiven" Ereignisse entgegennehmen und weiterverarbeiten. Für die Analyse des zeitlichen Verhaltens eines Targets sind aber vor allem globale Zusammenhänge von Bedeutung; deren Erkennung obliegt dem Kernstück des VTAs, dem verteilten Event-Recognizer. Er überprüft ankommende primitive Ereignisse und stellt fest, ob die vom Benutzer über eine spezielle Sprache (siehe [8]) spezifizierten globalen Zustandsänderungen eingetreten sind; gegebenenfalls werden die damit verbundenen Aktionen ausgeführt. Auf Basis dieses Mechanismus können natürlich beliebig komplexe Meßmethoden einfach und effizient programmiert werden.

Der VTA-Host hat die Benutzerschnittstelle für das gesamte Monitoring-System bereitzustellen. Basierend auf einem leistungsfähigen Multiwindow-System werden hier Möglichkeiten für die Systemkonfiguration, die Definition von Ereignissen und Aktionen, die Aktivierung und Deaktivierung von Monitoring-Vorgängen und schließlich die Analyse und Visualisierung der erfaßten Daten geboten.

Die konkreten Anwendungsmöglichkeiten des VTAs sind vielfältig und sowohl für die Forschung als auch die unmittelbare Praxis von Interesse. Zwei der wichtigsten Anwendungskategorien sind

- Experimentelle Erfassung/Analyse von Zeitparametern des technischen Prozesses für die theoretische Forschung oder das praktische Requirements Engineering.
- Einsatz des VTAs zur Überprüfung des Zeitverhaltens eines verteilten Echtzeitsystems, v.a. in der Testphase.

Dieses breite Feld gestattet natürlich nicht, die potentiellen Einsatzmöglichkeiten schon jetzt erschöpfend zu erfassen; das Konzept des VTAs muß daher in Hinblick auf maximale Flexibilität in bezug auf zukünftige Anwendungen erstellt werden. Daß objektorietiertes Design derartige Flexibilität sehr gut unterstützt, ist evident.

Eine ganze Reihe weiterer Punkte zeigt, warum Flexibilität geradezu *das* zentrale Kriterium für das Design des VTAs – und damit die Anwendung eines objektorientierten Entwurfs – ist:

- **Verschiedenste Target Hardware.** Der VTA soll Target-Systeme unterstützen, die aus (möglicherweise unterschiedlichen) busgekoppelten Multiprozessor-Knoten bestehen, die ihrerseits über ein beliebiges Netzwerk verbunden sein können. Keinerlei Restriktionen dürfen durch die zugrundeliegende Target-Hardware gegeben sein: Pipeline-Architekturen, Caches und Memory-Management Units sollen, für den Benutzer transparent, unterstützt werden.

  Zu diesem Zweck muß das Monitoring-System flexibel gestaltet und den Objekten der realen Welt, im konkreten Falle also der Target-Hardware, angepaßt werden. Benötigte Software-Komponenten können dazu elegant über (dynamische) Klassen und deren Objekte modelliert werden. Um die interne Software-Struktur einfach zu halten, werden beispielsweise Klassen definiert, deren gemeinsame Oberklasse

(standardisierte) Methoden für den Zugriff auf die Target-Hardware festlegt. Die Implementierung der Methoden ist jedoch in den abgeleiteten Klassen versteckt, die sie entsprechend den Hardware-Anforderungen und Gegebenheiten erledigen.

In diesem Zusammenhang ist zu bemerken, daß die im Zuge verschiedener Generationen von Prozessoren einer Prozessorfamilie (z.B. Motorola 68000, 68010, 68020, 68030, 68040) zu beobachtende „Evolution der Mechanismen" optimal mit der Idee der Vererbung im objektorientierten Modell in Einklang steht. Die Verfügbarkeit eines neuen Prozessors erfordert dann nur die Bereitstellung einer entsprechenden Klasse, die von einem geeigneten Vorgängermodell abgeleitet werden kann; diese Aufgabe kann übrigens ohne weiteres von einem Anwender übernommen werden. Die für die Integration in das Gesamtsystem erforderliche dynamische (Re)konfiguration scheint durch den Mechanismus der Polymorphie bzw. der dynamischen Klassen keine Schwierigkeit zu bereiten. Somit greifen sämtliche im vorigen Abschnitt aufgezählten Argumente und Vorteile des objektorientierten Paradigmas, wie Anpassungszeit, bessere Wartbarkeit, geringerer Wartungsaufwand (die Liste könnte beliebig verlängert werden).

- **Verschiedenste Target Software.** Der VTA soll Target-Systeme unterstützen, deren Software unter Zuhilfenahme der unterschiedlichsten Entwicklungswerkzeuge und für verschiedenste Target-Betriebssysteme geschrieben wurde. Rücksicht muß im speziellem auf die dem potentiellen Benutzer vertraute Sichtweise genommen werden: Bei der Entwicklung von (Echtzeit)-Software kommen, wie auch in jener herkömmlicher kommerzieller Applikationen, bestimmte Werkzeuge zum Einsatz, die den Software-Lifecycle erleichtern. Die Palette reicht von den unterschiedlichsten CASE-Systemen über Compiler und Linker bis zu diversen Debuggern.

  Die Software-Ingenieure sind üblicherweise an die von ihren Werkzeugen „angebotenen" Sichtweisen gewöhnt, ja oft schon von diesen abhängig: Sie arbeiten mit bestimmten funktionalen Dekompositionsmechanismen, um Problemstellungen zu verfeinern, verwenden z.B. Task-Maps, um diverse Grobstrukturen zu entwerfen, implementieren Software in höheren Programmiersprachen und testen das Ganze mit Hilfe eines Debuggers. Zwingt man nun einen solchen Entwickler, bei Gebrauch des VTAs seine vertraute Sichtweise und sein eng daran gekoppeltes Wissen aufzugeben und anstelle dessen andere, vom Monitoring-System vorgegebene Abstraktionen verwenden zu müssen, so wird die Akzeptanz des VTAs (oder wenigstens die Ausnutzung seiner Möglichkeiten) im praktischen Einsatz sehr gering bleiben.

  Besser ist es daher, dem VTA die Sichtweise des Benutzers beizubringen, also im Endeffekt für jede (unterstützte) Entwicklungsumgebung eine „maßgeschneiderte" Version bereitzustellen. Aus diesem Grund sollte die System-Software des VTAs auf einem flexiblen Grundsystem basieren, das durch verschiedene *Views* zu den spezifischen VTAs erweitert wird. Diese Views werden auf die speziellen Abstraktionen der jeweils verwendeten Entwicklungswerkzeuge zugeschnitten: Auf der untersten Ebene könnten verschiedene *System-Views* die unterschiedlichen Target-Prozessoren(+Betriebssysteme) auf dem Niveau eines maschinensprachlichen System-State Debuggers abdecken; darüber sind verschiedenste High-level Language Views denkbar, die ein Monitoring auf Hochsprachenebene unterstützen, und

schließlich sogar noch (höhere) Views, die auf der Ebene von funktionalen Dekompositionen oder Task-Maps angesiedelt sind.

Es ist klar, daß ein (geschicktes) objektorientiertes Design der VTA System-Software das Problem der Bereitstellung vieler verschiedener Views gut unterstützt. So ist es möglich, einen View von einem bereits bestehenden, ähnlichen abzuleiten, und ohne Eingriff in das Grundsystem zu integrieren. Durch Offenlegung der Schnittstellen zum Grundsystem bzw. zu den anderen Views kann die Bereitstellung eines neuen Views unter Umständen sogar Anwendern überantwortet werden.

- **Programmierbare Meßmethoden.** Von zentraler Bedeutung für jedes Monitoring-System sind die angebotenen Möglichkeiten der Definition und Anwendung von Meßmethoden. Der VTA stellt dem Benutzer eine ausgefeilte objektorientierte Spezifikationssprache GOLDMINE (siehe [8]) zur Verfügung, mit deren Hilfe die kompliziertesten Meßmethoden rasch und effizient formuliert werden können; der Ableitung neuer Verfahren von geeigneten bereits existierenden kommt hier besonderer Stellenwert zu.

  Die Abbildung von GOLDMINE auf die Software des VTAs ist natürlich besonders einfach, wenn letztere objektorientiert ist. Übrigens ist das Vorhandensein von dynamischen Klassen hier von speziellem Interesse, da eine Neucompilation des VTAs bei Hinzukommen einer neuen (benutzerdefinierten) Meßmethode klarerweise nicht in Frage kommt.

Gestützt auf diese und ähnliche Argumente fiel schließlich die Entscheidung, den VTA konsequent objektorientiert zu entwerfen. In der Phase des Grobentwurfes wurde eine objektorientierte Entwurfsmethode (CRC-Cards, siehe [2]) eingesetzt, die trotz ihrer Einfachheit überraschend gute Resultate brachte. Allerdings wurde im Zuge dieser Arbeit die (zunächst hintangestellte) Frage immer dringlicher, wie eine auf Basis von CRC-Cards entworfene objektorientierte Systemstruktur letztlich auf unserer verteilte heterogene Architektur realisiert werden sollte.

Im ersten Ansatz wurde versucht, Teile der Software objektorientiert zu entwerfen und dann funktional in C unter Berücksichtigung der unterschiedlichen Betriebssysteme (SunOS, pSOS) zu implementieren. Die hierfür notwendigen, aufwendigen Kommunikationsprotokolle zogen jedoch immer wiederkehrende, intensive Absprachen der Mitarbeiter nach sich, die den Fortgang des Gesamtprojekts untragbar zu verzögern drohten. Der bei weitem bessere Weg schien zu sein, allen Entwicklern als gemeinsame Ausgangsbasis ein universelles Betriebssystem und eine geeignete Implementierungssprache, die objektorientiertes verteiltes Programmieren unterstützen, anzubieten: DOBOS (Distributed Object Based Operating System).

4 Das DOBOS Konzept

Obwohl fast alle zur Zeit existierenden objektorientierten Systeme für nicht parallele Software gedacht sind, passen Objekte gut in die Konzepte der parallelen Programmierung. Ihre logische Autonomie formt sie zu Einheiten, die im Prinzip auch gleichzeitig

exekutiert werden können. Gelingt es, die Modellierungsstärken, die sich durch das objektorientierte Paradigma ergeben, auf parallele Systeme auszuweiten, so stehen ungeahnte Möglichkeiten bei der Entwicklung komplexer Applikationen zur Verfügung. DOBOS versucht, die herkömmliche (sequentielle) objektorientierte Betrachtungsweise in Hinblick auf heterogene verteilte Systeme zu erweitern, wobei aber besonderer Wert auf

- effiziente und einfache Implementierung der neu zur Verfügung gestellten DOBOS-Mechanismen auf Basis traditioneller Betriebssysteme (u.a. UNIX) und
- einfache und trotzdem dichte Anbindung/Integration gewöhnlicher (nicht mit DOBOS erstellter) Software(-Systeme)

gelegt wird.

4.1 Passive Objekte

Fast alle existierenden sequentiellen Entwicklungsumgebungen unterstützen das objektorientierte Paradigma auf der Basis von *passiven Objekten*: Ein Objekt wird erst nach Erhalt einer Nachricht (nach einem Methodenaufruf) aktiviert. Der Empfänger dieser Nachricht beginnt daraufhin mit der Verarbeitung der entsprechenden Befehle; der Absender ist hingegen gezwungen, auf das Ende der Verarbeitung (die mit der Rückgabe etwaiger Ergebnisse verbunden sein kann) zu warten. Zu jedem Zeitpunkt kann daher immer nur ein Objekt tatsächlich aktiv sein.

Diese Betrachtungweise soll auch für verteilte Systeme unter DOBOS gelten, mit der Auflage, daß aufgrund der hier möglichen gleichzeitigen Aktivierung von Methoden passiver Objekte Mechanismen für den gegenseitigen Ausschluß bereitgestellt werden müssen. Die Methodenaufrufe werden dabei in der Regel als herkömmlichen Funktions- oder Prozeduraufrufe realisiert, was die geforderte einfach Anbindung/Integration herkömmlicher Software sehr vereinfacht.

4.2 Aktive Objekte

Wie können nun Konzepte des objektorientierten Paradigmas mit Begriffen der Parallelität (Prozeß, Kommunikation, Synchronisation) in Einklang gebracht werden? Ein Ansatz (Smalltalk-80) verfolgt das Ziel, sie nebeneinander existieren zu lassen und stellt dem Programmierer zwei unterschiedliche Arten von „Modulen“ zur Verfügung: Objekte aus der einen, Prozesse aus der anderen Welt. DOBOS hingegen räumt dem objektorientierten Paradigma Priorität ein und benutzt *aktive Objekte* als Einheit der Parallelität.

Ein aktives Objekt verfügt über ein eigenständiges inhärentes „Leben“ und seinen eigenen „Lebensraum“ (*Kontext*). Es bestimmt von sich aus, wann es bereit ist, Nachrichten (Methodenaufrufe) zu empfangen, zu verarbeiten und in diesem Zuge Methodenaufrufe an andere Objekte abzusetzen. Auf diese Weise diktiert es im Zusammenspiel mit anderen aktiven Objekten die Geschicke seiner Umgebung und formt mit allen anderen Objekten (aktiv und passiv) das Objektsystem. Aktive Objekte verwenden ihre passiven Gegenspieler vorwiegend als Dienstelemente, deren Methoden normalerweise als Funktionen im Kontext der aufrufenden aktiven Instanz ausgeführt werden. Derartige Methoden sind

daher implizit *synchron* in dem Sinne, daß der Auftraggeber grundsätzlich auf die Beendigung der aufgerufenen Methode „warten“ muß. Methoden der aktiven Objekte können hingegen wahlweise synchron oder asynchron sein.

Im Falle des Aufrufs einer *asynchronen* Methode ist der Auftraggeber nicht gezwungen, nichtstuend auf deren Termination zu warten, sondern kann währenddessen seinen eigenen Agenden nachgehen. Dieser Mechanismus ist natürlich nur in Situationen von Interesse, in denen die aufrufende Instanz keine Rücklieferung von Resultaten benötigt (Triggermethoden). Die folgende Abbildung 2 zeigt ein Beispiel, in dem das aktive Objekt *a1* eine als asynchron deklarierte Methode *m1* des aktiven Objekts *a2* aufruft. *a1* wartet nicht darauf, daß die Methode *m1* von Objekt *a2* tatsächlich zur Ausführung gelangt, sondern fährt augenblicklich mit der weiteren Exekution seiner eigenen Aktionen (wenn vorhanden) fort. Wie für jederman leicht ersichtlich können auf diese Weise Problemstellungen schneller erledigt werden: Der Grad der Parallelität des Gesamtsystems steigt mit der Anzahl der asynchronen Methodenaufrufe.

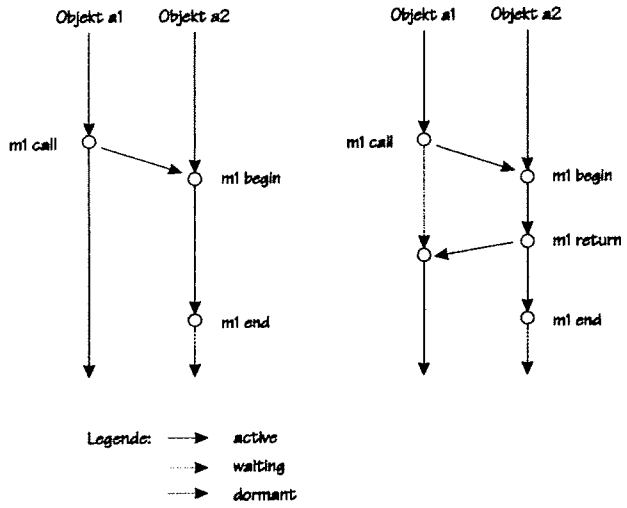


Abbildung 2: Asynchroner und synchroner Methodenaufruf

Im Gegensatz dazu bietet der Aufruf einer synchronen Methode eines aktiven Objekts die Möglichkeit, Verarbeitungsresultate zurückzuliefern und – quasi als Seiteneffekt – die beiden beteiligten aktiven Objekte zu synchronisieren. Nun könnte man zur Auffassung gelangen, daß der Aufruf synchroner Methoden in aktiven bzw. passiven Objekte a priori semantisch äquivalent sein muß; dies stimmt jedoch nicht. Erstere erlauben nämlich im Prinzip eine (in DOBOS allerdings nicht unterstützte) Mischform der Rücklieferung von Resultaten: Man betrachte dazu wiederum zwei aktive Objekte *a1* und *a2*, wobei *a1* die als synchron deklarierte Methode *m1* von *a2* aufrufen soll. Natürlich muß Objekt *a1* auf die Rücklieferung der gewünschten Parameter warten; das besagt jedoch noch lange nicht, daß die Rücklieferung erst nach Beendigung der Methode *m1* geschehen muß!

Für aktive Objekte lassen sich folgende Zustände unterscheiden (an dieser Stelle sollte nochmals auf den Umstand hingewiesen werden, daß passive Objekte über kein eigenständiges Leben verfügen und ihre Methoden im Kontext aktiver Objekte zur Ausführung gelangen):

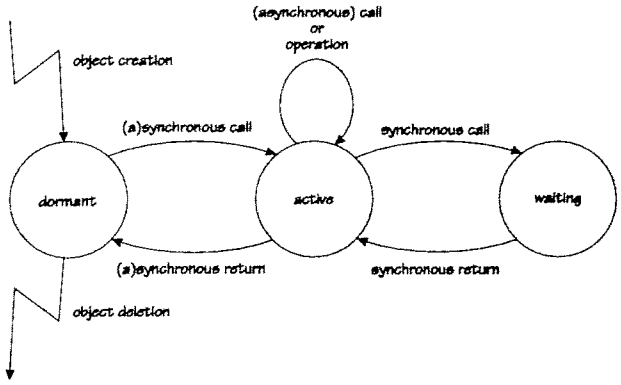


Abbildung 3: Zustandsänderung aktiver Objekte

- **Dormant State**. Nach seiner Erzeugung (Instantiierung) befindet sich ein aktives Objekt solange im Zustand *dormant* bis ein Methodenaufruf erfolgt oder das Objekt wieder zerstört wird.

- **Active State**. Wird eine Methode eines aktiven Objekts aufgerufen, so geht das Objekt vom Zustand *dormant* in den Zustand *active* über. Nun werden die Methodenspezifischen Operationen exekutiert, so beispielsweise die Instanzvariablen modifiziert, weitere Dienste entweder über asynchrone Methoden anderer aktiver Objekte oder über Methoden passiver Objekte angefordert. Jedem aktiven Objekt ist ein Puffer für eintreffende Nachrichten (Methodenanfragen) zugeordnet, die in ihm solange gespeichert werden, bis das Objekt bereit ist, sie zu verarbeiten. Die Pufferung arbeitet losgelöst vom aktuellen Zustand des Objekts, sodaß Nachrichten unabhängig von den gerade stattfindenden Operationen aufgenommen und ihrem Auftrittszeitpunkt entsprechend eingeordnet werden.

- **Waiting State**. Bei Aufruf einer synchronen Methode findet ein weiterer Zustandsübergang statt: Die aufrufende Instanz geht bis zur Beendigung der aufgerufenen Methode (die mit der Rücklieferung von Parametern verbunden sein kann) in den Status *waiting* über. Dieser Zustand ist allerdings nur dann mit einem tatsächlichen Warten des aufrufenden aktiven Objekts verbunden, wenn die aufgerufene Methode ebenfalls einem aktiven Objekt gehört. Der vorhin beschriebene Mechanismus der Pufferung von Nachrichten wird in keinem Fall beeinträchtigt.

  Man beachte übrigens potentielle Deadlock-Gefahren, wie aus der folgenden Abbildung 4 ersichtlich.

4.3 Domains

Die oft riesige Anzahl von Objekten, aus denen sich ein komplexes Software-System zusammensetzt, verlangt nach einem leistungsfähigen Strukturierungskonzept. In DOBOS wird dafür, zusätzlich zu dem bekannten (statischen) Konzept verschachtelter Klassen (*nested classes*), eine (dynamische) Hierarchie von *Domains* bereitgestellt. Jedes Objekt liegt in genau einer Domain und darf Methoden von Objekten aufrufen, die in seiner

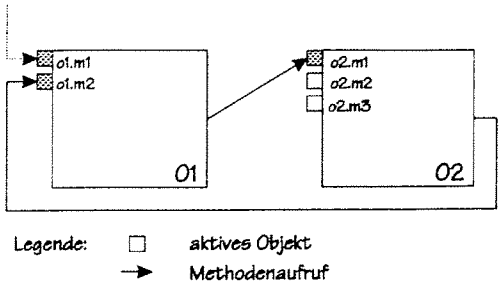


Abbildung 4: Deadlock aktiver Objekte

Domain oder in irgendeiner in der Hierarchie darüberliegenden Domain liegen. Nicht erlaubt sind direkte Methodenaufrufe von Objekten, die in der Hierarchie weiter unten angesiedelt sind. Die folgende Abbildung 5 soll dies verdeutlichen; aktive Objekte werden durch Rechtecke, ihre Domain mit durchbrochener Linienführung, passive Objekte durch Dreiecke dargestellt.

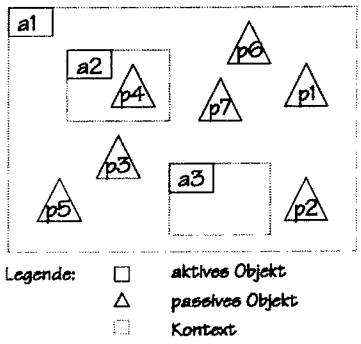


Abbildung 5: Hierarchische Objektstruktur

Die oberste Ebene der Domain-Hierarchie wird als *System Domain* bezeichnet; jede Subdomain ist eindeutig mit einem aktiven Objekt assoziiert. Bei der Instanziierung eines Objekts wird (implizit oder explizit) festgelegt, in welcher (erreichbaren) Domain dieses Objekt angelegt werden soll. Die Instanziierung eines aktiven Objekts bewirkt darüberhinaus die Errichtung der assoziierten Subdomain (im Falle der Instanziierung in der System Domain einer *Top Level Domain*). In bezug auf die Referenz von Objekten unterstützt DOBOS *Object Identifier*, die

- innerhalb jeder Top Level Domain mit allen Subdomains und
- innerhalb der System Domain.

eindeutig sind. Erst dadurch ist es möglich, die vorhin erwähnten Methodenaufrufe innerhalb der Domain-Hierarchie auch praktisch zu realisieren. Am Rande sollte noch erwähnt werden, daß die Verwendung von verschachtelten Klassen in den Domains eine attraktive Alternative zu der expliziten Verwaltung diverser Objektreferenzen darstellt, da hier der implizite *Parent-Operator* zur Verfügung steht.

Die Zuordnung der diversen Objekte auf verschiedene Prozessoren des verteilten Systems erfolgt in DOBOS dadurch, daß jedem Objekt der System Domain ein fixer *node*

of residence zugeteilt werden muß. Auf diese Weise wird auch jeder Top Level Domain ein eindeutiger Prozessorknoten zugewiesen; alle in dem entsprechenden Zweig der Domain-Hierarchie liegenden Objekte werden dann auf diesem Knoten angelegt. Die folgende Abbildung 6 zeigt ein einfaches Beispiel:

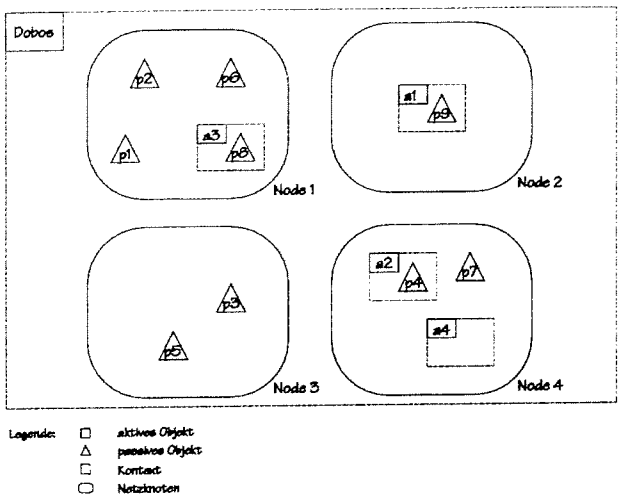


Abbildung 6: DOBOS-*Architektur*

Zu beachten ist, daß das Gesamtsytem trotz der Verteilung der Objekte in der System Domain homogen ist. Instanziierung und Methodenaufrufe sowie Objektreferenzierung werden gegebenenfalls, für den Benutzer transparent, über Knotengrenzen hinweg (*remote*) durchgeführt (dies impliziert natürlich Einschränkungen bezüglich der bei einem Methodenaufruf zulässigen Parameter-Typen). Der remote Methodenaufruf stellt an das Betriebssystem und die untergelagerte Hardware hohe Anforderungen. Aus Gründen der Performance unterscheidet DOBOS deshalb verschiedene *Environments*: Bei der Instanziierung eines Objekts kann angegeben werden, von wo aus Methodenaufrufe an das Objekt kommen können; gegenwärtig werden folgende Environments unterstützt:

- **Common**: Methodenaufrufe von allen Knoten aus möglich.
- **SunOS**: Methodenaufrufe nur vom VTA-Host aus möglich.
- **pSOS**: Methodenaufrufe nur von den VTA-Targets aus möglich.

4.4 Sonstige Eigenschaften

In diesem Abschnitt werden kurz einige weitere, das Konzept von DOBOS betreffende Eigenschaften aufgezählt bzw. gegebenenfalls erläutert. An dieser Stelle muß noch einmal betont werden, daß das Konzept für DOBOS sehr stark durch pragmatische Gesichtspunkte bestimmt wurde; ein in Punkto Implementierung problematisches Feature (wie das Function Overloading) wurde nur dann in das Konzept aufgenommen, wenn kein (Aus)Weg daran vorbeiführte.

- **Mehrfach-Vererbung.** DOBOS läßt die Ableitung einer Klasse von einer oder mehreren Oberklassen zu, wobei Sichtbarkeitsregeln und Zugriffsrechte zu tragen

kommen. Bei einer „public" Ableitung bleiben die Zugriffsattribute der geerbten Klassenelemente erhalten: Instanzvariablen sind innerhalb der Klasse erreichbar, vor willkürlichen Zugriffen von außen bleiben sie geschützt; die Methoden der Oberklassen sind weiterhin zugänglich. Wird die Ableitung hingegen „private" durchgeführt, so ist der Zugriff auf Instanzvariablen der Oberklasse nicht mehr gestattet, sie dürfen nur noch über die Methoden der Oberklasse, die nun jedoch von außen nicht mehr sichtbar sind, angesprochen werden. „Private" Ableitungen dienen somit dem Verbergen von Information gegenüber der Schnittstelle der Oberklasse. Um den Benutzer größtmögliche Freiheit zu gewähren, können zusätzlich Instanzvariablen und Methoden auch selektiv public oder private geerbt werden. Instanzvariablen dürfen nach Belieben in der abgeleiteten Klasse hinzugefügt, Methoden, soferne sie nicht als virtuell deklariert wurden, uneingeschränkt erweitert und redefiniert werden.

- **Polymorphismus.** DOBOS unterstützt das Konzept der virtuellen Funktionen. Wurde eine Methode in einer Oberklasse als virtuell deklariert, so kann sie in davon abgeleiteten Klassen unter demselben Namen und gleichbleibendem Parametertypprofil implementiert werden. Sodann wird beim Methodenaufruf an Objekte, die durch Zeiger oder Referenzen auf die Basisklasse bezeichnet werden, erst zur Laufzeit entschieden, welche Methode (die der Basisklasse oder jene der abgeleiteten Klasse) aktiviert werden soll.

- **Flexible Instanziierungsmechanismen.** In DOBOS kann die Instanziierung eines Objekts auf zwei Arten erfolgen:

  - *Interne Objekte* werden analog zu lokalen Variablen in den Methoden deklariert. Ihre Konstruktoren bwz. Destruktoren werden automatisch bei Eintritt bzw. Verlassen des jeweiligen Programmblocks aufgerufen. Sie gehören implizit jener Domain an, die mit dem aktiven Objekt assoziiert ist, in deren Kontext die Methode tatsächlich ausgeführt wird.
  - *Objekte in Domains* werden durch den Operator *new* erzeugt und durch den Operator *delete* beseitigt. In jedem Fall muß die (mit den Sichtbarkeitsregeln der Domain-Hierarchie verträgliche) Domain des Objekts angegeben werden.

- **Statisches Type Checking.** Das Konzept von DOBOS ist so ausgelegt, daß jegliche Typüberprüfung bereits zur Übersetzungszeit vorgenommen werden kann.

- **Kein Function Overloading.** Aus pragmatischen Gründen unterstützt die momentane Version von DOBOS dieses – an sich bequeme – Feature nicht.

- **Persistente Objekte.** Sehr viele Software-Systeme operieren mit Daten, die eine über einzelne Programmabläufe hinausgehende Lebensdauer haben sollen und daher auf permanente Datenspeicher gesichert werden müssen. Die objektorientierten Modellierungsmöglichkeiten lassen sich aber nur ungenügend oder überhaupt nicht mit Dateien, satzorientierten oder relationalen Datenbanksystemen abdecken. Das komplexe Modell, das im Arbeitsspeicher mit Hilfe der objektorientierten Programmierung aufgebaut wurde, müßte für jede Dateioperation in eine flache Struktur

(Datei, Tabelle, Record) gepreßt und später aus dieser wieder zusammengesetzt werden.

In der Vergangenheit wurden deshalb objektorientierte Datenbanksysteme entwickelt, die derartige Nachteile vermeiden (z.B. Poet). Die zur Zeit verfügbaren Systeme erwiesen sich jedoch für die Verwendung in DOBOS als zu eng an das sequentielle Paradigma („native" C++) gekoppelt. Da die im Zusammenhang mit dem VTA erhobenen Anforderungen in diesem Punkt glücklicherweise relativ gering sind, reicht die Bereitstellung *persistent* (also permanent) deklarierter Objekte in DOBOS für unsere Zwecke bei weitem aus.

- **Gegenseitiger Ausschluß in passiven Objekten.** DOBOS stellt vordefinierte Klassen zur Verfügung, deren Methoden die Funktionalität zum gegenseitigen Ausschluß paralleler Methodenaufrufe bei Zugriffen auf gemeinsame Daten bieten.

- **Einfache Anbindung/Integration existierender (C-)Software-Systeme.** Angesichts der Vielzahl der (public domain oder sonstwie leicht erhältlichen) Software-Tools für UNIX und C bzw. C++ wäre es nicht sehr effizient, die Implementierung des VTAs auf Basis eines Systems zu versuchen, daß die Verwendung derartiger Tools ausschließt. In DOBOS ist es deshalb möglich,

  - bei der Entwicklung der Applikations-Software unter DOBOS gewöhnliche C-Libraries zu verwenden und darüberhinaus
  - aus gewöhnlichen (also „isolierten") UNIX-Prozessen heraus beliebige Methoden von zur Applikation gehörenden DOBOS-Objekten (in der System Domain) aufzurufen.

  Von besonderer Bedeutung für den VTA ist es, daß dadurch Interface-Builder und andere Tools, die für das Design von Multiwindow-Systemen (v.a. unter X-Windows) geeignet sind, verwendet werden können. Die erwähnten Mechanismen gestatten tatsächlich eine nahtlose Anbindung derartiger Callback-basierender Software-Systeme an DOBOS-Applikationen.

5 Ausblick

In dieser Arbeit wurde einleitend das (sequentielle) objektorientierte Paradigma vorgestellt und dargelegt, wieso es speziell für den Entwurf der System-Software für das verteilte Monitoring-System VTA so gut geeignet ist. Ausgehend von dem entsprechenden Anforderungsprofil wurde das Konzept eines leistungsfähigen Distributed Object-Based Operating System DOBOS vorgestellt, welches die Welt der objektorientierten Programmierung mit der der Parallelität homogen vereinigt und trotzdem offen gegenüber „normaler" Software und darüberhinaus einfach und effizient zu implementieren ist.

Literatur

[1] Agha, G., „Concurrent Object-Oriented Programming“, Communications of the ACM, September 1990.

[2] Cargill, T. „CRC Cards“, The C++ Report, September 1991.

[3] Hontsch T., „Source-Level Monitoring verteilter Echtzeitsysteme“, Diplomarbeit TU Wien, Dezember 1991.

[4] Jensen D., Northcutt J.D., „Alpha: A Non-proprietary Operating System for Mission-critical Real-time Distributed Systems“, Proceedings IEEE Workshop on Experimental Distributed Systems, Oktober 1990.

[5] Klasek J., „Dynamische Software- und Hardware-Instrumentierung“, Diplomarbeit TU Wien, vor. Juni 1993.

[6] Mercer C., Tokuda H., „The ARTS Real-Time Object Model“, Proceedings Real-Time Systems Symposium, Dezember 1990.

[7] Schmid U., Stöckler S., „A Versatile Monitoring System for Distributed Real-Time Systems“, Proceedings of the SAFECOMP'92, Oktober 1992.

[8] Stöckler S., „Goldmine: Generic Object-oriented Language for Defining Monitors, Intervals and Events“, Technical Report 183/1-27, April 1992.

[9] Wegner P., „Concepts and Paradigms of Object-Oriented Programming“, Proceedings OOPSLA-89, Oktober 1989.

[10] Yonezawa A., „ABCL An Object-Oriented Concurrent System“, MIT Press, September 1989.