Technical report

Nr.: FIN-001-2013

Analyse und Vergleich von Frameworks für die Implementierung von Robotikanwendungen

Sebastian Zug, Thomas Poltrock, Felix Penzlin, Christoph Walter, Nico Hochgeschwender

EOS, EUK, Fraunhofer IFF, Hochschule Bonn-Rhein-Sieg

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-001-2013

Analyse und Vergleich von Frameworks für die Implementierung von Robotikanwendungen

Sebastian Zug, Thomas Poltrock, Felix Penzlin, Christoph Walter, Nico Hochgeschwender

EOS, EUK, Fraunhofer IFF, Hochschule Bonn-Rhein-Sieg

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Sebastian Zug
Postfach 4120
39016 Magdeburg
E-Mail: sebastian.zug@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 01.02.2013

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Kurzfassung

Robotersysteme werden im täglichen Einsatz und noch mehr im Forschungskontext für höchst unterschiedliche Aufgabenstellungen in verschiedensten Umgebungen eingesetzt oder untersucht. Aus der resultierenden Heterogenität der Hardware stellt sich mit jedem Projekt die Frage, ob ein entsprechendes Framework verwendet werden soll, das die Entwicklungsarbeit vereinfacht. Der interdisziplinären Aufgabenstellung entsprechend existiert eine Vielzahl von Entwicklungstools, die in den Designprozess einbezogen werden können. Die Spannweite reicht dabei von kleinen Programmen, die den Zugriff auf bestimmte (Sensor-)Schnittstellen erlauben bis hin zu ausgewachsenen Entwicklungsumgebungen, die auf komplexen Frameworks aufsetzen. Diese bieten häufig Konzepte und Methoden zur sensorischen Datenerfassung, zum Datenaustausch, zur Simulation u.Ä.. Die häufig als *Robotik-Entwicklungsumgebung* oder *Robotik-Middleware* deklarierten Implementierungen bieten für diese Basisaufgaben geeignete Abstraktionen und Konzepte. Zudem stellen die Implementierungen über die Einbindung entsprechender Bibliotheken den Zugriff auf typische Anwendungen aus dem Robotikkontext (Lokalisation, Messdatenfusion, Trajektorienplanung etc.) bereit. Daneben unterscheiden sich die verschiedenen Middleware-Applikationen in allgemeingültigen Kategorien, wie zum Beispiel den unterstützten Programmiersprachen, den Kommunikationsparadigmen, den resultierenden Codegrößen einer Anwendung oder den Echtzeiteigenschaften.

Wenn der Entwickler die Frage nach einer für sein System geeigneten Robotik-Middleware stellt, erschwert die angesprochene Vielfältigkeit die Entscheidung. Vor diesem Hintergrund gibt die vorliegende Arbeit einen Überblick über relevante Middlewareimplementierungen und analysiert deren Möglichkeiten und Grenzen. Mit einer ausführlichen Gegenüberstellung anhand konkreter Kriterien (unterstützte Betriebssysteme und Hardware, Echtzeitfähigkeit, Erweiterbarkeit usw.) wird eine objektive Vergleichbarkeit gewährleistet.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Motivation

Mit dem technologischen Fortschritt, steigender Performance der Rechner, immer leistungsfähigeren Sensoren usw. erweitert sich das Einsatzspektrum von Roboterapplikationen zusehends. Neben etablierten Anwendungen in der Industrie als stationärer Manipulator oder fahrerlose Transportplattform halten Roboter nun auch in alltäglichen Szenarien Einzug, wie Such- und Rettungsdiensten, Gebäudeüberwachungsaufgaben, militärischen Anwendungen u.a.. Die Aufzählung macht die unterschiedlichen Einsatzumgebungen (Luftraum, privater Haushalt, (Unter-)Wasser, Straßenverkehr usw.) und die Verschiedenartigkeit der Aufgaben für ein solches System (Pfadplanung, Kollisionsvermeidung, Interaktion mit Gegenständen und Menschen) deutlich. Die Aufzählung der Heterogenität ließe sich auch im Hinblick auf die Hardwarekonfiguration erweitern. Aus der Aufgabenstellung ergeben sich zudem nicht-funktionale Eingenschaften wie Fehlertoleranz, Echtzeitfähigkeit usw.. Damit sind Robotikanwendungen insgesamt ausgesprochen verschiedenartige Applikationen, deren ausgeprägte Individualität den Entwicklungsprozess erschwert.

Robotiksysteme basieren heute auf einem verteilten System, das Module zur Umgebungserfassung mit Datenverarbeitungsknoten und Schnittstellen zu Treiberbausteinen und Antrieben kombiniert. Der Entwicklungsprozess der Software für ein solches System umfasst neben der eigentlichen Anwendungsentwicklung zwei periphere Aufgaben: Zum einen muss die Kommunikation zwischen den Komponenten des Systems sichergestellt und zum anderen die Interfaces zur Hardware bereitgestellt werden. Um den Entwicklungsaufwand an dieser Stelle zu reduzieren, muss die beschriebene Vielfältigkeit gekapselt und der Zugriff auf eine uniforme Schnittstelle abgebildet werden. Erst damit ist eine Austauschbarkeit und Wiederverwendbarkeit von Komponenten umsetzbar. Dieser Schritt wird üblicherweise durch eine Kommunikationsmiddleware und einen Hardwareabstraktionlayer realisiert, so dass eine transparente Anwendungsentwicklung möglich wird.

In Anbetracht der angedeuteten Heterogenität der Robotikanwendungen argumentierten Nesnas u. a. [Nes+06] in ihrer Motivation zum NASA Robotikframework CLARAty noch im Jahr 2006:

> Within the NASA robotics community, and possibly within the research community, the majority of robotic software is designed and built from scratch

> for each new robot. To date, it may have been easier and more cost effective to do so.
>
> However, as robotic software gets more complicated and the time and effort to build reliable software increases, it becomes more important to develop reusable robotic software.

Dieser Aufgabe stellt sich seit mindestens einem Jahrzehnt eine breite Forschergemeinschaft. Daraus entstanden eine Fülle von verschieden orientierten Frameworks für die Entwicklung von Robotikanwendungen. Nunmehr steht der Anwender aber vor der Aufgabe aus der Vielzahl an verfügbaren Implementierungen das für seine konkrete Anwendung geeignetste zu finden. Entsprechend muss der Entwickler nach dem Abstecken des Szenarienkontexts, wie zum Beispiel der beteiligten Hardwarekomponenten, den nötigen Algorithmen, den erforderlichen Kommunikationseigenschaften usw., aufwendige Recherchen betreiben, um eine weitgehende Übereinstimmung sicherzustellen. Daneben dient die Arbeit insbesondere im Kontext mit vergleichbaren Aufstellungen als Überblick zur historischen Entwicklung über verschiedene Rahmenwerke.

Damit ergeben sich für die vorliegende Arbeit drei Leser-/Anwendergruppen:

- Entwickler, die einen konzentrierten Überblick über die Möglichkeiten und Grenzen der Robotiksoftware suchen,
- Anwendungsentwickler im Robotik-/Automatisierungskontext, die ein für ihr Projekt geeignetes Framework auswählen wollen und
- Programmierer zukünftiger Frameworks, die Lücken bestehender Arbeiten füllen wollen.

Vor diesem Hintergrund liegt die Zielstellung der vorliegenden Arbeit auf der Analyse und Kategorisierung der meistverbreiteten Frameworks. Dabei wird in Erweiterung der in Kapitel 2 aufgeführten Arbeiten:

- eine an den Fortschritt angepasste Auswahl an Frameworks getroffen,
- die Kategorisierung überarbeitet und damit
- eine detaillierte und aussagekräftige Gegenüberstellung in tabellarischer Form ermöglicht.

Die Auswahl der Frameworks für Robotik-Applikationen richtete sich dabei danach, inwieweit der Entwicklungsprozess über eine Hardwareabstraktionsschicht, eine Kommunikationsmiddleware und entsprechende Bibliotheksschnittstellen unterstützt wird. Zudem wurde vorausgesetzt, dass Tools zur Vereinfachung und Beschleunigung der Entwicklung, zum Beispiel für die Visualisierung von Daten, die Analyse der Systemkonfiguration, Mechanismen zum Logging, Simulation usw. bereitstehen. Nicht betrachtet

wurden folglich Entwicklungsplattformen, die auf einzelne Aspekt zugeschnitten sind und beispielsweise die Simulation in den Vordergrund stellen, wie WebBots [Mic98] oder USARSim [Car+07].

1.2 Vergleichbare Arbeiten

Arbeiten, die Entwicklungsframeworks für Robotikanwendungen vergleichen, existieren insbesondere von Elkady und Sobh [ES12] sowie Kramer und Scheutz [KS07]. Letztgenannter entwickelt ausgehend von einer umfangreichen Vorstellung von insgesamt 11 Frameworks ein Konzept, das eine übergreifende Vergleichbarkeit gewährleisten soll. Dazu entwickelten die Autoren einen 32 Einträge umfassenden Satz von Kriterien, der u.a. Architekturkonzepte, unterstützte Programmiersprachen und Simulatoreigenschaften beinhaltet, und ordnen diese in vier Hauptkategorien *Spezification*, *Platform support*, *Infrastructure* und *Implementation* ein. Besonderes Augenmerk lag dabei auf den unterstützten robotikspezifischen Bibliotheken, die sehr differenziert aufgeführt wurden. In einem zweiten Schritt wird die Anwendbarkeit im Hinblick auf Ressourcenanforderungen, Dokumentation, die Einbindung von Visualisierungswerkzeugen usw. untersucht. Die Ergebnisse wurden auf eine Metrik abgebildet und entsprechende Kennzahlen für jedes Framework berechnet, die Anhaltspunkte bei der Auswahl geben sollen. Obwohl die Arbeit lediglich 4 Jahre alt ist, fehlen wichtige aktuelle Entwicklungen. Kritisch muss angemerkt werden, dass die gewählte Darstellungsform weniger geeignet ist, um für den Leser schnell die Relevanz einer Entwicklungsumgebung ausgehend von einem spezifischen Anwendungsfall erkennbar zu machen.

Elkady und Sobh [ES12] bieten mit ihrer Arbeit aus dem Jahr 2012 einen aktuelleren Überblick und beziehen gegenwärtige Entwicklungen in den Vergleich ein. Allerdings ist die Beschreibung auf wenige Sätze pro Framwork beschränkt. Anhand der 13 durch die Autoren definierten Attribute (*System Model*, *Control Model*, *Fault Tolerance*, etc.) erfolgt dann aber in tabellarischer Form eine detaillierte Gegenüberstellung. Diese lässt aber verschiedene wichtige Aspekte, die die Entscheidungsgrundlage für den Anwendungsentwickler bilden, außer Acht. So fehlt beispielsweise eine Antwort auf die wichtige Frage nach der unterstützten Hardware und Sensorik.

Verschiedene Arbeiten wie beispielsweise Mohamed, Al-Jaroodi und Jawhar [MAJJ08] stellen die Kommunikationsmiddleware in den Mittelpunkt der Untersuchung und vergleichen verschiedene Frameworks in dieser Hinsicht. Die Frameworks werden in diesem Zusammenhang als Robotikmiddleware bezeichnet. Als Kriterien für die tabellarische Gegenüberstellung werden die Kommunikationsparadigmen, Discoverymechanismen und Fehlertoleranz herausgearbeitet. Mit der Beschränkung auf Kommunikationsaspekte trifft diese Darstellung die Ausrichtung der vorliegenden Arbeit nur teilweise.

Makarenko, Brooks und Kaupp [MBK07] untersuchen in ihrer Arbeit das Zusammenspiel der Komponenten eines Frameworks vor dem Hintergrund einer Wiederverwend-

barkeit der Implementierungen. Dabei machen die Autoren deutlich, dass die Basisfunktionalität der einzelnen Frameworks ihrer Meinung nach auf drei Kategorien abgebildet werden können: Hardwaretreiber und Verarbeitungsalgorithmen, Kommunikationsschnittstellen sowie dem Bindeglied zwischen diesen Komponenten. Dieser Terminologie folgt auch die vorliegende Arbeit und sieht in den vorzustellenden Frameworks insbesondere die integrative Funktion, die unterschiedliche Hardware, robotikspezifische Algorithmen, Kommunikationsmechanismen aber auch Simulationsumgebungen und Entwicklungstools zusammenführt.

Eine aktuelle und differenzierte Darstellung der Programmierframeworks für Roboter bieten Iñigo-Blasco u. a. [IB+12]. Insbesondere bemühen sie sich den Bogen zu den Multi-Agentensystemen und den dafür relevanten Bibliotheken oder Frameworks zu spannen. Dabei machen die Autoren deutlich, dass einige der robotikspezifischen Frameworks auf etablierten Multi-Agenten Ansätzen aufbauen. Ausgehend von einem Versuch die Kernkompetenzen dieser Forschungsrichtungen abzugrenzen werden in der Arbeit Validierungskriterien für Robotikframeworks hergeleitet und eine große Zahl aktueller Implementierungen daran gemessen.

In einer anderen Richtung bewegt sich die Arbeit von Shakhimardanov, Hochgeschwender und Kraetzschmar [SHK10]. Die wesentlichen Programmierabstraktionen bzw. Primitive für verschiedene Komponenten-basierte Robotersoftwareframeworks sind nach Auffassung der Autoren sehr ähnlich, d.h. Interoperabilität ist im Prinzip (konzeptionell) möglich. Ein Beispiel: die meisten Frameworks haben die Notation eines Datenports um asynchronen Datenfluss zwischen Komponenten zu modellieren. Die Arbeit untersucht auf der Basis einer Komponentenanalyse die Möglichkeiten einer Interaktionen zwischen verschiedenen Frameworks. Damit erfolgt zwangsläufig ein Vergleich der elementaren Funktionen und Konzepte, der aber auf einer abstrakten Ebene geführt wird.

2 Relevante Frameworks

In den letzten Jahren hat sich im Bereich der Robotik eine Entwicklung von Robotern, die für die Erfüllung spezifischer Aufgaben als eine kompakte Einheit entworfen wurden, hin zu autonomen modularen Systemen vollzogen. Kennzeichnend für diese „neue“ Robotergeneration ist die Kooperation und Interaktion zwischen heterogenen Hard- und Software-Modulen. Um die Integration dieser Komponenten in ein Gesamtsystem zu ermöglichen und dabei den Software-Entwicklungsprozess zu vereinfachen und zu beschleunigen, sind viele Frameworks mit unterschliedlichen Schwerpunkten entwickelt worden. Diese gehen zum Beispiel Integrationsprobleme wie Interoperabilität, Kommunikation und Konfiguration, die mit einem modularen System einhergehen, an.

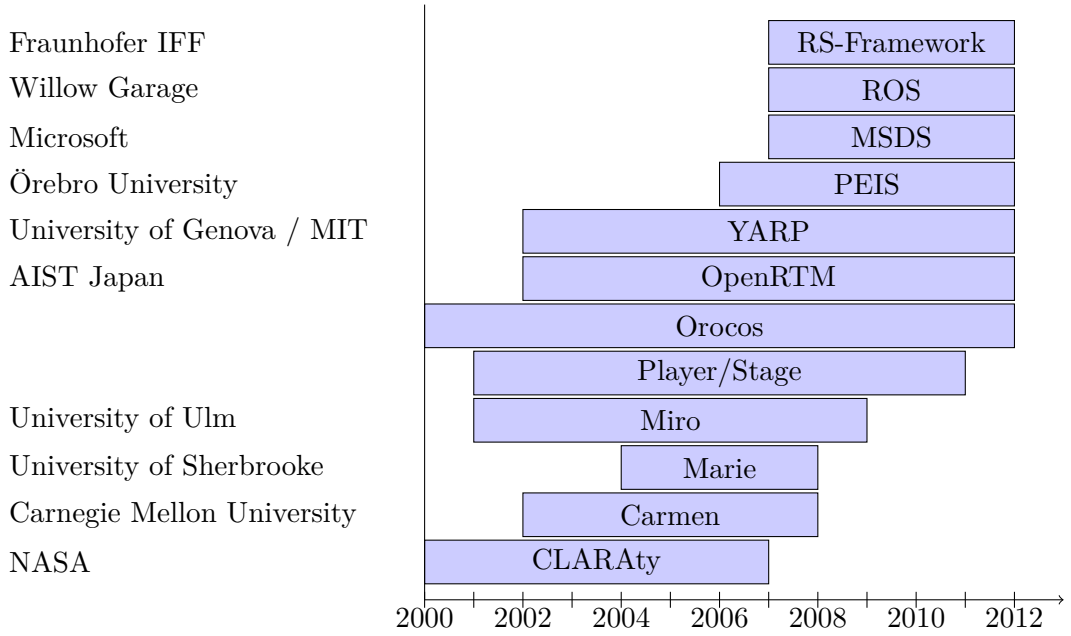

Abbildung 2.1: Chronologische Einordnung der in dieser Arbeit aufgeführten Frameworks mit zugehörigem (Haupt-)Entwicklerteam (sofern kein Namen genannt wird, stehen mehrere Einrichtungen hinter dem Projekt)

In diesem Kapitel werden 12 ausgewählte Konzepte vorgestellt. Dabei wurden nicht nur aktuelle Frameworks aufgeführt, sondern auch solche, die seit einiger Zeit nicht mehr

weiterentwickelt werden. Da diese aber entscheidende Impulse zur weiteren Entwicklung der Robotikframeworks gaben und damit wichtig für einen umfassenden Überblick sind, sollen diese nicht unerwähnt bleiben. Abbildung 2.1 illustriert die Chronologie der betrachteten Frameworks

Aus der Beschränkung auf 12 Frameworks ergibt sich zwingend, dass mit der vorliegenden Aufstellung keine Vollständigkeit angestrebt werden kann. Vor dem Hintergrund, dass eine robotikorientierte Analyse der auf SourceForge gehosteten Projekte im Jahr 2009 allein 539 relevante Implementierungen zählte, macht die Vielfältigkeit der Ansätze und Rahmenwerke deutlich [Kum]. Im Folgenden stützt sich diese Arbeit auf die wichtigsten Frameworks, die zum einen eine Erfüllung des in Abschnitt 3.1 beschriebenen Kriterienkataloges bieten und zum anderen eine Verbreitung gefunden haben.

2.1 Coupled Layer Architecture for Robotic Autonomy (CLARAty)[1]

CLARAty wurde von der NASA für autonome Erkundungsfahrzeuge entwickelt [Nes+03, Vol+01]. Ziel war, neben einer einfachen Erweiterbarkeit, die Wiederverwendbarkeit auf neuen Plattformen. CLARAty besteht aus zwei Schichten, dem *functional layer* und dem *decision layer*. Der *functional layer* maskiert die Hardware mit einer objektorientierten Abstraktion, die für verschiedene Komponenten entworfen wurde. Dabei unterscheidet man, wie Abbildung 2.2 zeigt, zwischen optionalen und zwingenden Objekten. Das Zusammenspiel und der Datenaustausch zwischen deren Instanzen ist statisch definiert und zur Laufzeit nicht erweiterbar. Die eigentlichen Hardwaretreiber (gelb dargestellt) lassen sich austauschen, so dass die Komponenten des *functional layer* auf die abgebildeten Robotermodelle der NASA gleichermaßen angewendet werden können.

Die Ebene des *decision layer* fasst die üblicherweise in drei Ebenen vorkommenden *planning* und *execution layer* zusammen. Hier werden die abstrakten Hauptziele auf ein Netz kleiner Teilaufgaben abgebildet, die dann ausgehend vom Systemzustand und den Fähigkeiten der Instanzen des *functional layer* ausgeführt werden. Dazu wurde ein aufwendiges Scheduling der Elemente des *functional layers* implementiert.

CLARAty ist vollständig in C implementiert. Als unterstützte Betriebssysteme werden in der Dokumentation VxWorks, Linux und Solaris genannt. Der Anwendungsentwickler wird bei der Integration und Komposition durch UML und Generierungswerkzeuge unterstützt.

[1] http://claraty.jpl.nasa.gov/

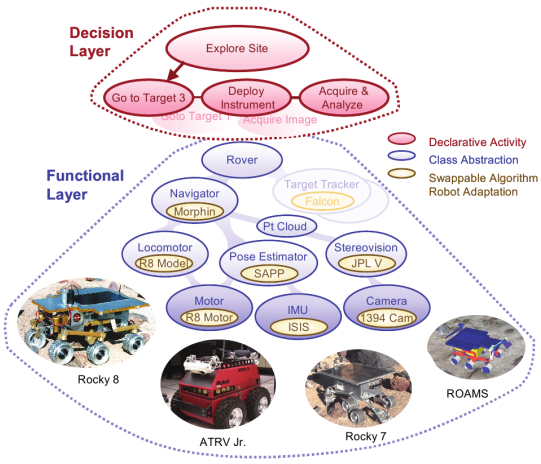

Abbildung 2.2: CLARAty-Architektur [Nes+06]

2.2 Carnegie Mellon Robot Navigation Toolkit (Carmen)[2]

Bereits aus der Bezeichnung Toolkit ist erkennbar, dass Carmen sich eher als Sammlung von verschiedenen robotikspezifischen Bibliotheken für die radgestützte Navigation versteht als ein allgemeingültiges Entwicklungsframework. In dieser Nische wird aber der Anforderungskatalog dieser Arbeit durch Carmen ausgefüllt. So bemühten sich die Entwickler, eine systematische Kapselung der Anwendungsmodule zu erzielen, die weitestgehend unabhängig von zugrundeliegenden Kommunikationskonfigurationen und grafischen Ausgaben sein sollte. Gleichzeitig sind alle Module standardisierten Koordinatensystemen und physikalischen Einheiten unterworfen [MRT03]. Die Kommunikation basiert auf der von der Carnegie Mellon University entwickelten Kommunikationsmiddelware, die unter der irreführenden Bezeichnung Inter Process Communication (IPC)[3] geführt wird. Dabei setzt die Kommunikation auf dem Publish/Subscribe-Paradigma auf und wird über TCP/IP-Protokolle abgebildet.

Bemerkenswert ist die Einrichtung eines globalen Parameterservers. Er liest generelle Konfigurationsgrößen (Basiseinheiten, Timeouts, Roboterkenndaten) aus einer Datei ein und stellt diese Informationen den einzelnen Module zur Verfügung. Mit diesem zentralen Ansatz soll eine Einheitlichkeit der für die Softwareentwicklung mobiler Systeme wichtigen Konstanten gewährleistet werden. Andere Frameworks, wie zum Beispiel ROS (siehe Abschnitt 2.11), setzen auf diesem Konzept auf und erweitern es.

Als Programmiersprache steht lediglich C zur Verfügung, eine Schnittstelle erlaubt jedoch die Integration von Java Code. Entsprechend sind die für den Datenaustausch

[2] http://carmen.sourceforge.net/home.html/
[3] http://www.cs.cmu.edu/afs/cs/project/TCA/www/ipc/ipc.html

hinterlegten Sensor-, Karten- oder Zustandsdatentypen als C-Strukturen definiert. Eine Anpassung durch den Nutzer ist nicht vorgesehen.

Als unterstützte Hardware werden auf der Projektwebseite sowohl komplette Robotersysteme wie Pioneer I und II sowie einzelne Sensoren unterstützt. Dies beschränkt sich jedoch auf einzelne Laserscanner von Sick und Hokuyo sowie GPS-Module, die nach dem NMEA-Protokoll Daten austauschen.

Carmen ist mit Modulen für Teleoperationsszenarien, 2D-Kartenerstellung, usw. stark auf Indoor-Szenarien mobiler radgestützter Roboter ausgerichtet. Dies wird u.a. an den auf der Projektseite veröffentlichten Referenzen deutlich, die überwiegend Lokalisationsprobleme in den Mittelpunkt stellen. Das vorhandene Simulationsmodul dient folglich zur Generierung von positionsrelevanten Sensordaten.

2.3 Mobile and Autonomous Robotics Integration Environment (MARIE)[4]

Mobile and Autonomous Robotics Integration Environment (MARIE) [Côt+04, C+06] ist ein *Middleware*-Rahmenwerk, welches den Fokus auf die Entwicklung und Integration neuer und existierender Software für Robotiksysteme legt. Das Ziel dieser Entwicklung liegt dabei darin, ein flexibles, verteiltes Komponentensystem zu schaffen, welches es den Robotikentwicklern ermöglicht, Programme für eine schnelle Entwicklung von Robotikapplikationen zu teilen, wiederzuverwenden und zu integrieren. Dabei steht vor allem die Interoperabilität zwischen verschiedenen Entwicklungsumgebungen für Robotikanwendungen im Vordergrund. Das bedeutet, dass im Zuge des Entwicklungsprozesses auf verschiedene Programmierumgebungen zurückgegriffen werden kann, um deren Vorteile zu nutzen, während MARIE als Schnittstelle zwischen diesen agiert. MARIE ist demnach kein Software-Rahmenwerk im eigentlichen Sinne, sondern eher als eine Art „Meta-Rahmenwerk“ zu sehen, welches Komponenten aus anderen Rahmenwerken integriert. Neben dem bereits genannten Aspekt der Wiederverwendbarkeit von vorhandenen Software-Lösungen stehen auch die Unterstützung von verschiedenen Kommunikationsprotokollen und -mechanismen sowie eine klare Definition von Schnittstellen und Abstraktionsebenen im Vordergrund dieser Entwicklung.

Die Software-Architektur von MARIE, welche einen komponentenbasierten Ansatz verfolgt, basiert auf folgenden Konzepten, um die zuvor erwähnte Zielstellung zu erreichen: Implementierung des *Mediator*-Entwurfsmusters Gamma u. a. [Gam+95], dreischichtiges Architekturmodell sowie Abstraktion von Kommunikationsprotokollen.
Mit der Umsetzung des *Mediator*-Entwurfsmusters geht eine zentrale Kontrolleinheit, der *Mediator*, einher, welcher unabhängig mit anderen Komponenten interagiert. Somit wird eine globale Interaktion zwischen den Komponenten ermöglicht, um das gewünschte Gesamtsystem zu realisieren. Der *Mediator* ist in MARIE als *Mediator*-Interoperabilitätsschicht

[4] http://sourceforge.net/projects/marie/files/

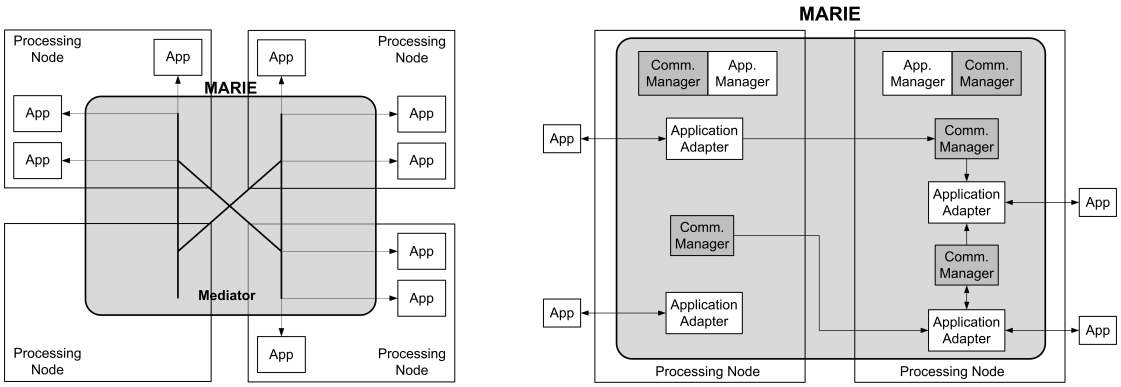

Abbildung 2.3: MARIE-Architektur (Vergleich Côté u. a. [Côt+04]).

(Mediator Interoperability Layer (MIL)) in Form eines virtuellen Raumes (engl. *virtual space*), in dem die einzelnen Komponenten über eine gemeinsame Sprache (z.B. Extensible Markup Language (XML)) interagieren können, realisiert. Das bedeutet, dass die Kommunikation zwischen zwei Komponenten nicht direkt stattfindet, sondern jede nur mit der Vermittlerschicht kommuniziert (Vergleich Abbildung 2.3). Daraus folgt, dass nicht jedes Modul alle Kommunikationsprotokolle seiner potentiellen Kommunikationspartner unterstützen muss, sondern sein eigenes Kommunikationsprotokoll verwenden kann, solange dieses von der MIL unterstützt wird. Auf diese Weise kann auf den Stärken der vielfältigen Protokolle aufgebaut werden. Dieses „Eins-zu-viele“-Interaktions-Modell ermöglicht des Weiteren eine lose Kopplung zwischen den einzelnen Komponenten. Solch eine lose Kopplung geht mit einer höheren Wiederverwendbarkeit, Interoperabilität und Erweiterbarkeit der Komponenten einher Côté u. a. [Côt+06]. Allerdings ist anzumerken, dass durch diesen Ansatz der *Mediator* selbst sehr komplex wird, was zu einer schlechteren Wartbarkeit und Erweiterbarkeit dieser Vermittlerkomponente führen kann.
Die dreischichtige Architektur definiert folgende drei Abstraktionsebenen: Kernschicht, Komponentenschicht und Anwendungsschicht. Die Kernschicht besteht aus Diensten für die Kommunikation, Betriebssystemoperationen auf unterster Ebene wie Speicherverwaltung und Threadkontrolle sowie Funktionen für verteiltes Rechnen. Oberhalb dieser Schicht bietet die Komponentenschicht Funktionen für das Implementieren von Komponenten. Nützliche Dienste und Funktionen für das Erzeugen und Verwalten integrierter Applikationen zur Robotersteuerung, die auf den verfügbaren Komponenten der unteren zwei Ebenen aufbauen, werden dagegen von der Anwendungsschicht bereitgestellt.
Die Abstraktion von Kommunikationsprotokollen in Form von Ports bedeutet eine Vereinfachung der Interoperabilität und Wiederverwendbarkeit von Komponenten, da MARIE nicht auf ein Kommunikationsprotokoll festgelegt ist. Stattdessen werden unterschiedliche Mechanismen unterstützt, wie z.B. Kommunikation über einen gemeinsam

genutzten Speicher, Inter-Prozess-Kommunikation Inter-process Communication (IPC) oder Verwendung von Transmission Control Protocol (TCP) und User Datagram Protocol (UDP) für einen netzwerkbasierten Nachrichtenaustausch.

Die Applikationsentwicklung mit MARIE basiert auf wiederverwendbaren Software-Blöcken, den Komponenten. Diese implementieren Funktionalitäten, indem existierende Applikationen, Programmierumgebungen oder Algorithmen gekapselt werden. Aus dem Konfigurieren und Verknüpfen dieser Komponenten ergibt sich die gewünschte Systemfunktionalität. Die Integration setzt dabei, genau wie Miro (siehe Abschnitt 2.4), auf ACE auf. Im Mittelpunkt des Verknüpfens von Komponenten steht die zuvor erwähnte zentrale Kontrolleinheit. MIL stellt für diesen Zweck vier funktionale Komponenten bereit: den Applikations-Adapter, den Kommunikations-Adapter, den Applikations-Manager und den Kommunikations-Manager (Vergleich Abbildung 2.3). Der Applikations-Adapter agiert als Proxy zwischen der MIL und der Anwendung und ermöglicht somit die Interaktion zwischen den einzelnen Applikationen. Die Daten, die zwischen Anwendungen ausgetauscht werden, werden von dem Kommunikations-Manager übersetzt. Dabei erfolgt ein Anpassen von inkompatiblen Kommunikationsprotokollen. Der Kommunikations-Manager erzeugt und verwaltet dagegen die Verbindungen zwischen den einzelnen Komponenten, während der Applikations-Manager diese instanziiert und verwaltet, und zwar lokal oder über Netzwerkknoten verteilt.

Zusätzlich zu dem durch MARIE beschriebenen Systementwurf existieren die beiden darauf aufsetzenden Entwicklungswerkzeuge *FlowDesigner* und *RobotFlow* Côté u. a. [Côt+04]. Diese unterstützen im Zuge der Applikationsentwicklung den Entwurf wiederverwendbarer Softwarekomponenten. *FlowDesigner* ist eine datenflussverarbeitende Bibliothek, die an eine grafische Programmierumgebung gekoppelt ist. Diese erlaubt das Erstellen wiederverwendbarer Softwareblöcke, die hierarchisch in einem Netzwerk von funktionalen Blöcken miteinander verbunden werden können. Die Motivation dieser Programmierumgebung liegt in der Unterstützung des Debuggingprozesses und des Betrachtens der über die Komponentenverbindungen übertragenen Daten. *RobotFlow* ist Robotik-Entwicklungswerkzeug, welches auf *FlowDesigner* basiert.Für eine weiterführende Betrachtung dieser beiden Werkzeuge sei an dieser Stelle auf Côté u. a. [Côt+04] verwiesen.

Die zuvor beschriebenen Ansätze machen MARIE sehr flexibel, was das Entwickeln und Integrieren von neuer und bereits existierender Software angeht. Im Fokus steht dabei die Interoperabilität zwischen und Wiederverwendung von Komponenten von Robotikapplikationen. Auf diese Weise wird der Anwendungsentwicklungsprozess vereinfacht und beschleunigt, was auch ein *Rapid-Prototyping* ermöglicht.
Publikationen, die auf MARIE aufsetzen, beinhalten Pfadplanung und Navigation [Bea+05], Multi-Roboter-Lokalisierung [Riv05], Multi-Roboter-Koordination [Gau+03] und Multi-Roboter-Formation [Lem+04].

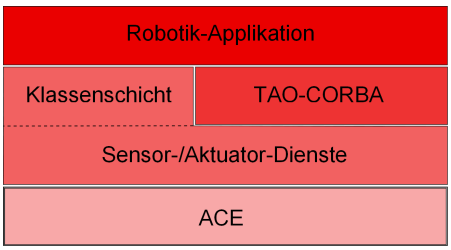

Abbildung 2.4: Miro-Architektur (Vergleich Enderle u. a. [End+01]).

2.4 MIddleware for RObotics (MIRO)[5]

Die unter anderem von Enderle u. a. [End+01] und Utz u. a. [Utz+02] beschriebene objektorientierte *Middleware* MIddleware for RObotics (MIRO) wurde an der Universität Ulm entwickelt wurde. Die Hauptmotivation dieser Entwicklung liegt in der Verbesserung des Software-Entwicklungsprozesses durch das Vereinfachen der Integration heterogener Software-Module. Während mit der Verwendung von C++ eine hohe Laufzeit-Effizienz einhergeht Utz u. a. [Utz+02], wird durch das Aufbauen auf der CORBA-Architektur sowohl eine Orts- als auch eine Programmiersprachenunabhängigkeit erreicht. Die Umsetzung des CORBA-Standards erfolgt dabei mittels TAO-Common Object Request Broker Architecture (CORBA) Siegel u. a. [Sie+96]. Idealerweise sollte diese Abstraktionsschichten eine plattformübergreifende Interoperabilität für viele (Echtzeit-)Betriebssysteme wie Windows, Unix/Linux, LynxOS und VxWorks bieten, gegenwärtig ist aber nur eine Linuximplementierung von MIRO verfügbar.

Die MIRO-*Middleware* bietet zwei Hauptfunktionalitäten Enderle u. a. [End+01]: Dies sind zum einen Dienste, welche die Sensor- und Aktuator-Fertigkeiten der spezifischen Robotikplattform adressieren. Zum anderen enthält Miro eine Klassenschicht, welche Entwurfsmuster für die Steuerung von Robotern implementiert. Sensoren und Aktuatoren werden dabei als Objekte modelliert, sodass ein Roboter als Aggregation solcher Objekte abgebildet wird. Die allgemeine Struktur von Miro ist in Abbildung 2.4 dargestellt. Kennzeichen der zugrunde liegenden Architektur sind drei Schichten Utz u. a. [Utz+02]: Die Geräteschicht, die Diensteschicht und die Klassenschicht.
Die Geräteschicht stellt objektorientierte Schnittstellen-Abstraktionen für einzelne Sensoren und Aktuatoren zur Verfügung. Dabei werden kontrollerspezifische nachrichten- oder paketbasierte Kommunikationsverbindungen wie Controller Area Network (CAN) oder serielle Verbindungen in „einfache“ Methodenaufrufe gekapselt. Somit können diese ohne spezielle Kenntnisse der auf Kontrollerebene verwendeten Kommunikationsprotokolle von den Dienst-Implementierungen genutzt werden.
Die Diensteschicht bietet objektorientierte Schnittstellen für Sensoren und Aktuatoren.

[5] http://miro-middleware.berlios.de/

Entsprechende Abstraktionen in Form sogenannter *Wrapper*-Klassen verbergen dabei die Eigenheiten der sensor- bzw. aktuatorspezifischen Schnittstellenfunktionen. Erreicht wird dies durch die CORBA-Interface Definition Language (IDL), sodass einzelne Dienste in Form netzwerktransparenter Objekte implementiert werden können. Mit der Verwendung solcher CORBA-Objekte gehen, wie bereits erwähnt, eine Orts- und eine Programmiersprachenunabhängigkeit einher. Voraussetzung dafür ist allerdings, dass sowohl die Plattform als auch die Programmiersprache den CORBA-Standard z.B. in Form entsprechender Bibliotheken unterstützt. Ortsunabhängig bedeutet dabei, dass die entsprechenden Objekte, die miteinander kooperieren, nicht auf demselben System laufen müssen, sondern verteilt in einem Netzwerk agieren können. Eine entsprechende CORBA-Unterstützung vorausgesetzt, können auf diese Weise heterogene Software-Module in ein System integriert werden. Darüber hinaus bietet die Diensteschicht eine Klassenhierarchie, die nützliche Abstraktionen und Generalisierungen für bestimmte Sensoren und Aktuatoren spezifiziert. Ein Beispiel dafür ist die *RangeSensor*-Klasse, die die Schnittstellenfunktionalität, die Sensoren wie Infrarot- oder Sonarsensoren gemeinsam haben, definiert.

Die Interaktion und Kommunikation zwischen den Diensten basiert auf dem Konzept des entfernten Methodenaufrufes Remote Method Invocation (RMI) und dem CORBA-*Notification-Service* Harrison, Levine und Schmidt [HLS97]. Dieser Dienst ermöglicht eine asynchrone, ereignisbasierte Kommunikation in Form eines *Publish-Subscribe*-Modelles über sogenannte Benachrichtigungskanäle. Die Verbreitung der Ereignisse als CORBA-Objekte erfolgt dabei mittels Multicast, welches von dem in Miro enthaltenen *Notify-Multicast*-Modul gesteuert wird.
Die Klassenschicht bietet funktionelle Module, die Techniken, die im Bereich der Robotersteuerung oft benötigt werden, implementieren. Zu den bereitgestellten Funktionalitäten zählen eine Verhaltensgenerierung, *Mapping*, Selbstlokalisierung, Pfadplanung sowie *Logging*- und Visualisierungsmöglichkeiten. Mit der einheitlichen Bereitstellung dieser Funktionalitäten auf den unterstützten Plattformen wird auch der Aspekt der Wiederverwendung von Code unterstrichen, was zu einer einfacheren und schnelleren Entwicklung von Robotiksoftware führt. Darüber hinaus bietet der *Logging*-Mechanismus auch die Möglichkeit, protokollierte Daten wiederabzuspielen, indem diese aufgezeichneten Ereignisse über einen Benachrichtigungskanal zur Verfügung gestellt werden. Damit geht die Möglichkeit einher, Tests und Bewertungen der implementierten Funktionen offline durchzuführen, was den Softwareentwicklungsprozess positiv beeinflusst.

Aktuell ist Miro für drei Roboterplattformen, die verschiedene Hardwarekomponenten besitzen und in unterschiedlichen Bereichen wie Büroumgebungen oder dynamischen Fußballspielen (RoboCup) Enderle u. a. [End+01] eingesetzt werden, implementiert. Dazu gehören der B21, der Pioneer-1 und der Sparrow-99. Außerdem beinhaltet Miro einige Dienst-Implementierungen, die im Zuge der Entwicklung von Robotikapplikationen oft benötigt werden: Dies umfasst einen Dienst für die Steuerung von Motoren, der auch Informationen über die Roboterposition mithilfe der Odometrie bereithält, und Diens-

te, die Daten unterschiedlicher Sensoren wie Laserscanner, Infrarot- und Drucksensoren sowie Kamerabilder zur Verfügung stellen. Weiterhin werden für jede dieser Komponenten Beispiel- und Test- sowie *Monitoring*-Programme, die das Testen, Evaluieren und Visualisieren der Funktionen bzw. der erzeugten Daten ermöglichen, bereitgestellt.
Publikationen, die auf der Miro-*Middleware* aufsetzen, reichen von Simultaneous Localization and Mapping (SLAM) [KUG04] über Pfadplanung und Navigation [Kra+00] bis hin zu Multi-Roboter-Koordination [USM05].

2.5 Player/Stage/Gazebo[6]

Das Player/Stage/Gazebo-System Gerkey, Vaughan und Howard [GVH03] Collett, MacDonald und Gerkey [CMG05] ist eine *Middleware*, die neben Infrastrukturkomponenten auch Gerätetreiber und einige Algorithmen für Robotikapplikationen wie Navigationsalgorithmen bereitstellt. Die Hauptmotivation dieser Entwicklung liegt in dem Anbieten von einheitlichen Programmierschnittstellen, um die Integration von heterogenen Sensoren und Aktuatoren in ein Gesamtsystem zu vereinfachen. Zwei Komponenten bilden den Kern dieser *Middleware*: Player und Stage/Gazebo. Während Player als Schnittstelle zu den verschiedenen Robotikgeräten dient und Treiber für viele Hardware-Module bereithält, von der eigentlichen Hardware abstrahiert und einheitliche Schnittstellen anbietet, stellen Stage und Gazebo einen grafischen 2-D- bzw. 3-D-Simulator dar, der die einzelnen Geräte in einer benutzerdefinierten Umgebung modelliert.

Die Architektur dieses Systemes, die in der Abbildung 2.5 dargestellt ist, besteht aus drei Ebenen. Die erste Ebene bilden die sogenannten Clienten, die die für die spezifische Robotikapplikation entwickelten Software-Komponenten darstellen. Die mittlere Schicht stellt das Player-Modul dar, das in zwei Komponenten unterteilt ist, und zwar in die Kernschicht und in die Transportschicht.
Aktuell ist eine TCP/IP-Transportschicht (Transmission Control Protocol/Internet Protocol) implementiert. Denkbar ist aber auch eine Kommunikation basierend auf CORBA oder JINI Waldo [Wal99]. Darüber hinaus besteht auch die Möglichkeit, ein monolithisches System aufzusetzen, indem die Kommunikation mittels eines internen Nachrichtenaustausches ohne eine koordinierende Netzwerkschicht erfolgt. Der Player-Kern stellt dagegen eine Programmierschnittstelle und entsprechende Funktionalitäten wie Geräte- und Treiberklassen zur Verfügung. Des Weiteren ist diese Kernschicht für die Koordination des Nachrichtenaustausches, die auf Warteschlangen basiert, und die Spezifikation der Nachrichtensyntax verantwortlich. Die Player-Schicht fungiert im Gesamtsystem in der aktuellen Implementierung als socketbasierter Geräte-Server und bietet dabei einfache Schnittstellen zu den Sensoren und Aktuatoren über ein Netzwerk an. Demnach wird von der eigentlichen Roboterhardware, die die dritte Ebene in dem Architekturmodell repräsentiert, abstrahiert.

[6] http://playerstage.sourceforge.net/

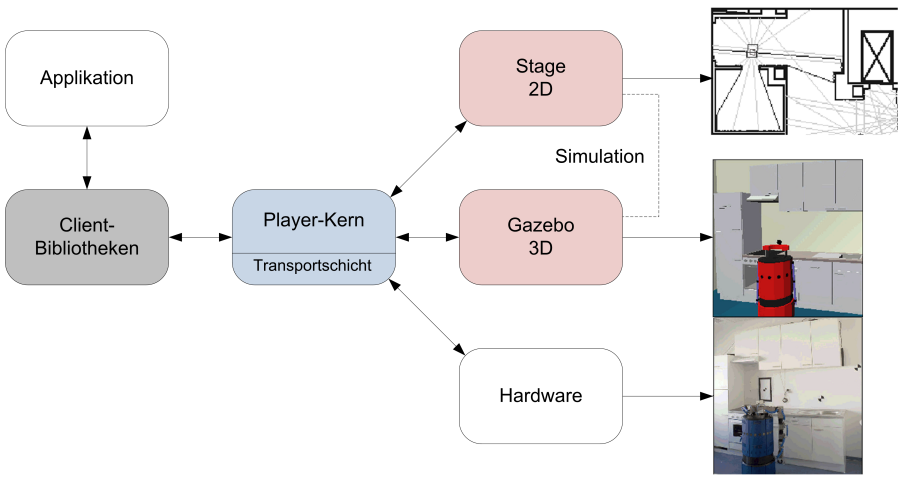

Abbildung 2.5: Player-Architektur (Vergleich Rusu u. a. [Rus+07]).

Das in Player verwendete Gerätemodell folgt dabei dem *device-as-file*-Modell Feiertag und Organick [FO71], welches Geräte in der Behandlung mit Dateien gleichsetzt, und trennt die Geräteschnittstellen von den Treibern. Die Schnittstelle spezifiziert dabei Daten-, Befehls- und Konfigurationsformate, während der Treiber das Gerät steuert und eine einheitliche Standardschnittstelle zu diesem anbietet. Diese Schnittstellen werden von den Clienten genutzt, um z.B. Daten von einem Sensor zu lesen oder einen Aktuator zu steuern. Dazu besitzt jeder Treiber - wie auch das Client-Programm - eine Warteschlange für eingehende Nachrichten. In dem Player-Konzept kann der Treiber mehrere „Rollen“ annehmen: Dieser kann einerseits Code darstellen, der sich mit dem physikalischen Gerät verbindet und mit diesem kommuniziert. Andererseits kann ein Algorithmus repräsentiert werden, der Daten von einem anderen Gerät empfängt, verarbeitet und anschließend wieder zurücksendet. Darüber hinaus ist auch die Funktion eines „virtuellen“ Treibers möglich, der bei Bedarf beliebige Daten generieren kann. Die Kommunikation mit anderen Treibern bzw. mit den Clienten erfolgt dabei durch das Senden von Nachrichten an die entsprechenden Nachrichten-Warteschlangen. Dabei wird auch das Senden von Daten an alle abonnierenden Clienten im Sinne eines *Publisher-Subscribe*-Modelles unterstützt. Die Transportschicht übernimmt in diesem Zusammenhang das *Routing* der Nachrichten aus den Warteschlangen an die entsprechenden Sockets und umgekehrt. Das Konzept des Nachrichtenaustausches basierend auf Warteschlangen ermöglicht somit, wie bereits erwähnt, monolithische Applikationen, die direkt mit den Geräten über deren Warteschlangen interagieren.
Der eben beschriebene Player-Server läuft auf der eigentlichen Roboterplattform, wobei sich die einzelnen Geräte mit diesem registrieren, sodass sie für die Clienten „nutzbar“ werden. Die Clienten sind in der aktuellen Implementierung wiederum über eine TCP-

Socket-Verbindung mit Player verbunden. Über diese Schicht können die Clienten mit den einzelnen Geräten mittels Nachrichtenaustausch interagieren. Dabei hat ein Client auch die Möglichkeit, einzelne Geräte zu abonnieren, sodass die gewünschten Daten mit einer zu spezifizierenden Frequenz zur Verfügung stehen. Jeder Client benutzt zudem eine eigene Socket-Verbindung für den Datentransfer. Es ergibt sich somit eine „Eins-zu-viele“-Client/Server-Architektur. Die Tatsache, dass die Serverkommunikation über die Socket-Schnittstelle erfolgt, bedeutet, dass Client-Programme in jeder Programmiersprache mit Socket-Unterstützung wie C/C++, Java oder Python implementiert werden können. Daraus folgt eine Programmiersprachenunabhängigkeit, womit auch eine Plattformneutralität einhergeht Gerkey, Vaughan und Howard [GVH03]. Darüber hinaus bedingt das Socket-Konzept noch einen weiteren Vorteil: Da ein Client-Programm von jedem netzwerkfähigen Rechner aus den Roboter steuern kann, ergibt sich somit auch eine Ortsunabhängigkeit, womit sich neue Möglichkeiten für das Aufbauen von verteilten Robotik-Steuerungssystemen ergeben. Des Weiteren zeichnet sich die Player-*Middleware* durch ihre Modularität und Flexibilität hinsichtlich neuer Hardware aus.

Die Simulationsumgebungen Stage und Gazebo dienen der zwei- bzw. dreidimensionalen Simulation benutzerdefinierter Szenarien. Dabei wird das Verhalten der realen Geräte bzw. das derer Treiber nachgebildet, sodass Client-Programme in einer Umgebung ähnlich der der realen Hardware entwickelt und getestet werden können. Auf diese virtuellen Geräte kann dabei, genau wie auf die realen, über den Player-Server zugegriffen werden, sodass sich clientseitig kein Unterschied in der Behandlung der einzelnen Komponenten ergibt. Auf diese Weise können Applikationen entwickelt werden, ohne die reale Hardware besitzen zu müssen, was eine schnelle Anwendungsentwicklung fördert und auch dabei hilft, Kosten bezogen auf den Entwicklungsprozess zu sparen.

Player/Stage/Gazebo bietet unter anderem Unterstützung für folgende Roboterplattformen: *MobileRobots*, *SegwayRMP*, *Acroname's Garcia*, *Botrics's Obot d100*, *Evolution Robotics*-, *CoroWare*- und *K-Team*-Roboter. Darüber hinaus beinhaltet dieses Rahmenwerk auch Komponenten für Pfadplanung, Lokalisierung und Kollisionsvermeidung bzw. Hinderniserkennung.
Projekte, die auf der Player-*Middleware* aufsetzen, reichen von SLAM Wolf und Sukhatme [WS05] und Multi-Roboter-Erkundung Howard, Matarić und Sukhatme [HMS02] über Multi-Roboter-*Mapping* und Planung Howard, Parker und Sukhatme [HPS04] bis hin zu Multi-Roboter-Koordination Jones und Matarić [JM04] und -Aufgabenverteilung Gerkey und Matarić [GM02].

2.6 OROCOS[7]

Das Ziel des Open Robot Control Software (OROCOS) Projekt Bruyninckx [Bru01] ist es, die Vorteile freier Software auf Robotik und Maschinensteuerung zu übertragen.

[7] http://www.orocos.org/

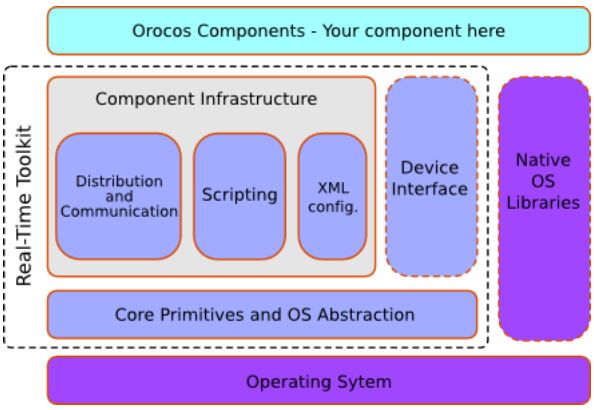

Abbildung 2.6: Aufbau einer Anwendung, die auf dem RTT basiert. (Quelle: http://www.orocos.org/rtt).

Hierfür wird ein modulares Framework entwickelt, das sich aus mehreren Bibliotheken zusammensetzt. Die Hauptbibliothek ist das Real-Time Toolkit (RTT). Es bietet Kommunikationsschnittstellen und Funktionalitäten zum Erstellen von Robotikanwendungen in C++. Gestartet wurde die Entwicklung im Jahr 2001 unter der Beteiligung von Forschungsgruppen an der KU Leuven, KTH Stockholm und LAAS Toulouse. Das Projekt setzt sich aus vier Bibliotheken zusammen: dem RTT, der Orocos Component Library (OCL), der Kinematics and Dynamics Library (KDL) und der Bayesian Filtering Library (BFL).

Das RTT erlaubt es dem Entwickler, komponentenbasierte Echtzeit-Anwendungen für die Steuerung oder Regelung zu erstellen. Um die Echtzeitfähigkeiten nutzen zu können, muss die Anwendung auf einem Echtzeitbetriebssystem ausgeführt werden. Unterstützt wird hierfür Linux mit einer der Echtzeiterweiterungen RTAI/LXRT oder Xenomai. Für die Anwendungsentwicklung, wenn auf harte Echtzeit verzichtet werden kann, können Anwendungen auch in einer gewöhnlichen Linux-Umgebung ausgeführt werden.

Auch für die Kommunikation zwischen Komponenten werden einheitliche Mechanismen bereitgestellt. Laufen alle Komponenten auf dem selben Knoten, kommt die MQueue Bibliothek zum Einsatz, die eine Echtzeit-Kommunikation zwischen Prozessen erlaubt. Sie setzt hierfür auf POSIX-Nachrichtenwarteschlangen auf.

Die Kommunikation zwischen Komponenten auf entfernten Knoten erfolgt über CORBA. Entweder unter Verwendung von ACE & TAO oder aber Omniorb, wenn auf Echtzeit verzichtet werden kann. Kommunikationsschnittstellen zu ROS Smits und Bruyninckx [SB11], hierbei werden ROS Nachrichten in Orocos-Typen umgewandelt, und YARP stehen ebenfalls bereit. Die Konfiguration erfolgt durch XML-Dateien, in denen das Zusammenspiel der einzelnen Komponenten beschrieben wird, wie Abbildung 2.6 zeigt.

Um eine schnelle und einfache Anpassung von Anwendungen zu erlauben, werden Scripte unterschützt. Programmfunktionalität kann in LUA Klotzbücher, Soetens und Bruyninckx [KSB10] beschrieben werden, während die Beschreibung von Zustandsautomaten in einer eigenen Sprache möglich ist.

Grundlegende Funktionalitäten werden von OCL abgedeckt. Neben einer Beispielimplementierung einer Komponente stehen hier häufig benötigte Komponenten zur Nutzung bereit, etwa für Aufgaben wie das Aufzeichnen von Datenflüssen, Kontrolle laufender Komponenten, Konfiguration der Komponenten einer Anwendung und Knoten für häufig verwendete Hardwarekomponenten wie Kinematiken von Kuka oder Laserdistanzsensoren, um nur eine kleine Auswahl zu nennen.

Die Berechnung einer kinematischen Kette erfolgt durch die KDL. Sie bietet ein anwendungsunabhängiges Framework für das Modellieren und die Berechnung von kinematischen Ketten, wie etwa für einen Gelenkarmroboter oder das biomechanische Modell des menschlichen Körpers. Die Bibliothek bietet entsprechend Klassenbibliotheken an, um zum einen geometrische Objekte zu beschreiben und zum anderen stehen zahlreiche kinematische Ketten für verschiedene Problemklassen bereit.

Für die Nutzung dynamischer Bayes'scher Netze steht die BFL bereit. Sie kann genutzt werden, um Schätzungen zu machen.

Der Fokus von OROCOS ist klar über die Industrierobotik definiert. Durch die anwendungsunabhängigen Lösungsansätze lassen sich die Bibliotheken jedoch auch für weitere Problemfelder nutzen.

2.7 Open Real Time Middleware (OpenRTM)[8]

Die aus Japan stammende RT-Middleware [And+05] wurde in Kooperation der Japanischen Ministerien für Wirtschaft, für Handel und für Industrie sowie der JARA und des AIST entwickelt. Das Ziel besteht in einer Modularisierung der Softwareentwicklung für Roboter, um deren Bau zu vereinfachen. Die Komponenten, aus denen für die Anwendungen Systeme zusammengefügt werden, heißen „RT-Components". Die RT-Middleware basiert auf CORBA. Ein weiteres wichtiges Ziel besteht darin, intelligentere Roboter zu ermöglichen, indem die notwendigen Rechenkapazitäten im Netzwerk verteilt werden. Die zum Aufbau verteilter Systeme notwendige Funktionalität ist somit vorhanden. Zielplattformen des Projekts liegen auch im Heimbereich.

2.8 Yet Another Robot Platform (YARP)[9]

YARP ist ein Framework, welches den Fokus auf die Steuerung humanoider Roboter legt. [FMN08, MFN06]. Ziel dabei ist es, die Anwendungsentwicklung durch Wiederverwend-

[8] http://www.openrtm.org/
[9] http://eris.liralab.it/yarp/

barkeit von Code und Modularität zu erleichtern sowie den Entwicklungsaufwand für das Einrichten einer Infrastruktur zwischen Komponenten zu minimieren. Erreicht wird dies einerseits durch die konsequente Verwendung von C++, wobei die Objektorientierung bzw. Klassenhierarchie bei der Code-Wiederverwendbarkeit und dem modularen Aufbau unterstützen. Andererseits ermöglicht das verwendete, transportneutrale Kommunikationsmodell, welches von den zugrunde liegenden Netzwerken und Protokollen entkoppelt ist, eine Interoperabilität zwischen heterogenen Komponenten und Plattformen.
Unterstützt wird dieser Aspekt der Portabilität auch durch die Verwendung von ACE, welches eine Betriebssystem- und somit auch Plattformunabhängigkeit gewährleistet. So werden z.B. Windows, Linux, QNX 6 und Mac OSX unterstützt.
Der in YARP verwendete Kommunikationsmechanismus basiert auf dem *Observer*-Entwurfsmuster Gamma u. a. [Gam+95]. Dabei senden spezielle *Port*-Objekte Nachrichten an alle *Observer*, welche ebenfalls durch *Port*-Objekte repräsentiert werden und sich verteilt in einem heterogenen Netzwerk befinden können. *Ports* stellen dabei die eigentliche Abstraktion der *Peer-to-Peer*-Kommunikationsverbindung, welche auf *Sockets* basiert, dar. Somit wird auch eine Ortsunabhängigkeit erreicht.
Abhängig von den Erfordernissen kann dabei das Übertragungsmedium und -protokoll gewählt werden: Zur Auswahl stehen etwa TCP/IP, UDP-Unicast und UDP-Multicast, aber auch gemeinsamer Speicher lässt sich zum Austausch von Daten nutzen. Unterstützt werden dabei sowohl eine asynchrone Kommunikation, welches das Standardverhalten des Frameworks darstellt, als auch eine synchrone. Dies wird durch entsprechende Methodenaufrufe realisiert.
Die Anwendungsentwicklung kann in zahlreichen Sprachen erfolgen: So stehen etwa Java, Python und Matlab zur Auswahl. Neben dem Ansprechen von Hardware unterstützt YARP auch Bildverarbeitung.

2.9 Physically Embedded Intelligent Systems (PEIS)[10]

The PEIS Kernel Broxvall, Seo und Kwon [BSK07] ging aus einer Kooperation des koreanischen *Electronics and Telecommunications Research Institute* (ETRI) und des schwedischen *Centre for Applied Autonomous Sensor Systems* hervor. Diese *Middleware* basiert auf der sogenannten Ökologie von Physically Embedded Intelligent Systems (PEIS). Dabei steht die Kooperation vieler Robotikgeräte, den PEIS, bei der Bewältigung einzelner Aufgaben im Vordergrund. Komplexe Funktionalitäten werden demnach durch das Zusammenwirken vieler einfacher Robotikkomponenten anstelle fortgeschrittener „großer“ Roboter erreicht. Im Fokus stehen dabei Anwendungen in häuslichen und Büroumgebungen, die Dienste zur Unterstützung des Menschen anbieten. In diesem Zusammenhang besteht die Hauptaufgabe der *Middleware* darin, ein gemeinsames Kommunikations- und Kooperationsmodell für die einzelnen Robotikgeräte wie statische Sensoren und Aktuato-

[10] http://www.aass.oru.se/~peis

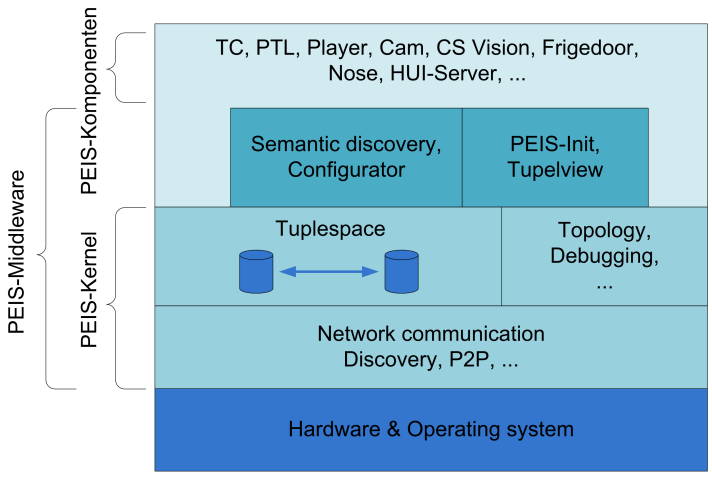

Abbildung 2.7: Architektur der PEIS-*Middleware* (Vergleich Broxvall, Seo und Kwon [BSK07]).

ren, „intelligente“ (Haushalts-)Geräte und traditionelle mobile Roboter anzubieten und deren Integration in ein Gesamtsystem zu erleichtern. Diese miteinander interagierenden und zusammenarbeitenden Komponenten bilden dann die erwähnte PEIS-Ökologie. Bei einem PEIS handelt es sich dabei um ein beliebiges Gerät, welches Rechen- und Kommunikationsressourcen besitzt. Allgemein wird ein PEIS als eine Menge von miteinander verbundenen Software-Komponenten, den PEIS-Komponenten, die sich in einem physikalischen Gerät befinden, definiert Broxvall, Seo und Kwon [BSK07]. Diese PEIS-Komponenten implementieren wiederum alle Funktionalitäten eines PEIS in Form entsprechender Algorithmen. Über Sensoren und/oder Aktuatoren ist auch deren Interaktion mit der Umgebung möglich, während die implementierten Funktionalitäten über entsprechende Ports anderen PEIS zur Verfügung gestellt werden können. Dies bildet die Basis für die Kooperation der einzelnen Komponenten in der PEIS-Ökologie.

Das Ermöglichen der Interaktion zwischen den einzelnen PEIS ist die Aufgabe der PEIS-*Middleware*, indem ein einheitliches Kommunikations- und Kooperationsmodell angeboten wird. Verantwortlich dafür zeigt sich der in der *Middleware* neben weiteren Softwarebibliotheken enthaltene PEIS-Kernel. Darüber hinaus bietet die Middleware auch Software-Komponenten, die das *Debugging* und die Visualisierung des aktuellen Status der PEIS-Ökologie ermöglichen. Die zugrunde liegende Schichtenarchitektur ist in der Abbildung 2.7 dargestellt. Der PEIS-Kernel implementiert ein dezentrales, verteiltes Kommunikationsmodell, welches auf einem gemeinsam genutzten Speicher, dem sogenannten *Tuplespace*, und einem Event-Mechanismus, welcher ein *Publish-Subscribe*-Modell ermöglicht, basiert. Ein Tupel beschreibt dabei, welche Funktionalität, repräsentiert durch einen Tupelschlüssel, und welche Daten eine Komponente in einem bestimm-

ten PEIS anbietet. Der PEIS-Kernel verbirgt in diesem Konzept die zugrunde liegende Netzwerkheterogenität, indem die Kommunikation über den Austausch von Tupeln des *Tuplespaces* unabhängig von den vorhandenen physikalischen Verbindungen realisiert wird.

Der PEIS-Kernel sitzt auf der Betriebssystemschicht auf und abstrahiert von den betriebssystemspezifischen Kommunikationsmethoden. Des Weiteren wird auf dieser Ebene auch ein Mechanismus für das Detektieren von PEIS-Geräten realisiert. Um die Kommunikation zwischen diesen Komponenten über den gemeinsam genutzten Speicher zu ermöglichen, werden alle in der lokalen Umgebung vorhandenen PEIS detektiert und miteinander verbunden. Dazu wird in der übergeordneten Ebene von den Kommunikationsverbindungen der einzelnen Geräte abstrahiert und ein *Peer-to-Peer*-Netzwerk, welches auf TCP/IP basiert, zwischen diesen aufgebaut. Auf diese Weise wird ein Informationsaustausch in Form von Tupeln zwischen den teilnehmenden PEIS möglich. Dieses Netzwerk aktualisiert sich dabei automatisch, sobald Komponenten das Netzwerk verlassen bzw. diesem beitreten. Ist eine PEIS-Komponente an bestimmten Daten einer anderen Komponente interessiert, abonniert diese den Tupelschlüssel des korrespondierenden Tupels. Liefert die *Publisher*-Komponente neue Werte, werden diese transparent über den PEIS-Kernel der *Subscriber*-Komponente über eine zuvor registrierte *Callback*-Funktion zur Verfügung gestellt. PEIS-Komponenten, die sich in demselben Gerät befinden, können über entsprechende Lese- und Schreiboperationen auch direkt auf die Tupel und somit auf die Daten der anderen Komponenten zugreifen. Für Komponenten anderer PEIS ist dagegen nur eine Interaktion über das *Publish-Subscribe*-Modell zulässig. Die Verwendung eines gemeinsam genutzten Speichers macht es darüber hinaus den Komponenten möglich, nach bestimmten Funktionalitäten anderer Komponenten, die für die aktuelle Aufgabenbewältigung benötigt werden, zu suchen. Dazu wird in dem *Tuplespace* nach einem Tupel gesucht, welches die gewünschte Funktionalität anbietet, und anschließend mit der korrespondierenden Komponente interagiert. Dadurch wird eine dynamische Selbstkonfiguration des Gesamtsystemes entsprechend der zu bewältigenden Aufgabe, des Zustandes der Umgebung und der in der Ökologie zur Verfügung stehenden Funktionalitäten bzw. Ressourcen möglich. Unterstützt wird dies durch die in die *Middleware* integrierte Komponente *PEISInit*, die auf jedem PEIS läuft. Diese verwaltet dessen Komponenten und bietet zudem eine semantische Beschreibung dieser an. Darüber hinaus macht dieses Modul den Ausfall einer PEIS-Komponente der Ökologie durch ein entsprechendes Fehlertupel bekannt. Zusätzlich sind einige Komponenten in der Lage, eigene Fehler zu erkennen, z.B. ein Operieren außerhalb der Systemgrenzen. Somit kann auf eine fehlerhafte Komponente mit einer dynamisch angepassten Konfiguration des Systemes reagiert werden. Ein in der PEIS-*Middleware* implementierter Konfigurationsmechanismus, der von jedem PEIS initiiert werden kann, ermöglicht diese reaktive Selbstkonfiguration Gritti, Broxvall und Saffiotti [GBS07].
In der auf der *Peer-to-Peer*-Netzwerkschicht aufsetzenden Ebene werden Funktionalitäten für die Verwaltung und Erhaltung des verteilten *Tuplespaces* sowie Schnittstellen-

spezifikationen, die von den Komponentenprogrammen genutzt werden, implementiert. So ist es z.B. der in der PEIS-*Middleware* enthaltenen Komponente *Tupleview* möglich, alle in der PEIS-Ökologie vorhandenen PEIS-Geräte mit deren Komponenten aufzulisten und den Inhalt des *Tuplespaces* anzuzeigen. Dies unterstützt das *Debugging* und *Monitoring* des Gesamtsystemes.

Der PEIS-Kernel ermöglicht demnach eine Kommunikation und Kooperation zwischen heterogenen Geräten in dynamischen Umgebungen, wobei die zugrunde liegende Heterogenität der Netzwerke verborgen wird. Unterstützt wird dabei ein breites Band an Hardwareplattformen, die von leistungsstarken 64-Bit-Mehrkernprozessoren bis hin zu 8-Bit-Mikrokontrollern reichen. Des Weiteren wird auch eine Vielfalt an Betriebssystemen wie Linux, Windows, MacOS und Embedded Linux bedient. Die Unterstützung für ressourcenbeschränkte eingebettete Systeme erfolgt dabei in Form einer „kleinen“ Version des PEIS-Kernels, dem *Tiny PEIS Kernel* Bordignon u. a. [Bor+07]. Somit wird eine hohe Portabilität dieser *Middleware*-Komponente erreicht. Robotikapplikationen, die diese in C geschriebene Softwarebibliothek nutzen, können in den Programmiersprachen C, LISP und Java implementiert werden. Mit der PEIS-*Middleware* werden auch PEIS-Komponenten für Navigation, Ausführungsüberwachung und Aufgabenplanung, Objekterkennung und Lokalisierung angeboten. Weiterhin werden Schnittstellenfunktionalitäten für Kameras und für die in Abschnitt 2.5 beschriebene Player/Stage/Gazebo-*Middleware* bereitgestellt.
Eingesetzt wird der PEIS-Kernel z.B. bei der Realisierung von intelligenten, meist häuslichen Umgebungen, in denen mobile Roboter und intelligente Haushaltsgeräte den Menschen im Alltag assistieren Saffiotti und Broxvall [SB05] Broxvall u. a. [Bro+06b] Broxvall u. a. [Bro+06a].

2.10 Microsoft Robotics Developer Studio[11]

Das Microsoft Robotics Developer Studio (MRDS) Jackson [Jac07]Microsoft© [Mic11] ist eine windowsbasierte Entwicklungs- und Laufzeitumgebung für das Entwerfen, Ausführen und *Debuggen* von skalierbaren und verteilten Robotikapplikationen. Die Hauptmotivation dieser Entwicklung liegt darin, einen Softwarestandard zur Robotersteuerung für den stark fragmentierten industriellen Robotikmarkt, in dem jeder Hersteller jeweils eigene spezialisierte Software und Programmierschnittstellen verwendet, anzubieten. Im Vordergrund steht dabei auch die Wiederverwendung von Software, sodass ein für einen bestimmten Roboter geschriebenes Programm auch auf einem anderen Roboter mit ähnlichen Eigenschaften verwendet werden kann. Im Kontext des MRDS wird ein Roboter - wie auch in den zuvor beschriebenen Rahmenwerken - als System von Sensoren und Aktuatoren, mit denen kommuniziert werden kann, betrachtet. Die Aufgabe dieses Rahmenwerkes ist es, das Abbilden von entkoppelten Software-Modulen, den sogenannten

[11] www.microsoft.com/robotics/

Services, auf die korrespondierenden Hardware-Komponenten bzw. Robotersubsysteme zu erleichtern. Diese *Services*, über die die Interaktion mit dem Roboter erreicht wird, bilden die Basisbausteine für das Entwickeln von Robotikapplikationen mit dem MRDS.

Die Architektur der serviceorientierten Laufzeitumgebung folgt dem Representational State Transfer (REST) Fielding [Fie00] und setzt auf der Common Language Runtime (CLR) auf. Des Weiteren besteht die Laufzeitumgebung aus zwei Komponenten, nämlich der Concurrency and Coordination Runtime (CCR) und den Decentralized Software Services (DSS).

Roboter, die in der Regel mit mehreren Sensoren und/oder Aktuatoren ausgestattet sind, zeichnen sich durch viele nebenläufige und asynchrone Operationen aus. Das Entwickeln und besonders das *Debugging* solcher parallelen Prozesse erweist sich jedoch meist als schwierig, da komplexe Themen der asynchronen Programmierung wie Koordination der Operationen, Datenkonsistenz und Fehlerbehandlung „manuell“ angegangen werden müssen. Genau an diesem Punkt setzt die CCR, eine *.NET*-Softwarebibliothek, an, indem sie die Komplexität dieser Themen vor dem Anwendungsentwickler durch entsprechende Abstraktionen verbirgt. Die CCR dient somit der Verwaltung asynchroner und nebenläufiger Operationen und bietet dazu ein nachrichtenorientiertes Programmiermodell. Dadurch wird ein Softwaredesign möglich, welches eine lose Kopplung von Software-Modulen erlaubt. Dies bedeutet, dass solche *Services* unabhängig voneinander entwickelt werden können und nur wenige Annahmen über andere Komponenten gemacht werden müssen. Darüber hinaus ergibt sich dadurch auch die Wiederverwendbarkeit dieser Bausteine. Die Kommunikation zwischen solchen lose gekoppelten *Services* erfolgt durch das Senden und Empfangen von XML-basierten Nachrichten an Ports. Diese stellen die Kommunikationsschnittstelle eines *Services* dar. Außerdem ist mit jedem Port eine Warteschlange, die Nachrichten für die Weiterverarbeitung zwischenspeichert, verknüpft. Die Aufgabe der CCR ist es nun, die ankommenden Nachrichten zu koordinieren. Dabei werden die mit den jeweiligen Nachrichtentypen registrierten *Handler*, eine Art *Callback*-Funktion, aufgerufen. Diese Aufgabe übernimmt der sogenannte *Arbiter*, der als Vermittler der Nachrichten fungiert. Dabei kann auch angegeben werden, welche dieser *Handler* parallel laufen können und welche sich gegenseitig ausschließen. Die CCR stellt somit eine Infrastruktur bereit, die eine nebenläufige Ausführung von Aktivitäten ermöglicht.

Eine Applikation zur Robotersteuerung ergibt sich aus der Komposition lose gekoppelter *Services*, die lokal auf einem Netzwerkknoten oder verteilt über ein Netzwerk laufen können. Diese Aufgabe der Orchestration verteilter Software-Module übernimmt die DSS, eine *.NET*-basierte Laufzeitumgebung, die auf der CCR aufsitzt. Dabei bilden das eigens entwickelte Decentralized Software Services Protocol (DSSP) und das Hypertext Transfer Protocol (HTTP) die Grundlage der Interaktion mit und zwischen verteilten *Services*. Mit der Verwendung von DSSP, einem Simple Object Access Protocol (SOAP)-basierten Protokoll, geht ein Event-Modell einher, welches eine *Publish-Subscribe*-Architektur ermöglicht. Der Austausch typisierter Nachrichten zwischen den

Abbildung 2.8: Simulationsumgebung VSE des MRDS

Ports der einzelnen *Services* erfolgt dabei über TCP/IP. Neben dem Nachrichtentransport stellt die DSS auch eine Ausführumgebung bereit, in der die *Services*, die durch ihren Zustand und Operationen auf diesem repräsentiert werden, aufgeführt werden.
Mit dem MRDS werden auch mehrere Basisdienste, die *Runtime Services*, bereitgestellt. Diese bieten z.B. eine Webschnittstelle, mit der über einen Browser sämtliche laufenden *Services* überwacht werden können. Außerdem unterstützen ein *Logging*- und ein Diagnose-*Service* beim *Debugging* und bei der Bewertung der Performanz der entwickelten Robotikapplikation. Darüber hinaus bietet das MRDS sogenannte *generic contracts*. Dabei handelt es sich um Schnittstellenspezifikationen für häufig in Robotiksystemen vorhandene Komponenten wie Sensoren, Motoren und Kameras. Implementiert ein *Service* einen entsprechenden *contract*, kann dieser mithilfe des automatischen Detektionsmechanismus, welcher auf Universal Plug and Play (UPNP) Jenronimo und Weast [JW03] basiert, von anderen *Services* ausfindig gemacht werden. Auf diese Weise wird eine automatische Selbstkonfiguration des Robotiksystemes möglich.
Das MRDS setzt zwingend eine *.NET*-Laufzeitumgebung voraus. Dies bedeutet, dass *Services* auf vielen Robotiksystemen mit limitierten Ressourcen nicht direkt lauffähig sind, was eine erhebliche Einschränkung darstellt. Stattdessen müssen diese Systeme z.B. über eine kabellose Verbindung mit einem Kontrollrechner, auf dem das MRDS läuft, kommunizieren. Das Aufbauen auf der *.NET*-Umgebung bedingt außerdem, dass die *Services* in einer *.NET*-kompatiblen Programmiersprache wie *C#*, *Visual Basic* oder *C++* implementiert werden müssen.

Neben der beschriebenen Laufzeitumgebung beinhaltet das MRDS auch eine Simulationsumgebung, die Visual Simulation Environment (VSE), mit deren Hilfe virtuel-

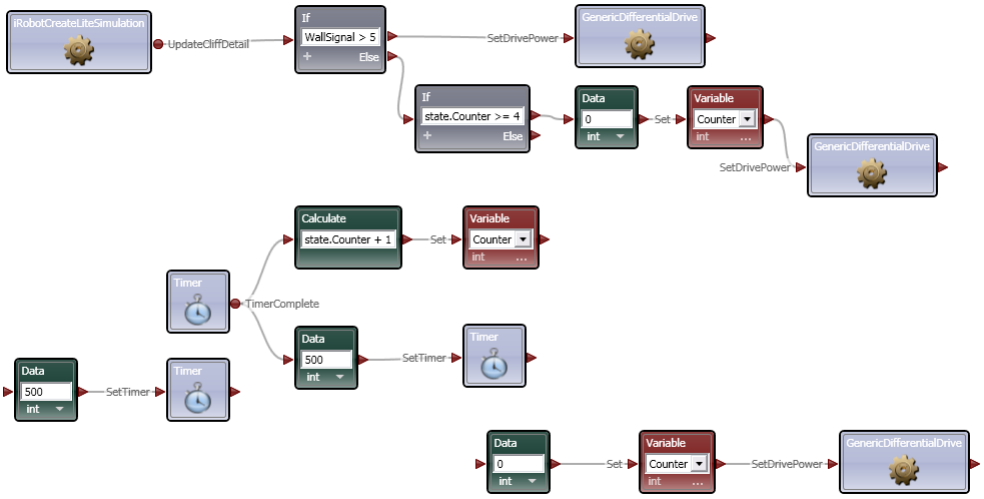

Abbildung 2.9: Grafische Programmiersprache VPL des MRDS

le Welten, in denen realistisch erscheinende physikalische Bedingungen herrschen, erstellt werden können (siehe Abbildung 2.8). Auf diese Weise können z.B. verschiedene Hardware-Setups wie Sensorkonfigurationen zunächst in der Simulation getestet werden, ohne die realen Entsprechungen besitzen zu müssen, was ein *Rapid-Prototyping* ermöglicht. Für den einfachen Einstieg in die Robotikwelt umfasst das MRDS auch eine grafische, datenflussbasierte Programmierumgebung (siehe Abbildung 2.9), die Visual Programming Language (VPL), die dem einfachen Einstieg in die Thematik des Modellierens von Robotikinteraktionen und -steuerungsapplikationen dient.

Zu den in der aktuellen Version (RDS 4) „von Haus aus“ in Form von bereitgestellten *Services* unterstützten Roboterplattformen gehören *iRobot Create und Roomba*, *Lego Mindstorms NXT*, *MobileRobots Pioneer P3DX* und *Parallax Eddie*. Außerdem werden vonseiten der jeweiligen Hersteller auch *Services* für *Parallax Boe-Bot*, *CoroWare CoRobot* und *Explorer*, *Lynxmotion Lynx 6 Robotic Arm* und Roboter der Marke *Robotics Connection* zur Verfügung gestellt.
Mit der aktuellsten Version des RDS hat weiterhin die Unterstützung des Kinect-Sensors Einzug gehalten. Allerdings werden aktuell nur noch Betriebssysteme ab Windows 7 unterstützt.

Einige Projekte, die auf MRDS aufsetzen bzw. die integrierte Simulationsumgebung zur Performanzevualierung verwenden, sind: autonomes Fahrzeug der Princeton Universität für die *DARPA Urban Grand Challenge* [Kor07], Steuerung und Simulation einer Roboter-Fußballmannschaft [Men+08], Verkehrszeichenerkennung [TAB09] und Evaluation einer SLAM-Implementierung [Cos+10].

2.11 Robotic Operating System (ROS)[12]

Die Open-Source-Architektur ROS wurde im Rahmen des STAIR-Projekts an der Universität Stanford und des Personal Robots Program bei Willow Garage entwickelt [Qui+09]. Die Zielstellung, die hinter ROS steht, ist es, einen wiederverwendbaren Code für die Forschung an Robotern zu entwickeln. Die Bezeichnung Betriebssystem ist für ROS eigentlich nicht ganz korrekt, denn es setzt auf bestehende Betriebsysteme wie Windows, Mac OS X oder Linux auf. Es erfüllt jedoch die Aufgaben eines Betriebssystems für den Entwickler, etwa die Hardwareabstraktion und den Nachrichtenaustausch zwischen Programmen, und bringt zahlreiche Programmbibliotheken und einen Paketmanager mit. ROS besteht aus zwei Teilen: zum einen dem Anteil, der als Betriebssystem fungiert (ROS), und zum anderen einer Sammlung von Paketen, die verschiedene Funktionalitäten bereitstellen, etwa für Lokalisierung und gleichzeitiges Kartieren (SLAM) oder zur Simulation. Auf Basis von ROS existieren bereits Arbeiten zur visuellen Aufmerksamkeit. Etwa in [Meg+10] wird ein entsprechendes System genutzt, um die visuelle Suche zu unterstützen.

ROS ist mittlerweile ein sehr verbreitetes Framework - nicht nur in der Zahl der Benutzer, sondern auch in der Anzahl der unterstützenden Roboterplattformen. Ein wesentliches Merkmal ist die Verfügbarkeit einer Werkzeugkette, welche typische Entwicklungsaufgaben wie z.B. Logging und Visualisierung von Systemzuständen unterstützt. Im Bereich der Skalierbarkeit hat ROS bedingt durch seine Kommunikationsinfrastruktur Probleme (siehe dazu: Shakhimardanov u. a. [Sha+11]), welche jedoch momentan durch eine Refaktorisierung der Kommunikationsinfrastruktur gelöst wird.

2.12 RS-Framework

Das RS-Framework wurde von der Abteilung Robotersysteme des Fraunhofer-Instituts für Fabrikbetrieb und -automatisierung für die Verwendung im Zusammenhang mit professionellen mobilen Inspektionsrobotern [Wal+12] entwickelt. Es wird weiterhin als Entwicklungsumgebung für andere Assistenz- und Serviceroboter eingesetzt und unterstützt Forschungs- und Entwicklungsarbeiten an einer Vielzahl von Robotikthemen mit informationstechnischen Schwerpunkten. Motivation der Entwicklung der dem Rahmenwerk zugrunde liegenden Architektur des Datenaustauschs war die Berücksichtigung nicht nur solcher Daten, die für Steuerung und Kontrolle des Roboters benötigt werden, sondern auch die zuverlässige Verarbeitung und Dokumentation von inspektionsrelevanten Daten. Hierfür wird ein Publish/Subscribe-System für spezifischen Nutzdaten zugeordnete Tags in Verbindung mit einem Mechanismus zur Flusskontrolle mit einem Client/Server-Ansatz zur Handhabung und transparenten Persistierung von Zeitserien der eigentlichen Nutzdaten kombiniert. Aus dieser Basis können höherwertige Funktionen wie dynamische

[12] http://www.ros.org/wiki/

Lastverteilung oder Stausteuerung in Form von Mediatoren bei Bedarf punktuell einer Anwendung hinzugefügt werden. Das Framework vereinigt dabei nicht nur hard-, soft- und non-real-time Komponenten zur Datenauswertung und -visualisierung [Wal+07] sowie für Regelungsaufgaben [Wal+09], sondern bietet durch den Einsatz von Mediatoren für dynamisches Quality-of-Service auch Verbesserungspotenzial für klassische Probleme der mobilen Servicerobotik wie beispielsweise der visuellen Navigation[Wal+11].

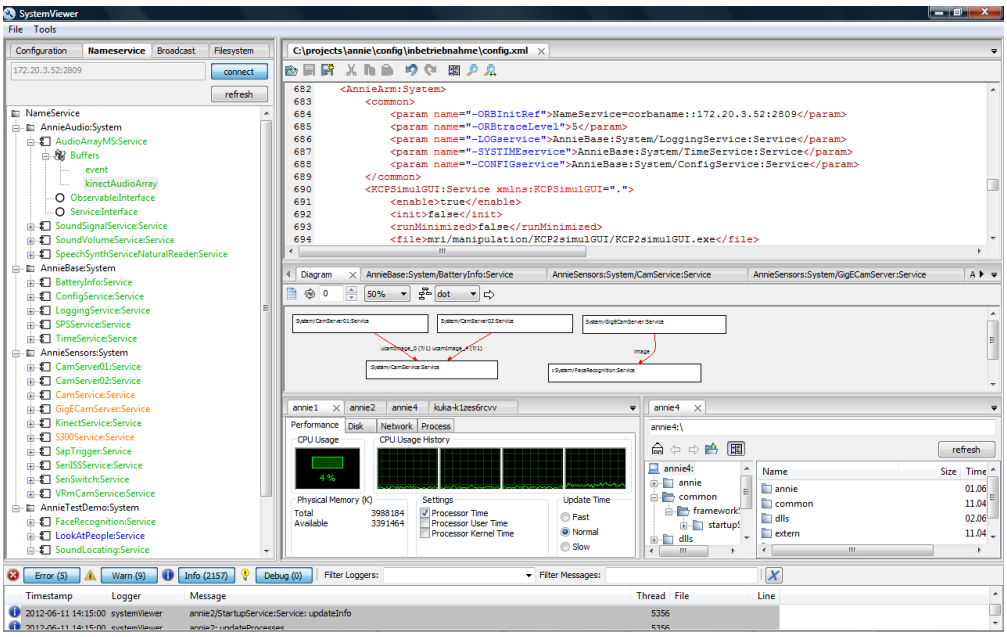

Abbildung 2.10: Grafisches Entwicklungswerkzeug mit laufender Anwendung. Aktive Dienste sind links als Baumansicht dargestellt; die Anwendung selber als XML- bzw. grafische Darstellung rechts oben; Werkzeuge zur Ferndiagnose, zentrale Log-Meldungen und Softwareverteilung rechts unten.

Die Architektur beinhaltet Service-orientierte Elemente wie die Verwendung einiger weniger, generischer Schnittstellen in Verbindung mit komplexen Datentypen wie XML. Das Prinzip der losen Kopplung von Diensten wird in erster Linie durch die Verwendung von über Netzwerk gekoppelter isolierter Prozesse erreicht. Als Kommunikations-Middleware kommt dabei CORBA zum Einsatz. Auf diese Weise wird eine natürliche Skalierung von Anwendungen über mehrere Rechnerknoten oder lokale Prozessoren erreicht. Für Performance-kritische Subsysteme sowie für die Integration kundenspezifischer Anwendungen wird die Verwendung von Diensten als shared-libraries unterstützt. Die Anwendungsentwicklung erfolgt auf höchster Ebene durch das Beschreiben von Anordnung und Kommunikationsbeziehungen solcher Dienste mit Hilfe von XML-Dateien. Die Erstellung von Diensten erfolgt objektorientiert in C++ und wird nicht nur durch

Framework-spezifische Klassenbibliotheken sondern auch durch solche mit reinen Anwendungsfunktionen unterstützt. Abbildung 2.10 zeigt die grafische Oberfläche eines Entwicklungswerkzeugs zur (Offline-) Erstellung und (Online-) Bearbeitung von Anwendungen sowie zu deren Analyse und Diagnose. Ebenfalls enthalten sind hier Werkzeuge zur Analyse von Log- und Statusmeldungen des gesamten Systems, Diagnose und Management von beteiligten Rechnerknoten sowie zur Verteilung und Aktualisierung von Softwaremodulen. Da das Rahmenwerk in erster Linie seine Stärken auf dem Gebiet der Modellierung des Datenflusses zwischen Komponenten sowie der nahe liegenden Systematisierung und Standardisierung von Datentypen, -formaten und Interpretationen hat, beinhaltet die IDE auch ein Plug-In System für die Visualisierung vielfältiger Daten (siehe Abbildung 2.11).

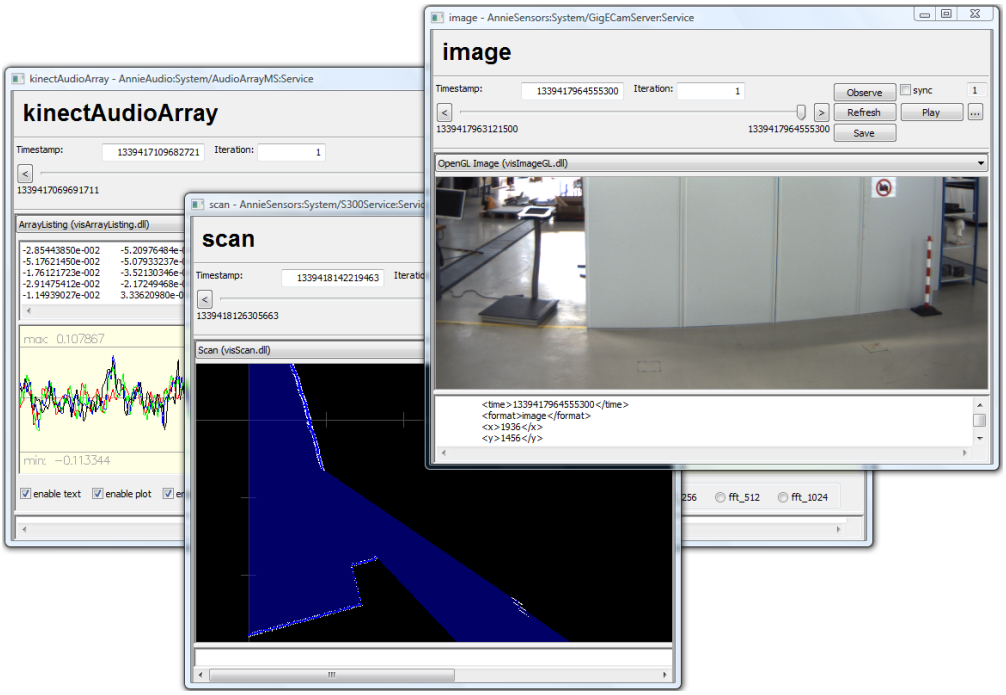

Abbildung 2.11: Plug-In basierte Live-Visualisierung verschiedener Arten von Sensordaten

3 Vergleich

3.1 Vergleichskriterien

Nachdem im vorangegangenen Abschnitt Beispiele für Robotik-Frameworks vorgestellt wurden, widmet sich dieser Abschnitt den Kriterien, anhand derer ein systematischer Vergleich durchgeführt werden kann. Der Kriterienkatalog integriert dabei Abschnitte zur Hardwareunterstützung, fasst die Kommunikationsmechanismen zusammen und betrachtet die Programmierkonzepte einschließlich der angebotenen Tools für das Debugging.

Eine ähnliche Kategorisierung, wenn auch mit anderer Differenzierungen ist bei Kramer und Scheutz [KS07] zu finden. In der vorliegenden Arbeit wurde der Kriterienkatalog auf die Bedürfnisse eines Anwenders zugeschnitten, der ausgehend von einem konkreten Einsatzfall ein geeignetes Framework sucht.

3.1.1 Hardwareunterstützung und Laufzeitumgebung

Robotikapplikationen integrieren vielfältige Sensoren und Aktoren. Um dennoch Applikationen auf verschiedenen Plattformen nutzen zu können, spielen Prinzipien wie Abstraktion und Modularität eine wichtige Rolle.

- **Betriebssystem** Eine Robotikentwicklungsumgebung sollte mehrere Betriebssysteme und dabei eine möglichst umfangreiche Abdeckung für häufig genutzte Bibliotheken, Komponenten oder Hardwaretreiber bieten.

  Weiterhin eignen sich verschiedene Betriebssysteme unterschiedlich gut für die Umsetzung der verschiedenen Subfunktionen eines komplexen Robotersystems. Für Steuer- und Regelungsaufgaben beispielsweise kommen Echtzeitsysteme wie VxWorks oder Windows Embedded Compact (ehemals Windows CE) zum Einsatz, während sich Desktopbetriebssysteme für beispielsweise gut für die Benutzerschnittstelle (HMI) eignen. Entsprechend sind in aktuellen kommerziellen Robotersystemen mitunter Virtualisierungslösungen enthalten, die den parallelen Einsatz verschiedener Betriebssysteme auf dem selben Rechnerknoten ermöglichen. Ein Beispiel hierfür ist die Robotersteuerung KRC4 von Kuka.

- **Unterstützung ressourcenbeschränkter Systeme** Die Interaktion und Kooperation mit ressourcenbeschränkten Systemen ist u.a. in Bereichen gefragt, in denen z.B. die Effizienz und Kosten der eingesetzten Komponenten eine tragende

Rolle spielen. Für ein breites Anwendungsspektrum des jeweiligen Rahmenwerkes ist eine entsprechende Unterstützung solcher Systeme wünschenswert.

- **Echtzeitfähigkeit** Robotikanwendungen umfassen häufig Anwendungselemente, die harte Echtzeitanforderungen an die Verarbeitung stellen. Regelschleifen zur Ansteuerung von Manipulatoren benötigen, um stabil zu laufen, ein deterministisches Zeitverhalten. Entsprechend wird unter diesem Kritikpunkt der Frage nachgegangen, ob und wenn ja in welcher Art und Weise Zeitaspekte in den jeweiligen Ansätzen berücksichtigt werden.

  Sofern möglich, sollte diese Option durch ein Entwicklungsframework mit abgedeckt werden, um Echtzeit und Nicht-Echtzeitanwendungen in einer Umgebung umsetzen zu können.

- **Hardwareunterstützung** Ein Framework sollte nicht nur eine die Vielfalt an Sensoren und Aktoren, sondern auch konkrete Robotersysteme spezifischer Hersteller, die auf diesen Komponenten aufsetzen, unterstützen. Die Hersteller, die hier Plattformen für die Forschung und Ausbildung anbieten, offerieren üblicherweise zumindest entsprechende Hardwareschnittstellen zu ihrer Plattform. Die Angebote sind dabei firmenspezifisch höchst unterschiedlich. Teilweise werden

  1. eine Softwareintegration in Open Source Entwicklungsumgebungen wie ROS (z.B. Kuka Labs mit dem youbot),
  2. eigene Frameworks (z.B. ARIA von MobileRobots; jetzt Adept) und/oder
  3. fertige Anwendungsprogramme für bestimmte Applikationsfelder an (z.B. Metralabs oder Neobotix mit ihren Tools für den Entertainment Bereich)

  geboten. Eine breite Hardwareunterstützung in Kombination mit einer entsprechenden Abstraktion garantiert eine hohe Portabilität und trägt damit zur Beschleunigung des Entwicklungsprozesses bei. In der folgenden Aufstellung wird entsprechend zwischen Sensoren, mobilen Robotern und Manipulatoren unterschieden.

3.1.2 Kommunikation

Die Infrastruktur umfasst Mechanismen, die eine Interaktion und Kooperation zwischen den einzelnen Komponenten des Gesamtsystemes ermöglichen und somit dessen Funktionalität als Ganzes gewährleisten.

- **Kommunikationsmiddleware** Damit Anwendungen verteilt über mehrere Rechnerknoten laufen können und somit eine Ortsunabhängigkeit gewährleisten, sind entsprechende Mechanismen erforderlich. Dies kann z.B. durch ein der Entwicklungsumgebung zugrunde liegendes Rahmenwerk wie CORBA oder in Form spezieller Komponenten wie Namens- oder Verzeichnisdiensten realisiert werden.

- **Kommunikationsprinzip** Das Kommunikationsprinzip beschreibt, in welcher Form der Nachrichtenaustausch zwischen den Komponenten erfolgt und in welcher Beziehung die Komponenten dabei zueinander stehen. Die Interaktion kann z.B. über eine Inter-Prozess-Kommunikation in Form eines gemeinsam genutzten Speichers, über Netzwerkprotokolle und über entfernte Prozeduraufrufe erfolgen. Als Interaktionsmuster sind die *Client-Server*-Beziehung und das *Publish-Subscribe*-Modell denkbar.
- **Echtzeitfähigkeit** Anknüpfend an die Echtzeitfähigkeit der Laufzeitumgebung ist das deterministische Verhalten der Kommunikationsverbindungen Voraussetzung für die Entwicklung zeitkritischer Systeme.

3.1.3 Programmierung

Diese Kategorie bezieht sich auf Kriterien, die einerseits Eigenschaften der konkreten Implementierung der jeweiligen Entwicklungsumgebung beschreiben und andererseits die Anwendungsentwicklung eben mit dieser betreffen.

- **Unterstützte Programmiersprachen** Bei der Anwendungsentwicklung sollte dem Entwickler möglichst die Wahl gelassen werden, in welcher Programmiersprache entwickelt wird. Eine domain-spezifische Frage zielt dabei auf die Möglichkeit der grafischen Programmierung.
- **Unterstützungsbibliotheken** Vordefinierte Komponenten z.B. für Pfadplanung, Verhaltensauswahl und Lokalisierung erleichtern den Entwicklungsprozess und fördern die Wiederverwendung von Software-Modulen, wobei gegebenenfalls entsprechende Anpassungen erforderlich sind.
- **Erweiterbarkeit** Erweiterbarkeit bedeutet hier die Unterstützung für das Hinzufügen neuer Software-Module und neuer Hardware-Komponenten in das bestehende Rahmenwerk.
- **Lizenzmodell** Der Typ der Lizenz der Frameworks bestimmt insbesondere im Fall der kommerziellen Nutzung über deren generelle Anwendbarkeit. Durch das gewählte Lizenzmodell wird die Breite der Entwicklungs-Community zumindest mitbestimmt. Eine aktive Community erleichtert die Entwicklungsarbeit und bieten in Wikis oder Foren eine Vielzahl von Antworten, Anregungen und Beispielcode.

3.1.4 Test und Debugging

Das Testen und Bewerten der implementierten Funktionalitäten nimmt im Entwicklungsprozess von Applikationen eine tragende Rolle ein. Um dem Entwickler in dem iterativen

Tabelle 3.1: Gegenüberstellung der Middlewareimplementierungen 1/IV (Hardware und Kommunikation) (○ = eingeschränkt, ● = umfangreich implementiert)

			CLARATy	CARMEN	MARIE	MIRO	Player
Hardwareunterstützung und Laufzeitumgebung	OS	Windows	○				○
		Linux	●	●	●	●	●
		andere	VxWorks Solaris				Solaris Mac OS X
	Unterst. <32Bit		○				
	Echtzeitfähigkeit		○				
	Sensorinterfaces		○	○	○	●	●
	Bsp.			Scanner GPS	Scanner GPS	Scanner GPS,US Kameras	Scanner Kameras Kinect
	Mobile Roboter		○	○	○	○	●
	Bsp.		Rocky 7/8 FIDO K9	Pioneer ATRV Segway	Pioneer ATRV Segway	Pioneer B21	Pioneer Khepera ATRV ...
	Manipulatoren		○				○
	Bsp.		Dexter				Scara
Kommunikation	Middleware		ACE/TAO	IPC	ACE	ACE/TAO	
	Paradigma		Publish / Subscribe	Publish / Subscribe Client / Server	Mediator	RMI Publish / Subscribe	Sockets Publish / Subscribe
	Echtzeitfähigkeit		○				
	dyn. Erweiterbarkeit		○	●	●	●	●
	Medien neben Ethernet					CAN	

Tabelle 3.2: Gegenüberstellung der Middlewareimplementierungen 2/IV (Hardware und Kommunikation)

OROCOS	YARP	PEIS	MRDS	ROS	RS-Framework	OpenRTM
●	●	●	●	○	●	●
●	●	●		●		●
Xenomai	QSX	TinyOS		Mac OS X	WinCE	Mac OS X
WinCE	QNX	Mac OS X		QSX		VxWorks
		●		●		
●		○			●	●
○	○	○	●	●	●	●
Kraft	Kameras	μC based	Scanner	Scanner	Scanner	Scanner
Drehmom.	Sound	Kameras	Kinect	Kinect	Kameras	Kameras
Kameras		Scanner	Kameras	Kameras	Mikrofone	Kinect
	○	○	○	●		●
	iCub	PeopleBot	Eddie	Pioneer		PeopleBot
			NXT	PR2		NXT
				AR.drone		Segway
●			○	●		●
LBR			AL5A	Katana		PA10
KR 160			KR 60	Robotnik		JACO
KR 361				youBot		
ACE/TAO	ACE		.NET	proprietär	OmniORB	ACE/RTC
	Sockets	Tupelspace	Publish / Subscribe	Publish / Subscribe	Client / Server +	Publish / Subscribe
		Publish / Subscribe			Publish / Subscribe	
●	○				○	●
●	●	●	●	●	○	●
CAN		802.15.4		RS232		
		Bluetooth				

Tabelle 3.3: Gegenüberstellung der Middlewareimplementierungen 3/IV (Programmierumgebung) (○ = eingeschränkt, ● = umfangreich implementiert)

		CLARATy	CARMEN	MARIE	MIRO	Player
Programmierung	Programmiersprachen Bsp.	C++	○ C Java	C++	C++	● C++,Java Python Matlab
	grafische Prog.	● via UML		● RobotFlow		
	Robotikbibliotheken	○	○	●	○	●
	- Navigation	○	●	●		●
	- Sensorprocessing	○	○	○	●	○
	- Bildverarbeitung				●	○
	- Kinematiken					○
	- Aktorregelung	●		○		
	Erweiterbarkeit		○	●	○	●
	Lizenz		GPL	GPL	GPL	GPL
	Open Source	○	●	●	●	●
	Community					●
Debugging	Monitoring		○			●
	Logging		●		●	○
	Simulation	●	○ 2D Maps	● Player Webbots		● 2D/3D phy.Engine

Tabelle 3.4: Gegenüberstellung der Middlewareimplementierungen (Programmierumgebung) 4/IV

OROCOS	YARP	PEIS	MRDS	ROS	RS-Framework	OpenRTM
○ C++ Python	● C++,Java Python Matlab	● C Java LISP	● C# C++ VB	● C++,Java Python Octave	C++	● C++,Java Python
● Simulink			● VPL	○		●
○ ● ●	 ○ ○	○ ○ ○ ○ 	○ ● ○ ○ ○ 	● ● ● ● ○ 	○ ○ ○ 	○ ○ ○
	●		○	●		
GPL LGPL	LGPL	FDL,GPL LGPL		BSD	kommerz.	LGPL
●	●	●		●		●
●	○		○	●		○
●	●	●	○	●	●	●
○	●		○	●	●	●
	● Player	● Player	● 3D phy.Engine	● Player		● 3D phy.Engine

und zyklischen Prozess des Implementierens und anschließenden Testens zu unterstützen, sollte die Entwicklungsumgebung entsprechende Mechanismen dazu bereitstellen.

- **Monitoring** Das *Monitoring* bezieht sich auf die Überwachung einzelner Komponenten und deren Beziehungen zueinander, sodass Aussagen über den Status des Robotersystemes abgeleitet werden können. Eine grafische Schnittstelle, die die Visualisierung einzelner Komponenten, des Gesamtsystemes oder einzelner Parameter übernimmt, vereinfacht die Entwicklungs erheblich.
- **Logging** Das *Logging* der Anwendungsoperation unterstützt einerseits das *Debugging* und ermöglicht andererseits eine Wiederholung dieser Anwendungsausführung im Sinne eines Wiederabspielens einer Aufzeichnung. Somit wird eine Offline-Analyse der implementierten Funktionalitäten möglich, sodass auch Aussagen über die Performance dieser bzw. des Gesamtsystemes getroffen werden können.
- **Simulation** Die Simulation der realen Welt ermöglicht es den Entwicklern, ihre Anwendungen zu testen, ohne die entsprechende Hardware besitzen zu müssen, indem diese geeignet modelliert wird. Die Simulatoren können dabei in Form von „einfachen“ zweidimensionalen bis hin zu komplexen 3-D-Umsetzungen mit realistischen physikalischen Gegebenheiten vorliegen. Ebenso kann ein Zusammenspiel von realen und virtuellen Sensoren und/oder Aktuatoren unterstützt werden.

3.2 Gegenüberstellung

Im Folgenden werden die Frameworks in der Tabelle 3.1 hinsichtlich der Abdeckung der vorgenannten Kriterien kategorisiert. Die tabellarische Darstellung erstreckt sich über jeweils 2 Doppelseiten, in denen die Frameworks in der Reihenfolge ihrer Vorstellung Berücksichtigung finden. Die Klassifikation erfolgte in 3 Stufen. Sofern ein Aspekt einen Schwerpunkt des Frameworks bildet und entsprechend ausgeprägt unterstützt wird dieser mit einem gefüllten Kreis gekennzeichnet. Kriterien, die nur teilweise implementiert sind, wurden mit einem Kreis markiert. Sofern ein Punkt des Kataloges nicht unterstützt wird oder keine Informationen darüber verfügbar waren, bleibt das Feld frei.

Für die Analyse wurden die Dokumentationen, Veröffentlichungen und Codes der Frameworks untersucht. Dabei stand das jeweilige Kernprojekt im Vordergrund, individuelle Implementierungen, die einzelne Bewertungen möglicherweise verändern, wurden nicht berücksichtigt.

4 Zusammenfassung

4.1 Ergebnisse

Mit der Gegenüberstellung der Frameworks in der Tabelle 3.1 werden folgende Tendenzen deutlich:

1. Einige Robotikstandardplattformen (Pioneer, Segway, PeopleBot, usw.) und Sensoren, insb. Laserscanner, verschiedene Kameramodelle und Kinect, werden von einer Vielzahl von Frameworks nativ unterstützt. Vor diesem Hintergrund sollte sich jedes Team, das eine Robotikanwendung entwickelt, kritisch fragen, ob eine individuelle Hardware wirklich erforderlich ist. Der Aufwand für die Bereitstellung der notwendigen Treiber und Schnittstellen ist zum Teil erheblich.

2. Es bestehen mit Frameworks wie Orocos oder OpenRTM sehr spezialisierte Implementierungen, die auf die Echtzeitfähigkeit eines Sensor-Aktor-Systems abzielen. Als Anwendungsszenarien lassen sich anhand der nativ unterstützten Hardware insbesondere Manipulatoren ausmachen. Die Community des dominierenden ROS-Frameworks bemüht sich gegenwärtig, diese Lücke durch die Etablierung eines „ROS-for-Industry“-Stacks zu schließen.

3. Im Hinblick auf die Simulationsumgebung eines Frameworks scheint das Player-Projekt mit seiner Integration in verschiedene andere Frameworks (ROS, PEIS, YARP) eine hohe Akzeptanzrate zu treffen. Nur wenige Robotikframeworks bauen in diesem Punkt auf einer eigenen Implementierung auf (OpenRTM, MRDS).

4. Nur wenige Frameworks setzen neben Ethernet-basierten Verbindungen auch auf anderen Medien auf. Das PEIS-Projekt geht an dieser Stelle am weitesten und stößt mit der Integration von TinyOS Knoten die Tür zu den Sensornetzen auf. ROS bietet mit einem auf dem RS232-Standard aufbauenden Protokoll zumindest eine Einbindung der verbreiteten Arduino-Hardware.

5. Die Aktivitäten der Entwickler-Communities ist unterschiedlich ausgeprägt. Das liegt zum einen daran, das einige Frameworks nicht mehr weiter entwickelt werden, zum anderen aber auch an der insbesondere beim OpenRTM-Projekt deutlich gewordenen lokalen Ausrichtung. Einzelne Foren und Wikis sind zum Beispiel in japanischer Sprache gehalten. Analog kommt dieses Framework insbesondere auf

japanisch/asiatischen Plattformen zum Einsatz, die nicht über den Verbreitungsgrad vergleichbarer Hardware verfügen.

6. Hinsichtlich der Programmiersprachen steht C++ schon wegen der vielfältigen (Standard-)Bibliotheken an erster Stelle. Zudem bieten verschiedene Frameworkimplementierungen bereits `swig` Projekte an, mit denen Schnittstellen für die wichtigsten Programmiersprachen automatisch generiert werden können. Grafische Programmiertools stehen eher im Hintergrund.

Um die unterschiedlichen Stärken der einzelnen Frameworks kombinieren zu können, existiert eine Vielzahl von Interfaces, die für die Integration zum Beispiel von OROCOS oder OperRTM in ROS bereitgehalten werden. Diese Interoperabilität wird in OROCOS möglich da ROS als Basis-„Middleware“ interpretiert wird [SB11, BAK10]. Damit lassen sich dann zum Beispiel mobile Systeme mit echtzeitgesteuerten Manipulationen komfortabel umsetzen.

4.2 Ausblick

Im Rahmen der vorliegenden Arbeit wurden robotikspezifische Frameworks analysiert und gegenübergestellt. Dafür wurde ein Anforderungskatalog erarbeitet der sich als Basis für eine Evaluierung und Klassifikation der Frameworks eignet. Im Ergebnis entstand eine tabellarische Gegenüberstellung, die für Anwendern ein Leitfaden zur Identifikation einer geeigneten Robotik-Middleware sein kann.

Die in [Sma07] aufgeworfene Frage, ob eine einheitliche *Middleware* für alle Anwendungsanforderungen möglich sei, muss auf der Basis dieser Untersuchung Nein beantwortet werden. Dies ist darauf zurückzuführen, dass mit einem Robotiksystem Probleme bzw. Schwierigkeiten wie ein hoher Grad an Heterogenität, begrenzte Ressourcen und eine hohe Wahrscheinlichkeit für das Auftreten von Fehlern einhergehen. Diese Tatsachen machen die Entwickelung eines einheitlichen Frameworks nahezu unmöglich. Allerdings zeigt die zuvor beschriebene Integration von Schnittstellen zu anderen Frameworks einen Weg auf, die Bemühungen zu bündeln. Aktuelle Forschungsprojekte in Europa wie zum Beispiel BRICS[1] und Proteus[2] versuchen die Heterogenität der Frameworks durch modellbasierte Konzepte zu kapseln. Dazu werden Werkzeugketten entwickelt mit denen, unabhängig von bestimmten Frameworks, komponentenbasierte Roboteranwendungen entworfen werden können. Für diese Projekte gilt jedoch auch, dass am Ende Code für ein konkretes Framework generiert werden muss.

[1] http://www.best-of-robotics.org/
[2] http://www.anr-proteus.fr/

Abkürzungsverzeichnis

BFL	Bayesian Filtering Library.
CAN	Controller Area Network.
CCR	Concurrency and Coordination Runtime.
CLR	Common Language Runtime.
CORBA	Common Object Request Broker Architecture.
DSS	Decentralized Software Services.
DSSP	Decentralized Software Services Protocol.
GPS	Global Positioning System.
HTTP	Hypertext Transfer Protocol.
IDL	Interface Definition Language.
IPC	Inter-process Communication.
KDL	Kinematics and Dynamics Library.
MARIE	Mobile and Autonomous Robotics Integration Environment.
MIL	Mediator Interoperability Layer.
MIRO	MIddleware for RObotics.
MRDS	Microsoft Robotics Developer Studio.
OCL	Orocos Component Library.
OROCOS	Open Robot Control Software.
PEIS	Physically Embedded Intelligent Systems.
REST	Representational State Transfer.
RMI	Remote Method Invocation.
RTT	Real-Time Toolkit.

SLAM	Simultaneous Localization and Mapping.
SOAP	Simple Object Access Protocol.
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol.
UPNP	Universal Plug and Play.
US	Ultra Sonic.
VPL	Visual Programming Language.
VSE	Visual Simulation Environment.
XML	Extensible Markup Language.

Glossar

AL5A
Einfacher Manipulator der Firma Lynxmotion Inc.
http://www.lynxmotion.com/c-124-al5a.aspx.

AR.drone
Quadcopter der Firma Parrot
http://ardrone2.paarot.com/.

ATRV
Familie mobiler Roboter der Firma iRobot (nicht mehr am Markt).

B21
Mobiler Roboter der Firma Grinnell More's Real World Interface, Inc. (nicht mehr am Markt).

Eddie
Roboterplattform der Firma Parallax Inc.
http://productinfo/Robotics/EddieRobotPlatform/tabid/942/Default.aspx.

iCub
Humanoider Roboter des EU Projektes RobotCub
http://www.icub.org/index.php.

JACO
Forschungsmanipulator der Firma Kinova
http://kinovarobotics.com/.

Katana
Robotermanipulator der Firma Neuronics
http://http://www.neuronics.ch/cms_de/web/index.php?id=364&s=linux_robot.

Khepera
Familie mobiler Roboter der Firma K-Team's
http://http://www.k-team.com/mobile-robotics-products/khepera-iii.

KR
Baureihenbezeichnung industrieller Manipulatoren der Firma KUKA.

LBR
Manipulator der Firma KUKA Roboter GmbH
http://www.kuka-robotics.com/germany/de/products/addons/lwr/.

NXT
Controller der Mindstorm Baukästen der Firma Lego Group
http://mindstorms.lego.com/en-us/Default.aspx.

PA10
Industrieller Manipulator der Firma Mitsubishi.

PeopleBot
Roboterplattform der Firma ActiveMedia
http://www.mobilerobots.com/ResearchRobots/PeopleBot.aspx.

Pioneer
Familie mobiler Roboter der Firma ActiveMedia
http://http://www.mobilerobots.com/Mobile_Robots.aspx.

PR2
Mobiler Roboter der Firma Willow Garage
http://www.willowgarage.com/pages/pr2/overview.

Robotnik
Manipulator der Firma Robotnik
http://www.robotnik.es/en/products/robotic-arms/modular-robotic-arm.

Segway
Sensorstabilisierte 1-Achsen Transportplattform der Firma Segway Inc.
http://segway.de/.

youBot
Manipulator der Firma KUKA
http://youbot-store.com/ybproductinfo.aspx.

Literatur

[And+05] N. Ando u. a. „RT-middleware: distributed component middleware for RT (robot technology)“. In: *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*. Aug. 2005, S. 3933 –3938.

[BAK10] G. Biggs, N. Ando und T. Kotoku. „Native Robot Software Framework Inter-operation“. In: *SIMPAR'10*. 2010, S. 180–191.

[BSK07] M. Broxvall, B. Seo und W. Kwon. „The PEIS Kernel: A Middleware for Ubiquitous Robotics“. In: *Proceedings of the IROS-07 Workshop on Ubiquitous Robotic Space Design and Applications*. San Diego, CA, USA, Okt. 2007.

[Bea+05] E. Beaudry u. a. „Reactive planning in a motivated behavioral architecture“. In: *Proceedings of the 20th national conference on Artificial intelligence*. Bd. 20. 3. Pittsburgh, USA, Juli 2005, S. 1242–1247.

[Bor+07] M. Bordignon u. a. „Seamless Integration of Robots and Tiny Embedded Devices in a Peis-Ecology“. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007)*. San Diego, CA, USA, Okt. 2007, S. 3101–3106.

[Bro+06a] M. Broxvall u. a. „An Ecological Approach to Odour Recognition in Intelligent Environments“. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2006)*. Orlando, FL, USA, Mai 2006, S. 2066–2071.

[Bro+06b] M. Broxvall u. a. „PEIS ecology: Integrating Robots into Smart Environments“. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2006)*. Orlando, FL, USA, Mai 2006, S. 212–218.

[Bru01] H. Bruyninckx. „Open robot control software: the OROCOS project“. In: *Proceedings IEEE International Conference on Robotics and Automation (ICRA2001)*. Bd. 3. Seoul, South Korea: IEEE, 2001, S. 2523–2528.

[CMG05] T. Collett, B. MacDonald und B. Gerkey. „Player 2.0: Toward a practical robot programming framework“. In: *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*. Sydney, Australien, Dez. 2005.

[C+06] C. Côté u. a. „Robotic Software Integration Using MARIE“. In: *International Journal of Advanced Robotic Systems* 3.1 (März 2006), S. 055–060.

[Car+07] S. Carpin u. a. „USARSim: a robot simulator for research and education“. In: *Robotics and Automation, 2007 IEEE International Conference on*. IEEE. 2007, S. 1400–1405.

[Cos+10] W. Costa u. a. „Evaluation of an ICP Based Algorithm for Simultaneous Localization and Mapping Using a 3D Simulated P3DX Robot“. In: *Proceedings of the Latin American Robotics Symposium and Intelligent Robotics Meeting (LARS 2010)*. São Bernardo do Campo, Brasilien, Okt. 2010, S. 103–108.

[Côt+04] C. Côté u. a. „Code reusability tools for programming mobile robots“. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)*. Sendai, Japan, Sep. 2004, S. 1820–1825.

[ES12] A. Elkady und T. Sobh. „Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography – accepted“. In: *Journal of Robotics* (2012).

[End+01] S. Enderle u. a. „Miro: Middleware for Autonomous Mobile Robots“. In: *Proceedings of 1st IFAC Conference on Telematics Applications in Automation and Robotics*. Weingarten, Deutschland, Juli 2001.

[FMN08] P. Fitzpatrick, G. Metta und L. Natale. „Towards long-lived robot genes“. In: *Robot. Auton. Syst.* 56.1 (2008), S. 29–45.

[FO71] R. J. Feiertag und E. I. Organick. „The Multics Input/Output system“. In: *Proceedings of the third ACM symposium on Operating systems principles (SOSP 1971)*. Palo Alto, CA, USA, Okt. 1971, S. 35–41.

[Fie00] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. dissertation, University of California, Irvine, USA. 2000.

[GBS07] M. Gritti, M. Broxvall und A. Saffiotti. „Reactive Self-Configuration of an Ecology of Robots“. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) Workshop for Network Robot Systems*. Rom, Italien, Apr. 2007, S. 49–56.

[GM02] B. Gerkey und M. Matarić. „Sold!: Auction methods for multirobot coordination“. In: *IEEE Transactions on Robotics and Automation* 18.5 (2002), S. 758–768.

[GVH03] B. Gerkey, R. Vaughan und A. Howard. „The player/stage project: Tools for multi-robot and distributed sensor systems“. In: *11th International Conference on Advanced Robotics (ICAR 2003)*. Coimbra, Portugal, Juni 2003, S. 317–323.

[Gam+95] E. Gamma u. a. *Design patterns: elements of reusable object-oriented software*. Reading, MA, USA: Addison-Wesley Professional, März 1995.

[Gau+03] N. Gaubert u. a. „Emulation of collaborative driving systems using mobile robots“. In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics(SMC 2003)*. Bd. 1. Washington, D.C., USA, Okt. 2003, S. 856–861.

[HLS97] T. Harrison, D. Levine und D. Schmidt. „The design and performance of a real-time CORBA event service“. In: *ACM SIGPLAN Notices* 32.10 (1997), S. 184–200.

[HMS02] A. Howard, M. Matarić und G. Sukhatme. „An Incremental Self-Deployment Algorithm for Mobile Sensor Networks“. In: *Autonomous Robots* 13 (2 2002), S. 113–126.

[HPS04] A. Howard, L. E. Parker und G. S. Sukhatme. „The SDR Experience: Experiments with a Large-Scale Heterogeneous Mobile Robot Team“. In: *Proceedings of the 9th International Symposium on Experimental Robotics (ISER 2004)*. Singapur, Juni 2004.

[IB+12] P. Iñigo-Blasco u. a. „Robotics software frameworks for multi-agent robotic systems development“. In: *Robotics and Autonomous Systems* 60 (2012), S. 803–821.

[JM04] C. Jones und M. Matarić. „Automatic synthesis of communication-based coordinated multi-robot systems“. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)*. Bd. 1. Sendai, Japan, Sep. 2004, S. 381–387.

[JW03] M. Jenronimo und J. Weast. *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play*. Intel Press, 2003.

[Jac07] J. Jackson. „Microsoft robotics studio: A technical introduction“. In: *Robotics & Automation Magazine, IEEE* 14.4 (2007), S. 82–87.

[KS07] J. Kramer und M. Scheutz. „Development Environments for Autonomous Mobile Robots: A Survey“. In: *Autonomous Robots* 22 (2 Feb. 2007), S. 101–132.

[KSB10] M. Klotzbücher, P. Soetens und H. Bruyninckx. „OROCOS RTT-Lua: an execution environment for building real-time robotic domain specific languages“. In: *International Workshop on Dynamic languages for RObotic and Sensors*. 2010, S. 284–289.

[KUG04] G. Kraetzschmar, K. Uhl und G. Gassull. „Probabilistic Quadtrees for Variable-Resolution Mapping of Large Environments“. In: *Proceedings of the 5th IFAC/EURON symposium on intelligent autonomous vehicles (IAV 2004)*. Lissabon, Portugal, Juli 2004.

[Kor07] A. Kornhauser. „DARPA Urban Challenge Princeton University Technical Paper“. In: (2007). Department of Operations Research and Financial Engineering, Princeton University, NJ, USA.

[Kra+00] G. Kraetzschmar u. a. „Integration of multiple representations and navigation concepts on autonomous mobile robots“. In: *Workshop SOAVE 2000: Selbstorganisation von adaptivem Verhalten*. Illmenau, Deutschland, Okt. 2000, S. 1–13.

[Kum] N. Kumaresan. „Miyazaki. K., 2001. Management and Policy over shifts in Innovation Trajectories: The Case of the Japanese Robotics Industry“. In: *Technology Analysis and Strategic Management* 13.3 (), S. 433–462.

[Lem+04] M. Lemay u. a. „Autonomous Initialization of Robot Formations“. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2004)*. New Orleans, LA, USA, Apr. 2004, S. 3018–3023.

[MAJJ08] N. Mohamed, J. Al-Jaroodi und I. Jawhar. „Middleware for Robotics: A Survey“. In: *Proceedings of the IEEE International Conference on Robotics, Automation, and Mechatronics (RAM 2008)*. Chengdu, China, Sep. 2008, S. 736–742.

[MBK07] A. Makarenko, A. Brooks und T. Kaupp. „On the benefits of making robotic software frameworks thin“. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07)*. Bd. 2. San Diego CA, USA, Okt. 2007.

[MFN06] G. Metta, P. Fitzpatrick und L. Natale. „YARP: Yet Another Robot Platform“. In: *International Journal of Advanced Robotics Systems, special issue on Software Development and Integration in Robotics* 3.1 (2006).

[MRT03] M. Montemerlo, N. Roy und S. Thrun. „Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit“. In: *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*. Bd. 3. IEEE. 2003, S. 2436–2441.

[Meg+10] D. Meger u. a. „Curious George: An Integrated Visual Search Platform“. In: Mai 2010, S. 107 –114.

[Men+08] E. Menegatti u. a. „3D Models of Humanoid Soccer Robot in USARSim and Robotics Studio Simulators“. In: *International Journal of Humanoid Robotics* 5.3 (2008), S. 523–546.

[Mic11] Microsoft©. *Microsoft® Robotics Developer Studio User Guide*. Online, 14 November 2011. 2011.

[Mic98] O. Michel. „Webots: a powerful realistic mobile robots simulator“. In: *Proceeding of the Second International Workshop on RoboCup*. 1998.

[Nes+03] I. Nesnas u. a. „CLARAty and challenges of developing interoperable robotic software“. In: *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*. Bd. 3. Okt. 2003, S. 2428 –2435.

[Nes+06] I. Nesnas u. a. „CLARAty: Challenges and steps toward reusable robotic software“. In: *International Journal of Advanced Robotic Systems* 3.1 (2006), S. 23–30.

[Qui+09] M. Quigley u. a. „ROS: an open-source Robot Operating System“. In: *ICRA Workshop on Open Source Software*. 2009.

[Riv05] F. Rivard. *Localisation relative de robots mobiles opérant en groupe*. Québec, Kanada: Département de génie électrique et de génie informatique, Université de Sherbrooke, 2005.

[Rus+07] R. Rusu u. a. „Extending Player/Stage/Gazebo towards cognitive robots acting in ubiquitous sensor-equipped environments“. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) Workshop for Network Robot Systems*. Rom, Italien, Apr. 2007.

[SB05] A. Saffiotti und M. Broxvall. „PEIS ecologies: Ambient Intelligence meets Autonomous Robotics“. In: *Proceedings of the joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies (sOc-EUSAI 2005)*. Grenoble, Frankreich, Okt. 2005, S. 277–281.

[SB11] R. Smits und H. Bruyninckx. „Composition of complex robot applications via data flow integration“. In: *Proceedings of the IEEE international conference on Robotics and Automation (ICRA2011)*. 2011, S. 5576–5580.

[SHK10] A. Shakhimardanov, N. Hochgeschwender und G. K. Kraetzschmar. „Component Models in Robotics Software“. In: *Workshop on Performance Metrics for Intelligent Systems (PerMIS'10)*. Baltimore, USA., Sep. 2010, S. 1030–1035.

[Sha+11] A. Shakhimardanov u. a. „Analysis of software connectors in robotics“. In: *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE. 2011, S. 1030–1035.

[Sie+96] J. Siegel u. a. *COBRA fundamentals and programming*. New York, NY, USA: John Wiley & Sons, Inc., 1996.

[Sma07] W. Smart. „Is a Common Middleware for Robotics Possible?“ In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07) Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*. San Diego, CA, USA, Nov. 2007.

[TAB09] P. Trung, N. Afzulpurkar und D. Bodhale. „Development of vision service in Robotics Studio for road signs recognition and control of LEGO MINDSTORMS ROBOT“. In: *Proceedings of the 2008 IEEE International Conference on Robotics and Biomimetics (ROBIO 2008)*. Bangkok, Thailand, Feb. 2009, S. 1176–1181.

[USM05] H. Utz, F. Stulp und A. Mühlenfeld. „Sharing belief in teams of heterogeneous robots“. In: *RoboCup 2004: The Eight RoboCup Competitions and Conferences* (2005), S. 508–515.

[Utz+02] H. Utz u. a. „Miro - middleware for mobile robot applications“. In: *IEEE Transactions on Robotics and Automation* 18.4 (Aug. 2002), S. 493–497.

[Vol+01] R. Volpe u. a. „The CLARAty architecture for robotic autonomy“. In: *Aerospace Conference, 2001, IEEE Proceedings.* Bd. 1. 2001, 1/121 –1/132 vol.1.

[WS05] D. Wolf und G. Sukhatme. „Mobile Robot Simultaneous Localization and Mapping in Dynamic Environments“. In: *Autonomous Robots Robots* 19 (1 2005), S. 53–65.

[Wal+07] C. Walter u. a. „Handling the Time Delay of Sensor Data Processing in a Model Based Remote Operation Environment“. In: *Proceedings of System 13th IASTED International Conference on Robotics and Applications*. Aug. 2007.

[Wal+09] C. Walter u. a. „Architectural Approach for the Implementation of a Position Control System for a Boat-Like Inspection Robot“. In: *Informatics in Control, Automation and Robotics* (2009), S. 61–73.

[Wal+11] C. Walter u. a. „A QoS enabled visual sensor-system applied to the problem of localizing mobile platforms in indoor environments“. In: *Sensors, 2011 IEEE*. IEEE. 2011, S. 1804–1807.

[Wal+12] C. Walter u. a. „Design considerations of robotic system for cleaning and inspection of large-diameter sewers“. In: *Journal of Field Robotics* 29.1 (2012), S. 186–214.

[Wal99] J. Waldo. „The Jini Architecture for Network-Centric Computing“. In: *Communications of the ACM* 42.7 (1999), S. 76–82.