OSCAR - Eine Systemarchitektur für den autonomen, mobilen Roboter MARVIN*

Stefan Blum
Lehrstuhl für Realzeit-Computersysteme
Technische Universit"at M"unchen
Stefan.Blum@rcs.ei.tum.de

Zusammenfassung

Die struktierte Entwicklung verteilter Software für autonome Systeme stellt nicht zu unterschätzende Ansprüche an das Design unter Gesichtspunkten des Software-Engineering. Im vorliegenden Beitrag wird die am Lehrstuhl für Realzeit-Computersysteme der Technischen Universität München entwickelte Systemarchitektur OSCAR (Operating System for the Control of Autonomous Robots) beschrieben. OSCAR stellt dabei als ein auf dem Middle-Ware Standard CORBA basierendes, abstraktes Betriebssystem die Entwicklungs- und Zielplattform des autonomen, mobilen Roboters MARVIN dar, der z.Z. für die Exploration von Innenräumen eingesetzt wird. Der Beitrag geht auf die interne Struktur von OSCAR näher ein und erläutert verschiedene Kommunikationskonzepte in Hinblick auf praktische Handhabung und Performance.

1 Einleitung

Informationsverarbeitung auf autonomen, mobilen Robotern bedeutet im weitesten Sinne die Erfassung der Umwelt durch Sensoren, die Auswertung der anfallenden Sensordaten unter teilweiser Berücksichtigung vorhandenen Wissens und die Generierung sinnvoller Aktionen in Hinblick auf eine gegebene Aufgabenstellung. In vielen Systemen wird die zu leistende Informationsverarbeitung auf eine Software-Architektur abgebildet [7, 1, 11, 10, 3]. Arkin [2] nennt als Bewertungskriterien solcher Architekturen

- die Unterstützung für parallele Verarbeitung,
- die Abbildbarkeit auf Roboter sowie die vorhandenen Rechnereinheiten,
- die Abbildbarkeit bezüglich der Aufgabenstellungen,

*Die vorliegende Arbeit wurde im Rahmen des Projekts *Exploration von Innenr"aumen mit optischen Sensoren auf mehreren, aufgabengerechten Abstraktionsebenen* (F"orderungsnummer Fa109/14-1) von der Deutschen Forschungsgemeinschaft (DFG) gef"ordert.

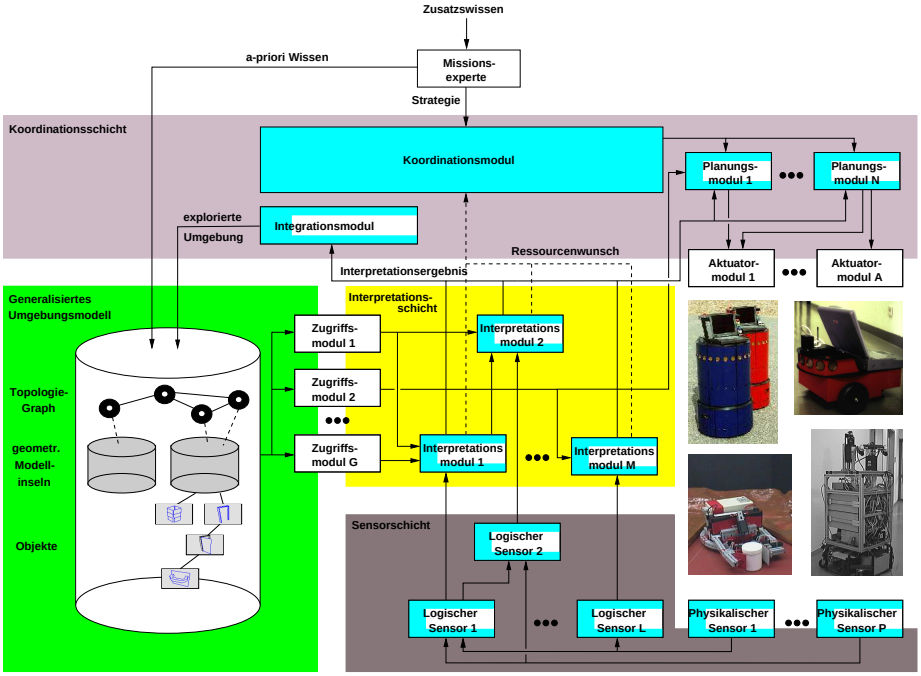


Abbildung 1: Systemstruktur

- die Unterstützung für modulares Design,
- die Robustheit gegen Ausfälle von Komponenten,
- die zur Entwicklung zur Verfügung gestellte Infrastruktur,
- die Flexiblität und Rekonfigurierbarkeit während der Laufzeit und
- die erreichte Performance bezüglich einer gegebenen Aufgabenstellung.

Auf der autonomen, mobilen Plattform MARVIN [5] wird die eigens entwickelte Systemarchitektur OSCAR (Operating System for the Control of Autonomous Robots) [4] eingesetzt, deren Design sich eng an die oben genannten Kriterien anlehnt. OSCAR setzt als Kommunikationsmechanismus die CORBA-Technologie [9] ein. Dieser Beitrag legt den Schwerpunkt auf interne Strukturen der Systemarchitektur, deren Aufbau im folgenden Abschnitt beschrieben ist. Abschnitt 3 stellt OSCAR aus Sicht des Applikationsentwicklers dar und behandelt Fragestellungen bei der Erstellung von Applikationen unter Gesichtspunkten des Software-Engineering. In Abschnitt 4 werden Ergebnisse von Performance-Analysen vorgestellt und zwei Einsatzszenarien beschrieben. Einen Ausblick auf künftige Arbeiten gibt Abschnitt 5.

2 Aufbau der Systemarchitektur

Die Architektur OSCAR besteht neben der intrinsischen Infrastruktur aus einer Menge hierarchisch anordenbarer Module, die bezüglich ihrer Schnittstellen, der übertragenen Datentypen und der Konfiguration standardisiert sind und im Rahmen eines Szenarios bestimmte Verarbeitungsaufgaben übernehmen. Dabei liegt der Schwerpunkt der Klassifikation der Module weniger auf dem Aspekt der Aufgabenplanung und -zerlegung (*task level*) [11, 10], sondern vielmehr bei deren Stellung bei Perzeptionsaufgaben. OSCAR ist dabei so ausgelegt, daß der anfallende Sensordatenstrom flexibel, d.h. abhängig von den Notwendigkeiten der Aufgabenstellung und der aktuellen Missionsphase entsprechend bearbeitet werden kann. Abb. 1 zeigt einen Überblick des Aufbaus von OSCAR. Die Einordnung der Module ist in [6] näher beschrieben. Als Modelldatenbank für Interpretationsprozesse sowie zur Speicherung explorierter Information auf topologischer und geometrischer Ebene wird *GEM (Generalized Environmental Model)* [13] eingesetzt. Zur Koordination und Ablaufsteuerung dienen z.Z. je nach Szenario unterschiedliche Koordinationsmodule.

2.1 Kommunikation

Prinzipiell existieren verschiedene Möglichkeiten Daten zwischen den Modulen zu übertragen. Das OSCAR-Streaming-Konzept unterscheidet zwischen *cue flow* und *configuration flow*.

Cue flow In Architekturen, die in robotischen Systemen eingesetzt werden, verläuft die Hauptströmungsrichtung gemessen an der Datenmenge von den physikalischen Sensoren bis zu Integrations-, Koordinations- und Planungsmodulen[1]. Charakteristisch ist dabei aufgrund der durch die Perzeption vorgenommene Abstraktion und Interpretation der Daten meist eine stete Abnahme der Datenaufkommens und der Übertragungshäufigkeit. Die zu übertragenden Daten (maschinenlesbare Sensorrohdaten, Merkmale oder Hypothesen) werden im Folgenden als *cues* bezeichnet, der Datenstrom als *cue flow*. Da Sensordaten von verschiedenen Sensoren bzw. Modulen teilweise asynchron anfallen, und CORBA-Aufrufe v.a. über Prozeß- bzw. Rechnergrenzen teuer sind (vgl. Abs. 4.1), werden *cues* generell nur nach Bedarf übertragen. Ein *cue* stellt dabei die kleinste übertragbare Einheit dar und besteht aus einer Sequenz an Zeitstempeln[2] und einem generischen Datentyp, der je nach Bedarf überladen wird (siehe Abs. 2.3). *Cues* können auf unterschiedlicher Art und Weise einzeln oder im Block übertragen werden: Bei einem *pull* Aufruf fordert das hierarchisch höhere Modul Daten an, bei einem *push* Aufruf liefert das hierarchisch niedriger angesiedelte Modul die Daten selbständig. Daneben existieren nicht-blockierende Triggeraufrufe (*prefetching*) und die Möglichkeit, auf *cues* mit bestimmter zeitlicher Indizierung zuzugreifen.

Configuration flow Der in die umgekehrte Richtung verlaufenden Datenfluß wird als *configuration flow* bezeichnet, unter den zwei unterschiedliche Formen

[1] Aktuatormodule werden über den *configuration flow* angesprochen.

[2] Die Zeitstempelsequenz beinhaltet immer den Zeitpunkt der Generierung des Datums; bei datenfusionierenden Modulen kann die Sequenz dynamisch erweitert werden.

fallen: Einerseits kann das Koordinationsmodul (siehe Abs. 2.4) entsprechend der aktuellen Missionsphase ein Modul aktivieren, deaktivieren, den Zeittakt variieren oder die Modulverschaltung dynamisch ändern[3]. Andererseits können Daten in dieser Richtung von Modul zu Modul z.B. zum Zweck der Kommandierung oder für Prädiktionen übermittelt werden.

2.2 Prinzipieller Aufbau eines Verarbeitungsmodul

Abb. 2 zeigt den Aufbau des universellen OSCAR-Verarbeitungsmoduls. Durch die Verwendung von CORBA (vgl. Abs. 2.3) ist ein Modul im Sinne einer atomaren Einheit transparent auf verschiedenen Rechnern eines verteilten Systems lauffähig[4]. Prinzipiell wird dabei ein Modul auf einen Prozeß abgebildet, der aus zwei Threads besteht. Eine Thread ist der für jedes Modul identische *Rahmen*, der den im zweiten Thread laufenden *Verarbeitungskern* kapselt. Als reaktiver Server integriert der Rahmen dabei verschiedene Schnittstellen, und wickelt sämtliche Kommunikation für Konfiguration und Datenaustausch ab. Grundsätzlich werden alle Ein- und Ausgangsdaten des *cue flow* in konfigurierbaren Ringpuffer zwischengespeichert, die einerseits ein zeitlich begrenzte Bewahrung der Historie, andererseits effektive Blockzugriffe ermöglichen. Dabei wird je ein Ringpuffer einem Datenkanal zugeordnet.

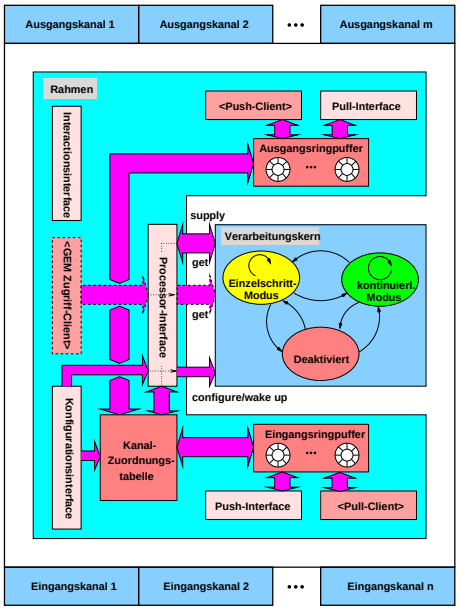


Abbildung 2: Verarbeitungsmodul

Modul aus Anwendersicht Der Verarbeitungskern ist als eigentlicher Applikationsträger ein Client, der über das *Processor-Interface* mit dem Rahmen kommuniziert. Dabei stehen verschiedene Funktionen zum Zugriff auf die Ein- und Ausgangskanäle sowie auf Konfigurationsdaten bereit. Ein Modul befindet sich immer in einem der drei Zustände *continuous mode*, *single-step mode* oder *deactivated*. Dabei erlaubt der *continuous mode* einen vollständig asynchronen Betrieb, wobei der sich zyklisch wiederholende Verarbeitungsalgorithmus in einem festen Zeittakt gefahren wird, soweit dies die Auslastung des PCs erlaubt. Der *single-step mode* hingegen repräsentiert die synchrone Abarbeitung des Verarbeitungsalgorithmus, d.h. das Modul arbeitet nur dann, wenn eine Trigger-Bedingung erfüllt ist. Wird ein Modul in der aktuellen Missionsphase nicht benötigt, so wird es in den Zustand *deactivated* geschaltet. Das Modul

[3] Dies ist z.B. beim Ausfall und dem damit verbundenen Neustart eines Moduls notwendig.

[4] Bei physikalischen Sensoren sowie Aktuator-Modulen ist diese Transparenz durch die Hardware-Kopplung nicht gegeben.

aus Anwendersicht wird im Folgenden *logisches Modul* genannt.

Implementierung Tatsächlich erlaubt die Implementierung der Rahmen-Infrastruktur die Unterbringung verschiedener Verarbeitungskerne *(„multi core“)* unter folgenden Nebenbedingungen[5]:

- Alle Kerne müssen auf dem gleichen Rechner lauffähig sein.
- Maximal ein Verarbeitungskern befindet sich im *continuous mode*, das dann alle anderen Kerne triggert[6].
- Im *multi core* Modul sind intern nur entweder *pull*- oder *push*-Aufrufe gestattet.
- Bezüglich des *cue flow* sind Verzweigung und Zusammenführung gemeinsam nicht erlaubt.

Die Einhaltung dieser Bedingungen erlaubt die Implementierung aller Verarbeitungskerne in gleichem Thread, was sich bezüglich der Performance der Datenübertragung bezahlt macht (vgl. Abs. 4.1). Der modulintern synchrone Betrieb ermöglicht zudem die Einsparung eines Ringpuffers[7]. Nachteilig wirkt sich allerdings der Ausfall eines Kerns aus, der dann des Ausfall und Neustart des gesamten Moduls bedingt.

2.3 CORBA

Die *Common Object Request Broker Architecture (CORBA)* ist ein von der *Object Management Group (OMG)* [9] entwickelter Middle-Ware Standard, der in vielen Bereichen eingesetzt wird. OSCAR nutzt zur Kommunikation seiner intrinsischen Infrastruktur ausschließlich CORBA 2.3. und setzt dabei auf die Implementierung ORBacus4.0 [8] auf. CORBA ist prinzipiell plattform-, betriebssystem- und programmiersprachenunabhängig, da es das Client-Server-Paradigma auf ein höheres Abstraktionsniveau stellt. CORBA bietet Lokalitätstransparenz, d.h. für einen Client ist es durch die Verwendung sog. Objekt-Referenzen abgesehen von Performance-Aspekten gleichgültig, ob sich das Server-Objekt lokal und auf einem anderen Rechner befindet.

```
typedef unsigned long TimestampType;
typedef sequence<TimestampType> TimestampSeq;
struct CueType
    {
    TimestampSeq mTimeS;
    any mCue;
    };
```

Abbildung 3: Überladbarer *cue* Datentyp

[5] Hierbei werden die einzelnen Kerne in der Kanalzuordnungstabelle des Rahmens registriert, Konfigurationsaufrufe sowie Eingangs- und Ausgangskanäle werden entsprechend für den Applikationsentwickler transparent auf die Kerne umgeleitet.

[6] Befinden sich alle Kerne im *single-step mode*, so wird das gesamte Modul von außen getriggert.

[7] Bei Ringpuffergröße eins werden die Daten direkt, d.h. ohne Kopiervorgang übergeben. Bei der Datenübertragung wird zwischen internem und externem Datenfluß unterschieden, wobei der externe Datenfluß zusätzlich Daten in den Ausgangsringpuffern des Moduls bereitstellt.

Der Programmierer ist vollends von der Implementierung von lowlevel-Protokollen auf Netzwerkebene entlastet. Der damit einhergehende Overhead verschafft CORBA oft den Ruf, als Kommunikationsmechanismus in der Robotik aufgrund von vorliegenden Echtzeitbedingungen und geringen Latenzzeitanforderungen ungeeignet zu sein. Dieser Aspekt läßt sich durch die geeignete Wahl der Kommunikationsmechanismen relativieren (siehe Abs. 4.1).

Als Spezifikation für Datentypen und Schnittstellen dient die C/C++-ähnliche, aber rein deklarative OMG IDL *(Interface Definition Language)*, mit der sämtliche in OSCAR übertragenen Datentypen und Komponentenschnittstellen definiert sind. Abb. 3 zeigt die Definition des universellen `CueType`, der durch die Verwendung des generischen Datentyps `any` die Übertragung beliebiger IDL-Datentypen erlaubt. Abb. 4 zeigt drei für OSCAR definierte Datentypen, die auch in Abs. 4.1 zur Performance-Messung benutzt werden. Die Verwendung des Datentyps `any` bedingt einen Overhead, da der sog. *typecode* mitübertragen werden muß [9]. Um auf unterschiedlichen Architekturen kompatibel zu sein, werden alle zu übertragenden Datentypen mit Ausnahme des opaquen Typs `octet` in das *CDR*[8]-Format gewandelt. Insgesamt nutzt OSCAR allerdings nur einen kleinen Teil der CORBA-Merkmale, die u.a. auch standardisierte Dienste beinhalten.

```
typedef boolean BooleanType;

struct LineSeg3DType {
    double mSx,mSy,mSz;
    double mEx,mEy,mEz; };
typedef sequence<LineSeg3DType> LineSeg3DSeq;

typedef sequence<octet> ImageType;
struct GvImageType {
    short mDx,mDy;
    ImageType mImage; };
```

Abbildung 4: *Cues* für versch. Anwendungen

2.4 Weitere Infrastruktur

OSCAR besitzt eine universelle Boot-Routine, die alle Module entsprechend einer Konfigurationsdatei auf den dabei angegebenen Rechnern startet und verschaltet. Dabei melden sich alle Module bei der *Registry* an, die dann auf jeden genutzen Rechner einen Monitor installiert, der die Prozesse der Module periodisch auf Funktionalität prüft. Fällt ein Modul wegen eines Fehlers aus, so gelangen alle Module in den sog. *suspend mode*. Das ausgefallene Modul kann wahlweise manuell oder automatisch neugestartet oder ausgetauscht werden. Per Definition besitzt das jeweilige Koordinationsmodul einer Anwendung als einziges die Möglichkeit, Module zu konfigurieren. Dies ist notwendig, um eventuelle Konflikte vorzubeugen. Die OSCAR-Module können jedoch diesbezüglich Ressourcenwünsche an das Koordinationsmodul äußern.

3 OSCAR als Framework

Die Verwendung der Architektur OSCAR unterstützt den Entwickler einer verteilten Applikation, indem sie ein Framework bereitstellt, das die Systemstruk-

[8] Common Data Representation

tur im Großen definiert, die Modulentwicklung aber weitgehend von der Implementierung der Architektur kapselt. Für die Erstellung eines logischen Moduls stehen Skeletons bereit, die u.a. mit dem Verarbeitungsalgorithmus, der Zahl der Ein- und Ausgangskanäle mit den entsprechenden *cue* Datentypen, der Menge und Art der konfigurierbaren Elemente und Aktionen[9] und mit dem zu vergebenden Modulnamen gefüllt werden müssen. Für die Implementierung existiert ein verbindlicher Leitfaden für die Kodierung *(Coding convention)*. Die vorgeschriebene Verwendung von Kommunikationsmakros führt dazu, daß der Entwickler nur marginal mit der CORBA-Technologie in Berührung kommt. Die Integration bereits vorhandener Algorithmen ist somit leicht zu bewerkstelligen.

Durch den Einsatz des Versionsverwaltungssystems CVS ist OSCAR grundsätzlich eine Multientwicklerplattform. Neben den einzelnen Dateibäumen der Modul-Entwickler *(sandbox)* existiert eine Produktversion, in Software nur nach eingehendem Test gelangt. Zum Testen einzelner Module und für Integrationstests sowie für das Debugging stehen verschiedene Möglichkeiten bereit:

- Die Module können mit *stand-alone*-Betrieb einerseits ohne OSCAR-Infrastruktur getestet werden, indem sie *Stubs* verwenden. Andererseits muß jedes Modul einen sog. *stub mode* implementieren, um bei Verwendung der OSCAR-Infrastruktur den Einzeltest eines mit ihm verbundenen Moduls zu ermöglichen, ohne dabei vom eigenen Verarbeitungsalgorithmus abzuhängen. Dadurch ist auch die Möglichkeit zur inkrementellen Integration gegeben.

- Generische Datenlogger können an Ein- und Ausgangskanäle gehängt werden, die den *cue flow* aufzeichnen und zu einem späteren Zeitpunkt wiedergegeben können. Die Datenloggern ermöglichen zudem ein vereinfachtes Post-Mortem-Debugging.

- Definierte Makros erlauben Debugausgaben parallel laufender Module wahlweise im eigenen oder in ein gemeinsames Fenster, was eine Rekonstruktion des zeitlichen Ablaufes ermöglicht.

- Jedes logisches Modul kann auf Wunsch auch in einem eigenen Gnu-Debugger gestartet werden[10].

Das Framework OSCAR kann aber keinen genauen und verbindlichen Leitfaden zur Granularität logischer Module geben. Im Vorgrund steht hierbei der Aspekt der Wiederverwendbarkeit eines Moduls im Sinne einer Komponente, zu feine Granularität jedoch erhöht die Abhängigkeit der Module untereinander und bedingt größeren Kommunikationsoverhead.

[9] Z.B. können bei der Aktivierung des Moduls bestimmte Ressourcen belegt werden.

[10] Das zu testende Modul kann dabei ggf. aus seiner *multi core* Umgebung herausgelöst werden.

```
typedef sequence<octet> OctSeq;
interface OB_Bench {
void BOctet(in long len, out OctSeq dat);
void BString(in long len,out string dat);
void BAny(in long len, out any dat);
};
```

Architektur	Celeron@400MHz
Betriebssystem	Linux 2.0
Compiler	gcc-2.95.2 -O1
Ethernet	10 Mbit
CORBA-Implement.	Orbacus-4.0
Concurrency Model	`reactive`

Abbildung 5: IDL-File zur Erzeugung des CORBA-Benchmarks (links), Technische Daten der Messung (rechts)

4 Ergebnisse

4.1 Performance

Die Leistungsfähigkeit einer Software-Architektur hängt bei verteilten Perzeptionsaufgaben bei einer gegebenen Implementierung der Verarbeitungsalgorithmen im Wesentlichen von folgenden Aspekten ab[11]:

- Scheduling und Sequencing
- Lastverteilung und damit Auslastung der nutzbaren Rechnerressourcen
- Latenzzeiten bei der Datenübertragung

Für OSCAR ist der zugrundeliegende Kommunikationsmechanismus entscheidend, da Scheduling und Sequencing und Lastverteilt frei konfiguriert werden können. Es folgt daher zunächst eine Performance-Analyse der CORBA-Implementierung, dann gemessene Richtzeiten die Übertragung von unterschiedliche OSCAR-Datentypen.

Messung des Zeitbedarfs von CORBA-Aufrufen Zur Messung des Zeitbedarfs eines CORBA-Aufrufs werden Datenpakete der variablen Größe s (100 Byte bis 32000 Byte) von einen Server zu einem Client übertragen[12]. Der Client mißt dabei die zum Aufruf benötigte Zeit t über 1000 Aufrufe gemittelt. Die effektiv nutzbare Datengröße ist bei jedem Aufruf gleich, der Aufruf `BOctet` überträgt das Paket mit dem Datentyp `octet`, beim Aufruf `BString` wird das Datenpaket zusätzlich der CDR-Schicht unterworfen. Die Verwendung des generischen Datentyps `any` erfordert den Zusatz des *typecode*.

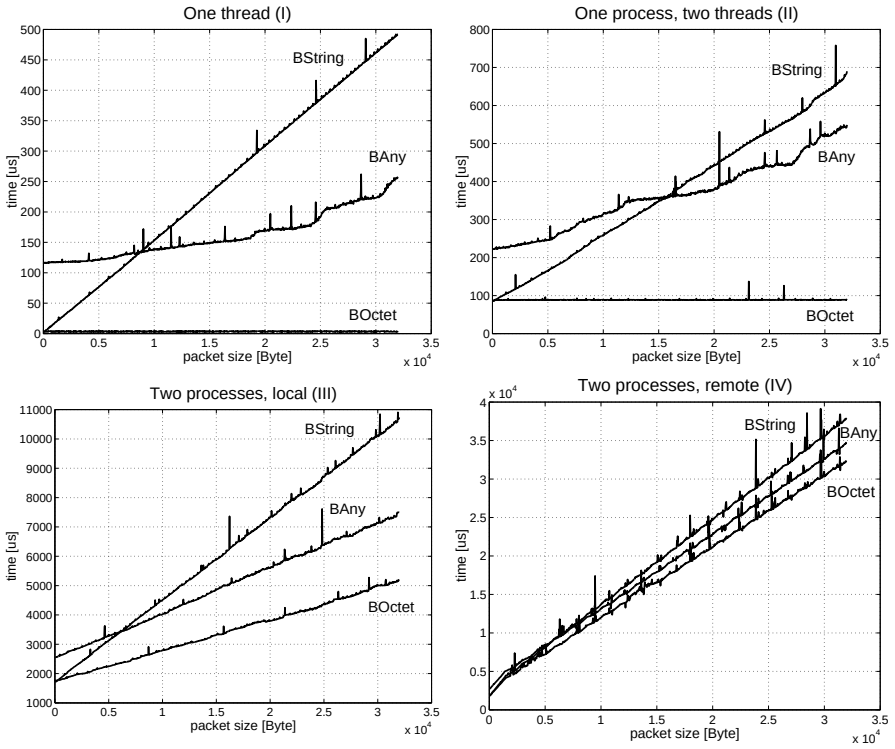
Insgesamt vier Konstellationen für Client (C) und Server (S) sind denkbar:

(I)	C und S befinden sich in einem Thread[13].
(II)	C und S befinden sich im gleichen Prozeß in unterschiedlichen Threads.
(III)	C und S laufen als eigenständige Prozesse auf einem PC.
(IV)	C und S laufen auf unterschiedlichen, über Ethernet verbundenen PCs.

[11] Verhaltensbasierte Architekturen müssen beispielsweise möglichst kleine Latenz- und Reaktionszeiten aufweisen [3], für Anwendungen in der Bildverarbeitung kann die Maxime ein möglichst hoher Durchsatz sein, der z.B. durch ein Pipeline-Verfahren realisiert wird.

[12] IDL-Datei und technische Daten siehe Abb. 5

[13] Dabei ruft der Client die Methode des Implementierungsobjektes des Servers direkt auf.



	I		II		III		IV	
$[\mu s]$	a	b	a	b	a	b	a	b
oct	3.46	0.0000	88.49	0.0000	1716.53	0.1068	2740.56	0.9278
str	0.10	0.0154	75.06	0.0184	1717.19	0.2797	2629.12	1.1047
any	99.78	0.0039	209.44	0.0096	2518.83	0.1545	3369.35	0.9775

Abbildung 6: Performance-Messungen unterschiedlicher CORBA-Aufrufe

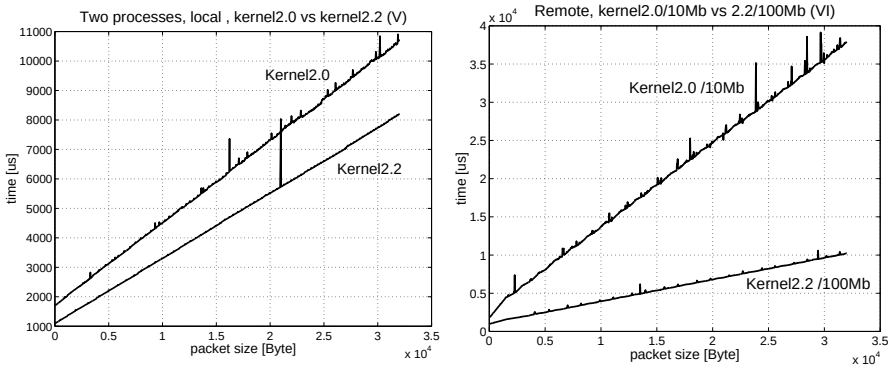
Bei allen Messungen wird ein lineares Meßmodell[14] $t(s) = a + b \cdot s$ angenommen, wobei die Parameter a und b durch Regression bestimmt wurden. Die Ergebnisse sind in Abb. 6 dargestellt. Bei den Messungen (IV) und (VI)[15], die eine Datenübertragung über das Ethernet erforderten, wurden größere Peaks gefiltert, die auf sonstige Netzwerkbelastung zurückgeführt werden[16]. Auffällig sind jedoch bei allen Messungen kleinere Peaks, deren Ursache zum jetzigen Zeitpunkt aber noch nicht analysiert wurde.

Eine Beschleunigung der Übertragung der Daten läßt sich durch den Einsatz des Linux-Kernels 2.2 und eines 100 Mbit-Ethernets erreichen, wie der Vergleichung der `BString`-Messung zeigt (vgl. Abb. 7). Beides steht aber im Augenblick auf der mobilen Plattform MARVIN noch nicht zur Verfügung.

[14] Es wird dabei davon ausgegengen, daß ein CORBA-Aufruf eine konstante Grundzeit a und eine von der Paketgröße s abhängige Zeit $b \cdot s$ benötigt.

[15] siehe unten.

[16] Bei den Messung wurde auf weitgehend definierte Bedingungen geachtet, allerdings wurde das Netzwerk in der Normalkonfiguration betrieben, d.h. keine Netzwerkdämonen deaktiviert.



$[\mu s]$			a	b
BString	V	(Lokal, Kernel 2.0)	1717.19	0.2797
BString	V	(Lokal, Kernel 2.2)	1096.79	0.2211
BString	VI	(Remote, Kernel 2.0, 10Mbit)	2629.12	1.1047
BString	VI	(Remote, Kernel 2.2, 100Mbit)	1095.08	0.2855

Abbildung 7: Meßverläufe bei Verwendung Linux-Kernel 2.2 (links) und zusä tzlich 100 Mbit-Ethernet (rechts)

Insgesamt zeigt sich, daß die Übertragung mittels des opaquen Datentyps `octet` naturgegeben am schnellsten ist. Je nach Konstellation übertrifft die Performance der Übertragung des generischen Typs `any` die des Typs `string` im Bereich der Paketgröße von ca. 200 Byte bis ca. 1800 Byte. Allerdings zeigt sich z.T. bei `any` ein nur noch schwach linearer Verlauf der Meßkurve.

Performance von OSCAR Der weitaus am häufigsten genutzte Datentransportmechanismus *cue flow* dient als Benchmark für die Architektur OSCAR. Dabei wurde exemplarisch die benötigte Zeit einer Übertragung der drei in Abs. 2.3 vorgestellte Datentypen für die drei Konstellation *multi core*[17], lokal auf einem PC befindliche Module und für Module auf unterschiedlichen PCs gemessen. Alle beteiligten Module wurden dabei im *single step mode* betrieben. Abb. 8 zeigt über 100 Messungen gemittelte Zeit pro Einzelübertragung.

Datentyp	Anwendung	Datengröße[18]	*multi core*	lokal	remote
`BooleanType`	Signal	1	86	1736	1820
`LineSegment3DSeq`	100 3D-Linien	4800	254	5782	10544
`GvImageType`	8bit PAL-Bild	442368	7851	140157	497219

Abbildung 8: OSCAR-Benchmarks in $[\mu s]$

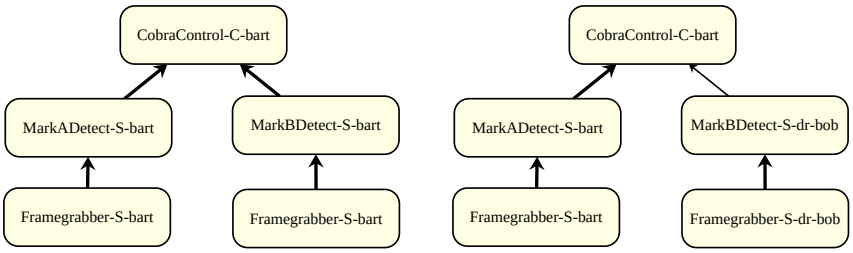


Abbildung 9: Modulverschaltung für nicht verteilte (links) und verteilte (rechts) Konfiguration

4.2 Anwendungsbeispiele

Integration von Bildverarbeitungsmodulen und Ablaufsteuerung In einer Applikation für einen Roboterarm mit fünf Freiheitsgraden wird OSCAR für die Steuerung und Integration von Modulen eingesetzt. Der Roboterarm selbst kann aufgrund mangelhafter interner Sensorik und Aktorik keine definierten und reproduzierbaren Bewegungen ausführen und muß daher mittels einer landmarkengestützten Auswertung zweier Kamerabilder gesteuert werden. Dabei werden insgesamt fünf logische OSCAR-Module eingesetzt: zwei Framegrabber-Module, je ein Modul zur Detektion unterschiedlicher Marken (`MarkADetect` und `MarkBDetect`) sowie ein Koordinationsmodul für die Ablaufsteuerung (`CobraControl`). Die Applikation wurde in zwei unterschiedlichen Konfigurationen realisiert: Bei der Verwendung nur eines PCs können die fünf Module zu nur insgesamt einem *multi core* Modul integriert werden[19] (siehe Abb. 9 links). Werden zwei PCs mit jeweils einem Framegrabber eingesetzt, so werden insgesamt zwei *multi core* Module gebildet. (siehe Abb. 9 rechts). Während bei der ersten Konfiguration die Bildakquisition und -auswertung der beiden Kameras sequenziell abläuft, wird bei der zweiten Konfiguration der *prefetch* Mechanismus zur Datenübertragung von `MarkBDetect` zu `CobraControl` verwendet, der die parallele, wenn auch nicht ganz zeitsynchrone Bildauswertung ermöglicht und damit die Verarbeitungszeit fast halbiert. **Exploration eines Ganges mit MARVIN** Der autonome, mobile Roboter MARVIN dient zur Zeit als Experimentierplattform für die Exploration eines Ganges. Ziel ist es, mittels einer Explorationsfahrt den Roboter im Rahmen späterer Missionsfahrten zu befähigen, bei Vorgabe einer Raumnummer die entsprechende Tür gezielt und auf dem kürztesten Wege anzufahren[6]. Die OSCAR-Applikation läuft dabei auf den beiden Onboard-PCs `marvin` und `scooter`[20] und besteht aus insgesamt zwölf logischen Modulen, die sich aufgrund der in Abs. 2.2 angegebenen Bedingungen zu insgesamt fünf *multi core* Modulen zusammenfassen lassen (vgl. Abb. 10). Das durch einen Zustandsautomaten implementierte

[17] Ringpufferlänge 1 bei internen Datenfluß, siehe Abs. 2.2

[18] nutzbare Datengröße in Byte, ohne Zeitstempelsequenz.

[19] In der Abbildung sind modulinterne *cue flow* Verbindungen fett gezeichnet, die Nomenklatur eines Modul setzt sich aus Modulnamen, Betriebsart (*single-step mode* (S) oder *continuous mode* (C)) und Rechnernamen zusammen.

[20] Celeron©400MHz.

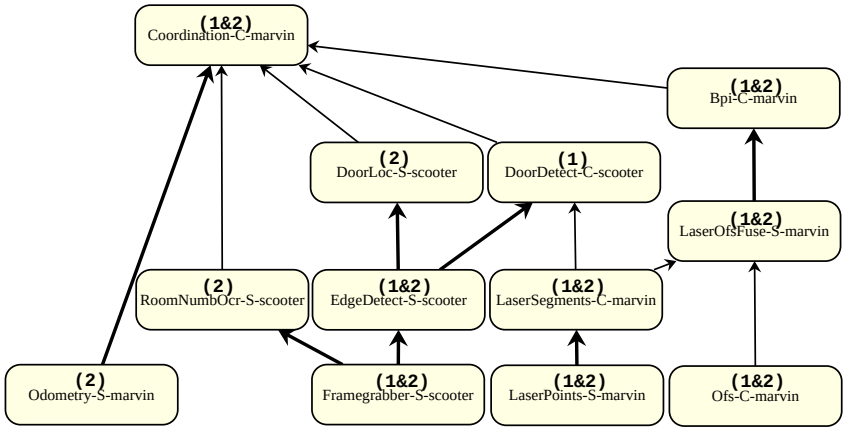


Abbildung 10: Für das Szenario „Exploration eines Ganges“ eingesetzte Module

Koordinationsmodul aktiviert je nach aktueller Phase der Exploration die dafür notwendigen Module; innerhalb einer Phase sorgt es für die zeitlich abgestimmte Triggerung von aktiven Einzelmodulen. Das Explorationsszenario ist mit zwei Missionsphasen realisiert. In Phase 1 verfolgt MARVIN das Verhalten *Wandfolgen mit gleichzeitiger Türsuche*. Phase 2 (Verhalten *Türidentifikation*) besteht aus einem festen Ablauf aus einem Rangiermanöver, um die Plattform in eine Position zur genaueren relativen Türlokalisation zu bringen, der Türlokalisation selbst, einer Fahrt vor die Tür und dem Lesen des Türschildes mit einem OCR-Modul[21]. Das Koordinationsmodul nimmt dabei auch eine Konfiguration einzelner Module vor, beispielsweise wird der Framegrabber zwischen Viertel- und Halbbildmodus umgeschaltet. Die Funktionalität der beteiligten Module ist genauer in [6] beschrieben.

5 Ausblick

OSCAR ist als entwicklerfreundliche Plattform für die Implementierung verteilter Applikationen für autonome Systeme einer ständigen Weiterentwicklung unterworfen. In künftigen Arbeiten werden u.a. folgende Aspekte im Vordergrund stehen:

Erweiterte Testmöglichkeiten für den Entwickler Im Augenblick ist ein Simulationssystem im Entstehen, das mittels eines „Drehbuches“ das reproduzierbare Testen von Verarbeitungsalgorihmen in den Modulen erlauben wird.
Weitgehend automatisierte Konfiguration Die in Abs. 2.4 beschriebene Konfigurationsdatei soll dabei schrittweise durch die Möglichkeit der Aufgabendefinition auf abstrakterer Ebene ersetzt werden. Dies betrifft Aspekte einerseits der automatischen Zusammenstellung von *multi core* Modulen, der Verteilung der eingesetzten Module auf die vorhandenen Rechnerressourcen *(load balan-*

[21] Die entsprechend aktiven Module sind in Abb. 10 mit `(1)`, `(2)` oder `(1&2)` gekennzeichnet.

cing) und der Generierung von Aktionsketten, andererseits der symbolischen Repräsentation einzusetzender Module und Ressourcen.

Generisches Koordinationsmodul Die Ablaufsteuerung u.a. der im letzten Abschnitt beschriebenen Applikationen wurde bisher durch spezialisierte Koordinationsmodule implementiert, die meist einen unflexiblen Zustandsautomaten beinhalten. Geplant ist hier die Integration eines generischen Koordinationsmoduls, das einen verhaltensbasierten Ansatz ähnlich [12] realisiert.

Unterstützung für Pipeline-Betrieb Im Augenblick ist durch die Betriebsarten *single-step mode* und *continuous mode* noch kein echter durchsatzerhöhender Pipeline-Betrieb möglich. Dies soll durch geeignete Synchronisationsmaßnahmen der Module untereinander realisiert werden.

Performancesteigerung bei der Datenübertragung Durch die Ersetzung des generischen CORBA-Datentyps `any` bei der Übertragung des *cue flow* besteht noch Potential für Performancesteigerungen. Dies soll durch ein proprietäres Verfahren ersetzt werden, das eine Serialisierung der Datentypen in `octet`-Sequenzen vornimmt. Dies ist dann möglich, wenn der Einsatz von OSCAR auf die gleiche Rechnerarchitektur beschränkt bleibt.

Echtzeitaspekte OSCAR setzt als abstraktes Betriebssystem auf Linux auf, das per se nicht echtzeitfähig ist. Allerdings liegt die für Explorationsaufgaben relevante Zeitauflösung im Millisekundenbereich, so daß es für bisherige Anwendungen nicht notwendig war, auf der Abstraktionsebene von OSCAR Echtzeitbetriebssysteme wie QNX, RTEMS oder RT-Linux einzusetzen. Geplant ist für OSCAR-Module allerdings eine Zeitüberwachung mit der Möglichkeit, weiche und harte Deadlines zu definieren, die somit im ersten Fall eine dynamische Lastverteilung, dynamisches Sequencing und Scheduling realisieren, im zweiten Fall zu einem Stoppen der Plattform im Sinne eines *fail-safe* führen[22].

Literatur

[1] J. S. Albus and A. M. Meystel. A Reference Model Architecture for Design and Implementation of Intelligent Control in Large and Complex Systems. *Int. J. of Intelligent Control and Systems*, 1(1):15–30, 1996.

[2] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[3] T. Bergener and A. Steinhage. An Architecture for Behavioral Organization using Dynamical Systems. In C. Wilke, S. Altmeyer, and T. Martinetz, editors, *Third German Workshop on Artificial Life*. Verlag Harri Deutsch, 1998.

[4] S. Blum. OSCAR-Homepage. http://www.rcs.ei.tum.de/research/rovi/oscar.

[5] S. Blum, D. Burschka, C. Eberst, T. Einsele, A. Hauck, N. O. Stöffler, and G. Färber. Autonome Exploration von Innenräumen mit der Multisensorik-Plattform MARVIN. In *Autonome Mobile Systeme*, Informatik aktuell, pages 138–147. Springer-Verlag, 1998.

[6] S. Blum, T. Einsele, A. Hauck, N. O. Stöffler, G. Färber, T. Schmitt, C. Zierl, and B. Radig. Eine konfigurierbare Systemarchitektur zur geometrisch-topologischen Exploration von Innenräumen. In *Autonome Mobile Systeme*, Informatik aktuell. Springer-Verlag, Nov. 1999.

[7] R. A. Brooks. A Robust Layered Control System For a Mobile Robot. *IEEE Journal of Robitics and Automatisation*, RA-2, No. 1:14–23, 1986.

[22] Dieses Verhalten ist muß dann allerdings doch über einen echtzeitfähigen Mechanismus implementiert werden.

[8] Object Oriented Concepts. *Orbacus*. http://www.ooc.com/ob/.

[9] OMG. CORBA/IIOP 2.3 specification. http://www.omg.org/corba, 1998.

[10] C. Schlegel and R. Wörz. Der Softwarerahmen SmartSoft zur Implementierung sensomotorischer Systeme. In *Autonome Mobile Systeme*, Informatik aktuell, pages 208–217. Springer-Verlag, 1998.

[11] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, and J. O'Sullivan. A Layered Architecture for Office Delivery Robots. In *First International Conference on Autonomous Agents*, Feb. 1998.

[12] A. Steinhage and T. Bergener. Dynamical Systems for the Behavioral Organization of an Anthropomorphic Mobile Robot. In *From animals to animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior*, pages 147–152. MIT Press, 1998.

[13] N. O. Stöffler, A. Hauck, and G. Färber. Ein geometrisch-symbolisches Umgebungsmodell zur Unterstützung verschiedener Perzeptionsaufgaben autonomer, mobiler Systeme. In G. Schmidt and F. Freyberger, editors, *Autonome Mobile Systeme*, Informatik aktuell, pages 108–117. Springer-Verlag, 1996.