DOBOS: EIN DISTRIBUTED OBJECT-BASED OPERATING SYSTEM FÜR SOFT REAL-TIME SYSTEMS

U. SCHMID, W. KASTNER*
Technische Universität Wien

Zusammenfassung

We present some important features of our *Distributed Object-Based Operating System* DOBOS designed for high-performance, heterogeneous, distributed soft real-time systems. It provides concurrent objects, location transpacency, persistence, and dynamic classes in the static type checking framework of C++, thus preserving the usability of existing software and development/debugging environments. DOBOS is built upon a runtime kernel layered on top of traditional operating systems, and a native C++-compiler in conjunction with some particular class libraries, forming a powerful yet relatively easy-to-provide concurrent object-oriented environment.

1. EINLEITUNG

Objektorientierung und verteilte Systeme sind aus der modernen Datenverarbeitung nicht mehr wegzudenken. Die Integration dieser beiden "Welten" ist zwar noch Gegenstand aktiver Forschung (siehe z.B. [2], [1], [4], [7]); es gibt aber bereits einige "kommerziell verfügbare" Systeme (etwa NEXT's *distributed objects*) und sogar einschlägige Standardisierungsaktivitäten — man denke nur an die OMA (*Object Management Architecture*) der OMG (*Object Management Group*).
Etwas anders ist hier die Situation der Echtzeit-Datenverarbeitung. *Hard real-time systems* erfordern bekanntlich Lösungsansätze (etwa für das Scheduling), die sich grundlegend von traditionellen Verfahren unterscheiden — deren Realisierung wird durch die Forderung nach objektorientierten Konzepten nicht gerade einfacher. Die einschlägige Forschung steht ziemlich am Anfang; vgl. [7] oder [4]. Im Gegensatz dazu sind die aus der traditionellen Datenverarbeitung stammenden Paradigmen für *soft real-time systems* durchaus brauchbar, wenngleich auch mit Einschränkungen und Modifikationen. Letztendliches Ziel solcher Bestrebungen sollte die Bereitstellung einer Programmierumgebung sein, die dem Entwickler all den Komfort der Objektorientierung bietet, den er —mit einigem Recht— erwarten kann. Das allmähliche Auftauchen von C++-Entwicklungsumgebungen für *embedded systems* oder gar Lösungen wie [3] belegen den konkreten Bedarf recht deutlich. Die den letztgenannten Systemen üblicherweise zugrundeliegende, relativ "naive" Übertragung herkömmlicher Paradigmen (etwa von OMA-Standards) auf den Bereich der Echtzeitsysteme ist allerdings aus Performance-Gründen eher wenig befriedigend.

Die vorliegende Arbeit ist einer praktisch verwendbaren Programmierumgebung für high-performance *distributed soft real-time systems* gewidmet. Konkret wurde uns die Nichtverfügbarkeit eines derartigen Systems sehr unangenehm bewußt, als wir im Zuge eines Forschungsprojektes (*Versatile Timing Analyzer*[1] *VTA for Distributed Real-Time Systems*, siehe z.B. [8]) vor dem Problem standen, den mit objektorientierten Methoden entworfenen, sehr komplexen VTA auf einer verteilten Architektur bestehend aus Sun Workstation(s) unter Solaris (UNIX) und 68030

[1]Gefördert vom Österreichischen Fonds zur Förderung der Wissenschaftlichen Forschung, Projekt-Nr. P8390.

VME-CPUs unter ISI/SUC's pSOS — implementieren zu müssen. Unsere konkrete Applikation setzt —neben den Standardkonzepten der Objektorientierung wie Mehrfachvererbung, Polymorphismus, usw.— unter anderem folgendes voraus (siehe [5]):

- *Parallelität*: Aus Performance-Gründen kann auf Parallelverarbeitung (inklusive Multitasking) nicht verzichtet werden.
- *Verteilte Objekte*: Objekte müssen transparent von jedem Rechner im verteilten System aus manipulierbar sein.
- *Dynamische Klassen*: Trotz *static type checking* muß es möglich sein, zur Übersetzungszeit (noch) nicht bekannte Klassen zu verwenden.
- *Persistente Objekte*: Wir benötigen wenigstens elementare Mechanismen zur Verwaltung von Objekten, die ihren Status auch über die Programmlaufzeit hinaus behalten können.
- *Übliche Programmiersprache*: Es ist vom Einarbeitungsaufwand der Projektmitarbeiter her gesehen kaum vertretbar, irgend eine "exotische" Programmiersprache einzuführen.
- *Einfache Anbindung (existierender) C/C++-Software*: Angesichts der unabsehbaren Fülle wertvoller Software sollte eine einfache (und trotzdem homogene) Integration derselben möglich sein.

All diese Features müssen natürlich unter dem Aspekt der maximalen Performance bereitgestellt werden. Entscheidende Bedeutung kommt schließlich auch der Verfügbarkeit einer leistungsfähigen Entwicklungsumgebung zu.

Eine Evaluation existierender Systeme machte bald klar, daß an einer —ursprünglich nicht vorgesehenen— Eigenentwicklung kein Weg vorbeiführen würde: Inadäquate Modellierungsmittel, fehlende bzw. "exotische" sprachliche Mechanismen, Probleme mit der Verteilung auf unsere heterogene Hardware-Architektur, unzureichende Integration existierende Software und nicht zuletzt auch mangelnde Performance ließen sowohl "kommerziell verfügbare" Systeme als auch Research-Prototypen rasch ausscheiden. Wir waren daher gezwungen, uns selbst um die Bereitstellung eines geeigneten Betriebssystems zu kümmern. Das Ergebnis der bisherigen Arbeiten daran (siehe auch [6], [5]) ist das Detailkonzept eines *Distributed Object-Based Operating Systems* DOBOS, das eine leistungsfähige Basis für die Entwicklung von *distributed soft real-time systems* darstellt.
DOBOS selbst basiert auf einem relativ einfachen Laufzeitsystem, das auf Solaris und pSOS^{+m} aufsetzt und die oben erwähnten Features auf Systemebene bereitstellt. Darauf aufbauend sorgen ein (Standard-)Compiler für C++ und einige notwendige Klassen-Bibliotheken dafür, daß diese Funktionalitäten dem Applikations-Programmierer auch tatsächlich zur Verfügung stehen. Ein wesentlicher Vorteil unseres Ansatzes besteht darin, daß sehr leicht existierende Entwicklungs/Debugging-Umgebungen eingesetzt werden können.

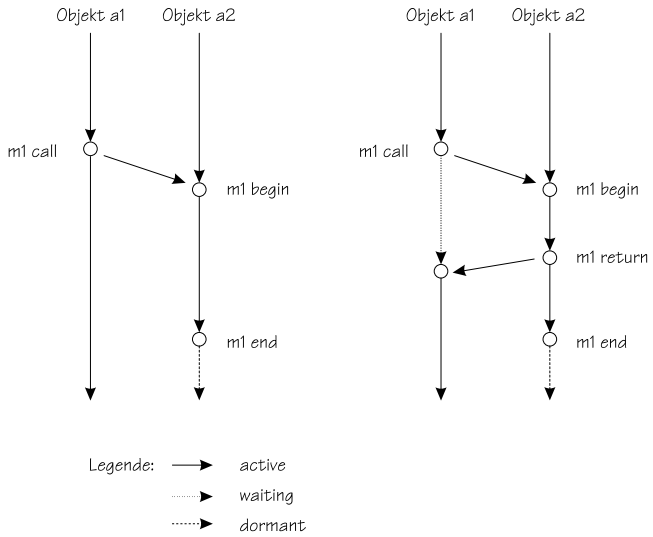
2. OBJEKTE IN DOBOS

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.

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 herkömmliche (schnelle) Funktionsaufrufe realisiert, was auch die geforderte einfach Anbindung/Integration herkömmlicher Software sehr vereinfacht.

Obwohl fast alle zur Zeit existierenden objektorientierten Systeme für nicht parallele Software gedacht sind, passen Objekte *a priori* gut in die Konzepte der parallelen Programmierung. Ihre logische Autonomie formt sie zu Einheiten, die im Prinzip auch gleichzeitig exekutiert werden können. In DOBOS wird Parallelität konkret mit Hilfe von *aktiven Objekten* modelliert. Ein aktives Objekt verfügt über ein eigenständiges inhärentes „Leben" (*thread of control*) 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. Aktive Objekte in DOBOS verwenden ihre passiven Gegenstücke 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.

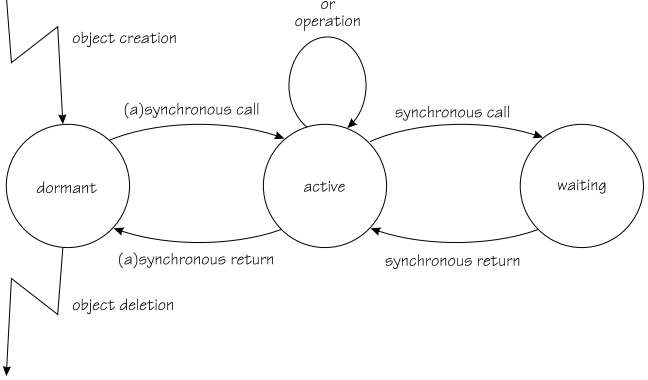
Im Falle des Aufrufs einer *asynchronen* Methode ist der Auftraggeber nicht gezwungen, nichtstuend auf deren Termination zu warten, sondern kann währenddessen seinen eigenen Aufgaben nachgehen. Der Grad der Parallelität des Gesamtsystems steigt somit mit der Anzahl der asynchronen Methodenaufrufe. Dieser Mechanismus ist natürlich nur in Situationen von Interesse, in denen die aufrufende Instanz keine Rücklieferung von Resultaten benötigt (Triggermethoden):



Die obenstehende Abbildung 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.
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.

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):

- **Dormant State**. Nach seiner Erzeugung (Instantiierung) befindet sich ein aktives Objekt

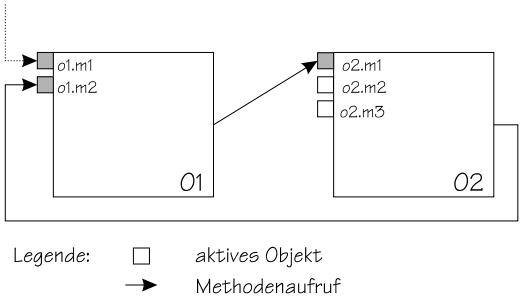


solange im Zustand *dormant*, bis ein Methodenaufruf erfolgt oder das Objekt wieder zerstört wird.

- **Active State**. Wird eine Methode eines aktiven Objekts aufgerufen, so geht das Objekt vom Zustand *dormant* in den Zustand *active* über. Nun werden die Methoden-spezifischen 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 (Methodenauslösungen) 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.

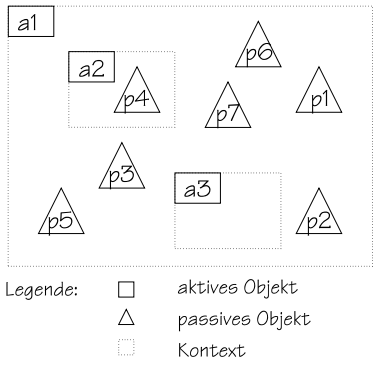
  Man beachte übrigens potentielle Deadlock-Gefahren, wie aus der folgenden Abbildung ersichtlich:



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

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

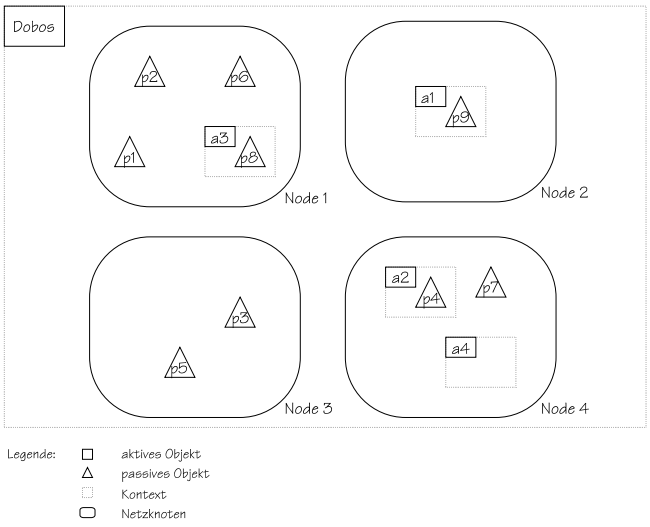


Die oberste Ebene der Domain-Hierarchie wird als *System Domain* bezeichnet; jede Subdomain ist eindeutig mit einem aktiven Objekt assoziiert. Bei der Instanziierung eines Objekts wird (implizit oder explizit) festgelegt, in welcher (erreichbaren) Domain dieses Objekt angelegt werden soll. Die Instanziierung 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.

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 umseitige Abbildung zeigt ein einfaches Beispiel.



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.
- **Solaris**: Methodenaufrufe nur von Solaris-Rechnern aus möglich.
- **pSOS**: Methodenaufrufe nur von pSOS^{+m}-Rechnern aus möglich.

4. WEITERE FEATURES

In diesem Abschnitt werden kurz einige weitere, das Konzept von DOBOS betreffende Eigenschaften erläutert. Zunächst einmal ist festzuhalten, daß die Objektorientierung in DOBOS völlig von der verwendeten Programmiersprache C++ stammt. Damit stehen automatisch die bekannten Mechanismen/Sprachmittel für Mehrfachvererbung mit Sichtbarkeitsregeln und Zugriffsrechten, Polymorphismus mittels virtuellen Funktionen, voll statische Typ-Überprüfung usw. zur Verfügung. Alle DOBOS-spezifischen Sprachkonstrukte werden durch eine standardmäßige Erweiterung von C++ (Overloading, Klassenbibliotheken) bereitgestellt. Einige der wichtigsten Features sind:

- **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.
- **Persistente Objekte.** 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 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. Obgleich solche Systeme teilweise sehr mächtige Funktionen bieten, ziehen wir einen anderen Ansatz vor: In DOBOS ist es möglich, Objekte *persistent* zu erzeugen und optional in logischen Einheiten zusammenzufassen. Im Gegensatz zu objektorientierten Datenbanken sind hier keine expliziten Schreib/Lese-Funktionen zur Rettung/Wiederherstellung eines persistenten Objektes notwendig; DOBOS kümmert sich implizit darum, daß alle persistenten Objekte nach dem Wiedereinschalten in dem zuletzt gültigen Zustand wiederhergestellt werden.

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 äußerst unvernünftig, ein System einzuführen, das deren Verwendung ausschließt. In DOBOS ist es deshalb möglich,

  - bei der Entwicklung der Applikations-Software unter DOBOS gewöhnliche C/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 konkret für unseren VTA ist es, daß dadurch Interface-Builder, Visualisierungssysteme 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.

Mangelnder Platz verbietet es uns, auf weitere Details oder gar auf die Implementierung einzugehen. Es ist uns aber ein Anliegen, abschließend all den vielen Mitarbeitern im Projekt VTA, insbesondere J. Klasek sowie St. Stöckler und H. Haberstroh für ihr Engagement in Sachen DOBOS zu danken.

Literatur

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

[2] Cahill, V., Baker, S., Horn, C., Starovic, G., *The Amadeus GRT – Generic Runtime Support for Distributed Persistent Programming*, Proceedings OOPSLA-93, 1993.

[3] DSET Corporation, *Distributed Systems Generator (DSG) System Overview*, Califon, New Jersey, May 1993.

[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] Schmid, U., Kastner, W., *DOBOS — Konzept eines Distributed Object-Based Operating Systems*, TU Wien, Technical Report Inst. für Automation Nr. 183/1-40, Dezember 1993.

[6] Klasek, J., *Dynamische Software- und Hardware-Instrumentierung*, Diplomarbeit TU Wien, Inst. für Automation, voraussichtlich Juni 1994.

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

[8] Schmid, U., *Monitoring Distributed Real-Time Systems*, to appear in Real-Time Systems, 1994.