Objektorientierte Software-Technik für industrielle Steuerungssysteme

Dissertation
der Fakultät für Informatik
der Eberhard–Karls–Universität, Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.–Inform. **Andreas Speck**
aus Tübingen

Tübingen
2000

Tag der mündlichen Qualifikation: 19.07.2000
Dekan: Prof. Dr. Klaus-Jörn Lange
1. Berichterstatter: Prof. Dr. sc. techn. Wolfgang Küchlin
2. Berichterstatter: Prof. Dr. Herbert Klaeren

Vorwort

Für die Betreuung und Unterstützung meiner Arbeit möchte ich mich ganz besonders bei Prof. Dr. W. Küchlin und Prof. Dr. H. Klaeren bedanken.

Ebenso möchte ich Prof. Dr.-Ing. G. Gruhler der Fachhochschule Reutlingen für seine Hilfe danken.

Mein besonderer Dank gilt auch allen Kolleginnen und Kollegen im In- und Ausland für die interessanten Anregungen und hilfreiche Zusammenarbeit.

Tübingen, den 05.06.2000

Andreas Speck

Inhaltsverzeichnis

Kapitel 1

Einleitung

In der industriellen Steuerungstechnik nimmt Software einen immer größeren Stellenwert ein. Aufgaben, die zuvor durch Hardware gelöst worden sind, werden zunehmend von Software-Systemen ausgeführt [Gal95, KGSL97a, Dou98b]. Die Gründe hierfür liegen in den vergleichsweise niedrigen Produktionskosten, der hohen Flexibilität von Software sowie der permanenten Leistungssteigerung von Standard-Hardware.
Im Gegensatz zur Fertigung von Hardware-Bausteinen beschränken sich die Produktionskosten für Software auf die einmaligen Entwicklungs- und eventuelle Anpassungskosten, da Software prinzipiell beliebig oft (verbunden mit nur sehr geringen Kosten) vervielfältigt werden kann.
Während eine Modifikation eines bestehenden Systems aufwendige Änderungen in der Elektronik und darüberhinaus möglicherweise Änderungen in den Produktionsabläufen erfordert, ist eine Anpassungen von Software in Form einer neuen Programmversion mit wesentlich weniger Kosten verbunden.
Parallel zur Verlagerung von Aufgaben von der Hardware in die Software, ist ein Trend zur Verwendung von Standard-Hardware zu beobachten [PTH97].
Diese Neuorientierung hat ihren Ursprung in den hohen (Weiter-)Entwicklungskosten der bisher eingesetzten Spezial-Hardware. Die Firma Bosch hat beispielsweise aufgrund dieser Schwierigkeit die eigenentwickelte Hardware der Robotersteuerung rho3 über einen Zeitraum von beinahe zehn Jahren kaum verändert. Führende Roboterhersteller wie die Firmen Bosch und Kuka haben den Übergang von proprietären Steuerungsrechnern zu PC-basierten Robotersteuerungen inzwischen vollzogen. Neben Robotersteuerungen werden auch andere industrielle Steuerungstypen auf PC-Plattformen portiert. So bietet die Firma Siemens sowohl speicherprogrammierbare Steuerungssysteme (Simatic) als auch numerische Steuerungen (Sinumeric) nicht mehr nur auf spezieller Hardware an, sondern auch als Portierung auf Siemens Industrie-PCs (Sicomp).
Trotz der wachsenden Zahl von Software-Lösungen auf standardisierter Hardware, wurde die Rolle und Bedeutung der Software-Technik auf diesem Gebiet bisher kaum erkannt. Das vollständige Potential flexibler Software-Lösungen auf kostengünstiger und leistungsfähiger Standard-Hardware kann jedoch nur durch die konsequente Anwendung der Prinzipien der Software-Technik ausgeschöpft werden. Parallel zur Situation in den 60iger Jahren [McI76, Bau93] droht anderenfalls mit zunehmender Komplexität und wachsender Software-Funktionalität eine neue Software-Krise im Bereich der industriellen Steuerungstechnik. Die Bedeutung der Software-Technik und wohlüberlegter Software-Gestaltung ist bereits seit langem bekannt und nach wie vor Gegenstand aktueller Forschungsarbeiten [Par72, LH89, Moi99].
Eine Übertragung dieses Wissens in das Gebiet der Steuerungstechnik ist ein konsequenter

und notwendiger Schritt. Hierzu ist interdisziplinäres Wissen in beiden Bereichen, sowohl der Informatik (objektorientierte Software-Technik, Echtzeitsysteme) wie auch der Automatisierungstechnik (Feldbussysteme, Steuerungstechnik) Voraussetzung. Diese Arbeit leistet einen Beitrag zum Schließen der Lücke zwischen Software-Technik und Automatisierungstechnik. Dazu sind zum einen bewährte Techniken der objektorientierten Software-Entwicklung unter Verwendung von Standard-Hardware auf die Domäne der Steuerungstechnik angewendet worden [KGSL97a]. Zum anderen sind neue Konzepte zur Entwicklung und Wiederverwendung von Software erarbeitet und verifiziert worden, die teilweise anwendungsgebietsübergreifende Bedeutung haben [SP00, PRST99]. Das Ergebnis dieser Arbeiten ist ein universelles, wiederverwendbares Hochleistungssteuerungssystem der nächsten Generation, bzw. Konzepte zu dessen Entwicklung und Wiederverwendung, sowie verschiedene konkrete Implementierungen eines solchen Steuerungssystems. Es ist universell, da es verschiedene Steuerungstypen (z.B. Robotersteuerung, einfache numerische Steuerung und speicherprogrammierbare Steuerung) umfaßt. Durch die Wiederverwendbarkeit kann es auf unterschiedliche Standardplattformen portiert werden, die deutlich leistungsfähiger sind als die bisheriger Steuerungen.

Der folgende Abschnitt diskutiert zunächst die Anforderungen an die neue Generation industrieller Steuerungssysteme. Diese stellen die Ausgangsbasis und Zielrichtung für alle folgenden Arbeiten dar. Anschließend folgt eine Übersicht über die in dieser Arbeit erstellten neuen Konzepte.
Kapitel 2 führt in die Grundlagen dieser Arbeit und den Stand der Forschung ein: objektorientierte Muster, Frameworks und Komponenten, sowie objektorientierte Vorgehensweisen zur Entwicklung von Echtzeitsystemen. Ein Überblick über industrielle Steuerungssysteme schließt das Kapitel ab.
Auf dieser Basis wird in Kapitel 3 die Entwicklung einer objektorientierten, universellen Robotersteuerung in den einzelnen Phasen Konzeptualisierung, Analyse und Design vorgestellt.
In Kapitel 4 werden drei Ansätze zur Wiederverwendung der Architektur und des Codes beschrieben. Dies sind zum einen ein Architekturmuster für industrielle Steuerungssysteme sowie ein Framework. Zum anderen wird ein in einer internationalen Zusammenarbeit entwickeltes domänenübergreifend verwendbares Architekturmodell zum Bau komponentenbasierter Frameworks sowie ein Steuerungs-Framework, das nach diesem Vorbild entwickelt worden ist, vorgestellt.
Ein Simulations- und Monitoring-Werkzeug für Robotersysteme wird in Kapitel 5 beschrieben. Dieses System basiert auf dem im vorhergegangenen Kapitel eingeführten komponentenbasierten Framework.
Kapitel 6 zeigt einige zentrale Implementierungsdetails der verschiedenen Realisierungen des Steuerungssystems und vergleicht die Leistungsdaten von Implementierungen auf ausgewählten Plattformen.
Abschließend werden die zentralen Konzepte zusammengefaßt und weiterführende Forschungsbereiche aufgezeigt.

1.1 Anforderungen an industrielle Steuerungssysteme

Mit dieser Arbeit wird die Entwicklung einer neuen Generation von Steuerungssystemen durch Anwendung von Prinzipien der Software-Technik vorgestellt. Die Konzepte hierzu und deren Implementierungen sind geleitet von den Anforderungen, die zu Beginn der Arbeit als

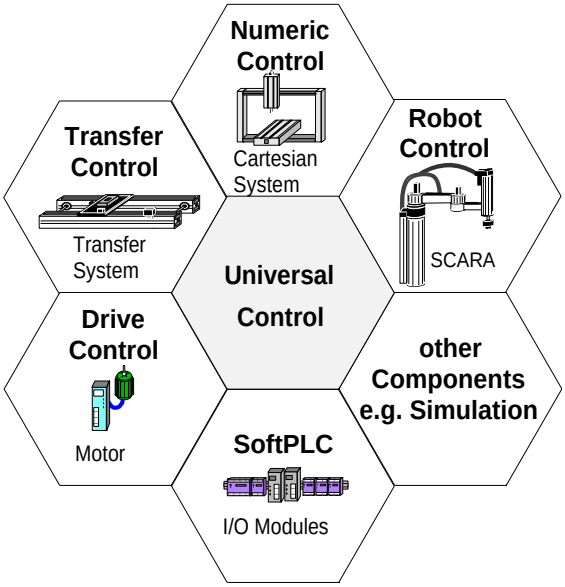


Abbildung 1.1: Universelles Steuerungssystem

charakteristische Eigenschaften für Steuerungssysteme der nächsten Generation ausgemacht worden sind. Sie orientieren sich an der Leistungsfähigkeit sowie insbesondere an den Beschränkungen der zur Zeit marktgängigen Steuerungssysteme (z.B. Robotersteuerung Kuka KR C1).
Damit dient diese Zusammenstellung als Aufgabenbeschreibung (Problem Statement) [Boo94] für diese Arbeit.

- Universelle Steuerung
  Ein universelles Steuerungssystem kann (im Gegensatz zu heute marktgängigen Steuerungen) verschiedene Typen von Automatisierungsgeräten steuern. Auf einem Basissystem ist die Ausführung beliebiger Steuerungsfunktionen möglich.

  Als Nachweis der universellen Funktionalität sollen für diese Arbeit unter anderem ein Horizontalknickarmroboter, ein Portalsystem und verschiedene periphere Geräte, wie z.B. E/A Module oder einfache Antriebe angesteuert werden.

- Einheitlichkeit
  Da eine universelle Steuerung unterschiedliche Steuerungsfunktionen in einem System (auf einer einheitlichen Hardware) vereint, ist dadurch die Basis für eine einheitliche Programmierumgebung geschaffen. Die verschiedenartige Programmierung der Steuerungstypen (z.B. Roboterprogrammierung, SPS-Programmierung und NC-Programmierung) wird von einer modernen Steuerung durch eine konsistente, einheitliche Umgebung unterstützt. Anweisungen zur Ansteuerung verschiedener Gerätetypen sollen in einem Anwendungsprogramm gemischt werden können.

- Portierbarkeit und Wiederverwendbarkeit
  Eine weitere Anforderung an moderne Steuerungssysteme ist die Portierbarkeit und damit deren partielle oder vollständige Wiederverwendbarkeit in verschiedenen Kontexten. Eine Steuerung soll plattformübergreifend eingesetzt werden können. Ziel dieser Arbeit ist daher die Erstellung und Implementierungen eines Entwurfs einer industriellen Steuerung, der auf unterschiedlichen Standardplattformen implementiert werden

kann, sowie die Erarbeitung und Verifikation geeigneter Konzepte zur Unterstützung der Wiederverwendbarkeit. Als Standardplattformen werden off-the-shelf Rechner wie Workstations oder PCs und ihre industrietauglichen Pendants betrachtet, auf denen POSIX-kompatible oder ähnliche Betriebssysteme ablaufen.

- Offene Steuerung
  Eine offene Steuerung ist für Erweiterungen (z.B. weitere Steuerungsfunktionen, alternative Geräteanbindungen oder die Steuerung zusätzlicher Gerätetypen) offen. Dies bedeutet, daß die Steuerung in der Lage ist, eine unterschiedliche Anzahl von Geräten zu kontrollieren. Eine dynamische Änderung der Zahl der Peripheriegeräte ist wünschenswert, wobei die Konfiguration des verwendeten Steuerungsrechners entsprechend der Menge der zu kontrollierenden Geräte möglich sein sollte.

- Multitasking
  Eine Steuerung der nächsten Generation unterstützt Multitasking, d.h. mehrere Steuerungsanwendungen können auf ihr konkurrent ablaufen. Neben einer Task zur Steuerung eines Roboterarms ist beispielsweise die zeitgleiche Ausführung von Tasks für die Kontrolle weiterer peripherer Geräte möglich.

- Leistungsfähigkeit
  Eine Steuerung der nächsten Generation muß einen Fortschritt auf dem Gebiet der Leistungsfähigkeit gegenüber bereits existierenden Systemen mitsichbringen. Mehr parallel ausgeführte Tasks pro Zeiteinheit bei gleichzeitig schnellerer Bearbeitung wird erwartet. Dies kann sich als eine automatische Konsequenz der Multitasking-Fähigkeit und der Portierbarkeit ergeben [1]. Die Portierbarkeit der Steuerung auf Hardware (Workstations und PCs), die gegenüber traditioneller Steuerungs-Hardware wesentlich leistungsfähiger ist, macht eine deutliche Leistungssteigerung des gesamten Steuerungssystems möglich.

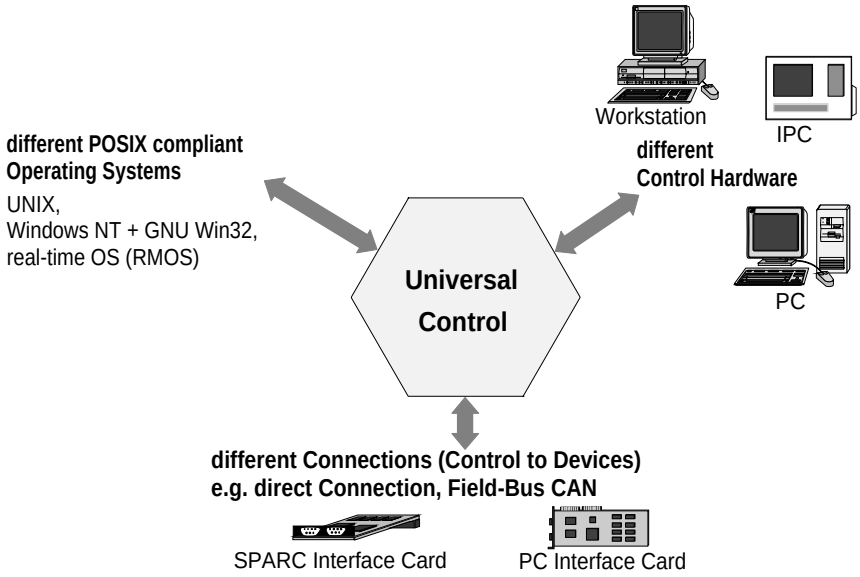


Abbildung 1.2: Plattformunabhängiges System

[1] Dies trifft insbesondere auf die in dieser Arbeit entwickelten universellen industriellen Steuerungssysteme zu.

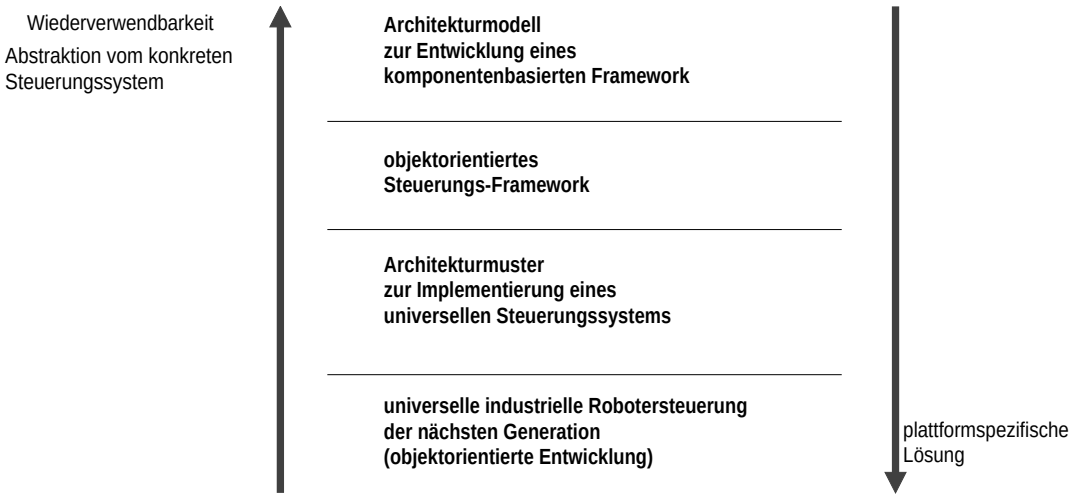


Abbildung 1.3: Übersicht über die Arbeit

1.2 Ansätze zur Entwicklung neuer Steuerungssysteme und Gliederung der Arbeit

Ziel der Arbeiten und Ergebnisse, die im folgenden dokumentiert sind, ist die Entwicklung bzw. die Bereitstellung geeigneter Konzepte und Implementierungen zur Entwicklung einer industriellen Steuerung der nächsten Generation, d.h. einer Steuerung, die die in Abschnitt 1.1 beschriebenen Anforderungen erfüllt.
Mit vier Ansätzen ist dieses Ziel in dieser Arbeit verfolgt (siehe Abbildung 1.3).
In einem ersten Ansatz realisiert diese Arbeit ein derartiges Steuerungssystem (siehe Kapitel 3). Im Gegensatz zu bisherigen Steuerungssystemen [Mic90, RNS93, McK95], nehmen in diesem Ansatz erstmalig Software-Technik Prinzipien einen zentralen Stellenwert ein. Durch diese Entscheidung ist die resultierende objektorientierte universelle Steuerung wegweisend für den Bereich der Steuerungstechnik. Mit HighRobot [KGSL97a] ist die erste universelle Robotersteuerung, die die Anforderungen aus dem vorherigen Abschnitt erfüllt, veröffentlicht. Andere vergleichbare Ansätze zur Entwicklung eines Steuerungssystems unter Anwendung objektorientierter Software-Technik erfüllen nicht die Anforderungen an Steuerungen der nächsten Generation [OSA96, Chr94] und umfassen darüberhinaus teilweise keine Implementierungen zu deren Verifikation [Chr94] oder sind auf einen Steuerungstyp beschränkt (z.B. SPS [DMM98]).
Während der erste Ansatz durch die Anwendung objektorientierter Software-Technik eine Neuerung auf dem Bereich der Steuerungstechnik hervorbringt, führen die folgenden Ansätze zu Neuerungen sowohl auf dem Bereich der Steuerungstechnik, wie auch auf dem Bereich der Software-Technik. Zielrichtung ist hierbei die Unterstützung der Wiederverwendbarkeit auf konzeptueller Ebene wie auch, sofern möglich, auf Implementierungsebene. Die Übertragung und Anpassung der Steuerung auf verschiedene Kontexte soll möglichst einfach und flexibel werden, um den Anforderungen *Portierbarkeit und Wiederverwendbarkeit*, *offene Steuerung*, *Leistungsfähigkeit* gerecht zu werden.
Mit dem zweiten Ansatz wird dazu ein Architekturmuster zur Entwicklung eines universellen objektorientierten Steuerungssystems eingeführt [KGSL97a] (siehe Abschnitt 4.1). Die Erfahrungen aus den ersten objektorientierten Entwicklungen sind auf diese Weise auf abstrakter

Ebene dokumentiert und dabei zu einer wiederverwendbaren Standardlösung überarbeitet und angepaßt worden. Die Übertragung der Erfahrungen auf unterschiedlichste Kontexte wird dadurch unterstützt und hat sich im Verlauf der Entwicklungsarbeiten (Portierungen auf verschiedenste Plattformen) bewährt [SP00]. Die Möglichkeiten, die sich durch den Einsatz von Architekturmustern auch auf dem Gebiet der Steuerungstechnik ergeben, sind bisher kaum erkannt worden. Vereinzelte Ansätze zur Verwendung von Mustern auf dem Echtzeit- und Steuerungsbereich wie z.B. [BMR+96, Sel96] beschreiben Design-Lösungen für ausgewählte Teile einer industriellen Steuerung, geben aber keine Gesamtlösung im Sinne einer Architektur.
Mit einem in dieser Arbeit entwickelten objektorientierten Steuerungs-Framework (siehe Abschnitt 4.2) wird neben der Wiederverwendung der abstrakten Architektur auch die Wiederverwendung auf Code-Ebene ermöglicht. Durch die nach dem Verfahren von H. A. Schmid [Sch97a, Sch98] identifizierten und implementierten variablen Elemente (Hot Spots) kann der Steuerungs-Code auf spezifische Gegebenheiten angepaßt werden. Das Framework ist in dieser Arbeit mehrfach zur Entwicklung von Steuerungen mit unterschiedlicher Funktionalität erprobt worden [SP00]. Es kann im Gegensatz zum Architekturmuster nur zur Entwicklung von Steuerungen, die auf ähnlichen Plattformen laufen sollen, verwendet werden, bietet für diesen Zweck aber weitergehende Unterstützung. Das in dieser Arbeit entwickelte Steuerungs-Framework basiert auf dem Architekturmuster und realisiert die Anforderungen aus Abschnitt 1.1. Analog zu Architekturmustern wird das Konzept der Frameworks aus dem Bereich der Software-Technik bisher kaum für die Entwicklung von Steuerungssystemen genutzt.
Eine weitere Abstraktionsstufe ist durch einen vierten Ansatz realisiert. In einer internationalen Zusammenarbeit sind die gemeinsamen Strukturen unterschiedlichster Frameworks identifiziert worden. Diese Strukturen sind in einem domänenübergreifend verwendbaren Architekturmodell zur Entwicklung von komponentenbasierten Frameworks herausgearbeitet und verfeinert [PRST99, Spe00] (siehe Abschnitt 4.3.1).
Bereits [Pre97, Szy97] führen komponentenbasierte Frameworks ein. Während diese wie alle übrigen existierenden Frameworks [Pre94, FS97, Sch98] eine spezifische Architektur implementieren und sich nur in einzelnen Details an allgemeingültigen Prinzipien ausrichten (z.B. an allgemeinen Techniken zur Implementierung von Hot Spots [Pre94, Pre97]), gibt das neu eingeführte Architekturmodell unabhängig zu einer spezifischen Architektur Orientierungshilfe zur Entwicklung von komponentenbasierten Frameworks in Form einer generischen Architektur für solche Frameworks. In diesem Sinne stellt das Architekturmodell ein Muster zur Entwicklung von komponentenbasierten Frameworks dar [PRST99].

Ein komponentenbasiertes Steuerungs-Framework ist nach diesem Vorbild entstanden (siehe Abschnitt 4.3). Es zeichnet sich dadurch gegenüber dem rein konventionell entwickelten Framework durch eine modularere Struktur und eine sich hieraus ergebende höhere Flexibilität aus. Die iterative Erweiterbarkeit anstelle frühzeitiger Festlegung aller variablen Elemente setzt den in [Cop99] propagierten Ansatz mit Namen “Piecemeal Growth” um.

Das komponentenbasierte Steuerungs-Framework wird von einem Simulations- und Monitoring-Werkzeug RoboSiM [Spe98, Spe99, SK99] für Robotersysteme ergänzt (siehe Kapitel 5). Im Gegensatz zu bestehenden Robotersimulationen [MM95, AS94, Bic94a, KR94] simuliert RoboSiM virtuelle Roboterarme ohne Verwendung zusätzlicher Hardware. Steuerung, Simulation und Monitoring werden auf derselben Steuerungsplattform ausgeführt. Darüberhinaus erlaubt RoboSiM gleichzeitiges Monitoring realer Geräte und die Simulation virtueller Einheiten und unterscheidet sich hierin von allen anderen Robotersimulations- und Monitoringsystemen.

Kapitel 2

Umfeld der Arbeit und Stand der Forschung

In diesem Kapitel wird in die Grundlagen dieser Arbeit eingeführt. Die Beschreibungen über objektorientierte Muster, objektorientierte Frameworks und Software-Komponenten sowie die objektorientierte Entwicklung von Echtzeitsystemen geben im wesentlichen den aktuellen Stand der Forschung wieder. Sämtliche Gebiete werden erst seit Beginn der 90iger Jahre intensiv bearbeitet und stehen nach wie vor im Brennpunkt der aktuellen Forschung [SFJ96, FS97, Szy97, Sel99].
Die abschließende Einführung in industrielle Steuerungssysteme stellt hauptsächlich Grundlagenwissen vor. Lediglich die in Abschnitt 2.4.5 beschriebenen offenen Steuerungssysteme und neuen Ansätze zum Bau von Steuerungssystemen stellen den aktuellen Stand der Forschung dar. Kurz angesprochen werden die Forschungsgebiete über den Einsatz paralleler Hardware als Robotersteuerungen und die Verwendung objektorientierter Programmiersprachen für industrielle Steuerungssysteme.
Entsprechend der Aufgabenstellung und damit des Ziels dieser Arbeit (siehe Abschnitt 1.1) diskutieren die folgenden Kapitel die aktuellen Entwicklungen auf den Forschungsgebieten, die zu neuen Konzepten für den Bau modularer, universeller und portierbarer industrieller Steuerungssysteme in Beziehung stehen. Die Forschung im Bereich der Entwicklung neuer Steuerungsalgorithmen oder Steuerungsverfahren wird nicht betrachtet, da dies nicht das Ziel der Arbeit ist.

2.1 Objektorientierte Muster (Patterns)

In den etablierten Ingenieurwissenschaften, wie Maschinenbau, Elektrotechnik, Bauingenieurwesen und Architektur werden Erfahrungswerte aus Handbüchern verwendet. Diese Erfahrungen basieren auf immer wieder erfolgreich erprobten Lösungen.
Bei der Konstruktion von Maschinen wird auf Konstruktionskataloge zurückgegriffen, in denen Muster für Standardteile gesammelt sind.
In der Elektrotechnik werden Bauteile anhand bekannter Muster zu größeren Einheiten (Baugruppen) kombiniert. Zusätzlich beschreiben Sammlungen von Abschätzungen das Verhalten der Baugruppen. Durch diese Abschätzungen lassen sich mit weitaus geringerem Rechenaufwand Lösungen finden. Zum Teil werden die numerischen Berechnungen wegen deren Rundungsfehlern sogar übertroffen.
Ein Fernziel des Software Engineering ist die Entwicklung von Handbüchern als Sammlungen wiederverwendbarer Erfahrungen. Objektorientierte Muster sind ein Schritt in diese

Richtung. Sie ermöglichen zum einen, daß erfolgreiche Architekturentwürfe wiederverwendet werden können. Zum anderen erleichtern Muster die Dokumentation von Software, da Muster zur Beschreibung der Architektur verwendet werden können. [SFJ96]

2.1.1 Entwicklungsgeschichte

Im Software Engineering ist das Vorgehen, Muster zur Entwicklung von komplexen Systemen zu verwenden, von anderen ingenieurwissenschaftlichen Disziplinen übernommen worden. Das Konzept bei der Entwicklung von Systemen auf wiederverwendbare Muster zurückzugreifen, ist zuerst von C. Alexander et al. 1977 [AIS+77] formuliert worden:

> "Each pattern describes a problem which occurs over and over again in our environment, and describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing the same thing twice."

Alexander bezieht sich dabei auf die Entwicklung von Gebäuden und Städten, aber seine Aussage kann als grundlegende Definition von Mustern in der objektorientierten Software-Entwicklung verwendet werden [GHJV94].

Das erste allgemein verwendbare objektorientierte Muster war das 1988 von G. E. Krasner und S. T. Pope veröffentlichte Model View Controller Muster [KP88]. Dieses Muster beschreibt ein Design zur Entwicklung von graphisch interaktiven Systemen. Ursprünglich war es nur für Smalltalk-80 vorgesehen, aber die Model View Controller Architektur kann auch durch andere objektorientierte Programmiersprachen wie z.B. C++ oder Java realisiert werden.

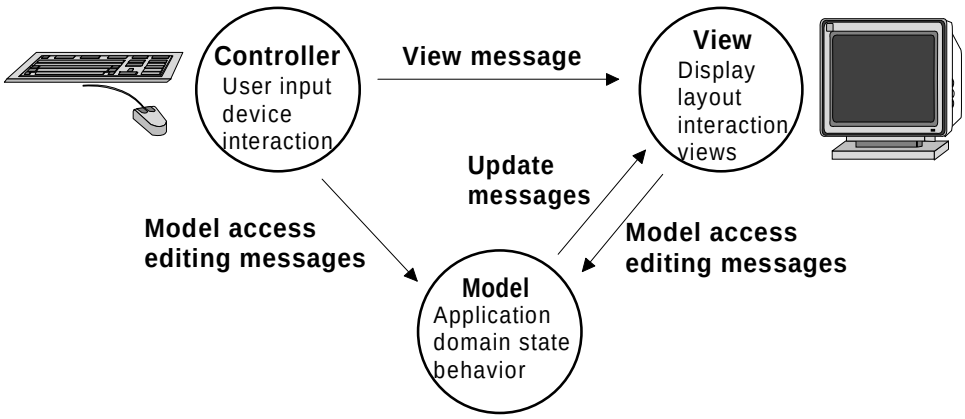


Abbildung 2.1: Model View Controller Muster [KP88]

Krasner und Pope ordnen die Klassen einer graphischen Benutzeroberfläche in drei Gruppen ein: *Model*, *View* und *Controller* Klassen (siehe Abbildung 2.1). Dabei übernehmen diese Gruppen von Klassen unterschiedliche Aufgaben:

- Das *Model* beinhaltet die Kernfunktionalität und die zentralen Daten des Systems. Beispielsweise werden in einem einfachen Informationssystem alle Informationen im *Model* gespeichert. Zusätzlich sind die Algorithmen zur Verarbeitung der Informationen Teil des *Model*.

- Die *View* Komponenten zeigen dem Benutzer die Informationen auf dem Bildschirm an. Dazu erhält der *View* vom *Model* die notwendigen Daten. Im System können mehrere Varianten des *View* vorkommen. Unterstützt z.B. ein View immer ein bestimmtes Window, so können mehrere Windows gleichzeitig aufgeblendet sein.

- Der *Controller* übernimmt die Kommunikation mit der Umgebung. Diese Klassen nehmen die Kommandos des Benutzers entgegen und kommunizieren mit anderen Geräten, z.B. über ein lokales Netz mit weiteren Rechnern. Die Ereignisse, die von außen empfangen werden, werden in Anfragen umgesetzt und an die anderen Komponenten im System weitergegeben.

Insgesamt bleiben Krasner und Pope bei der Erklärung ihres Muster auf einer sehr abstrakten Ebene. Inzwischen gibt es wesentlich konkretere Entwurfsbeschreibungen des Model View Controller Musters, wie z.B. in [BMR+96].

Nach 1988 sind weitere objektorientierte Muster beschrieben worden: P. Coad konzentriert sich z.B. hauptsächlich auf Muster zur Bildung eines Objektmodells in der objektorientierten Analyse [Coa92]. J.O. Coplien veröffentlichte 1992 Muster zur Codierung, bzw. Richtlinien für die objektorientierte Software-Entwicklung [Cop92].
Auf der Konferenz für objektorientierte Programmierung, Systeme, Sprachen und Applikationen 1991 (Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'91) gab es einen ersten Workshop mit dem Ziel, eine Sammlung von Mustern zusammenzustellen. Vorbild für die Sammlung von objektorientierten Mustern waren verschiedene Kataloge mit Beschreibungen von Algorithmen wie beispielsweise *The Art of Computer Programming* von D. E. Knuth [Knu73]. Aus dem Workshop und weiteren Konferenzen entstand unter anderem 1994 die erste Sammlung von objektorientierten Mustern als Buch von E. Gamma, R. Helm, R. Johnson und J. Vlissides [GHJV94]. Die Muster in diesem Katalog sind vom Framework ET++, das zur Entwicklung von graphischen Benutzeroberflächen dient, abgeleitet worden. Seitdem sind weitere Musterkataloge von anderen Autoren herausgegeben worden, wie z.B. [Pre94, BMR+96, CNM95]. Die Mitgliederzeitschrift *Communications of the ACM* (CACM) der amerikanischen Computer-Gesellschaft (Association for Computing Machinery, ACM) hat 1996 dem Thema objektorientierte Muster ein Sonderkapitel (Special Issue on Software Patterns) [Spe96] gewidmet.
Die ersten Muster, bzw. Mustersammlungen, hatten ihren Schwerpunkt auf der objektorientierten Modellierung von Systemen (objektorientierte Analyse) [Coa92] und dem objektorientierten Design sowie der objektorientierten Programmierung, wie z.B. in [GHJV94, Pre94]. Inzwischen werden Muster zunehmend zur Steigerung der Effizienz und Zuverlässigkeit von Software sowie zur Entwicklung von parallelen und verteilten Systemen vorgestellt [CNM95, BMR+96, Cop96, Sch96a].

2.1.2 Definitionen objektorientierter Muster

Objektorientierte Muster sind Lösungsvorlagen für Problemstellungen, die immer wiederkehren. Dabei beschreibt ein Muster nur den abstrakten Lösungsansatz für das Design oder die Architektur eines Systems. Von einem Muster können immer wieder neue, einer konkreten Problemstellung angepaßte, Lösungsvarianten abgeleitet werden.
Eine grundlegende Definition objektorientierter Muster findet sich in den Aussagen von Alexander (siehe Kap. 2.1.1). Folgende weitere allgemeingültige Aussagen können über Muster gemacht werden:

- Muster sind abstrakte Standardlösungen:
  Muster sind nur abstrakte Beschreibungen eines Lösungsansatzes [GHJV94, BMR+96]. Im Gegensatz zu Frameworks enthalten sie keine spezifischen Implementierungen.

  Muster sind wiederverwendbare Lösungen für Standardprobleme. Sie entstehen durch Abstraktion mehrfach erfolgreich verwendeter Architekturen. Dadurch daß Muster Standardlösungen bieten, tragen sie auch zur besseren Dokumentation von komplexen Problemlösungen bei.

- Muster sind eine Erweiterung der bisherigen objektorientierten Analyse und Design Methoden:
  Objektorientierte Muster machen bestehende Methoden und Techniken nicht überflüssig, sondern bieten Lösungen für bestimmte, immer wiederkehrende Aufgabenstellungen. Damit erfüllen Muster eine Funktion, die von bereits existierenden Methoden nicht erbracht werden kann. [Pre94, GHJV94]

- Auswahl von Mustern:
  Muster können für ein sehr großes Spektrum an Entwicklungsaufgaben verwendet werden. Unter anderem eignen sich Muster auch zum Design von Echtzeitsystemen [Dou98b, Dou98a]. Bei der Auswahl, welche Muster für die Architektur eines Systems verwendet werden sollen, muß der Kontext der Aufgabe [GHJV94] beachtet werden. Darüberhinaus müssen die ausgewählten Muster an die konkrete Umgebung angepaßt werden, da sie nur abstrakte Designvorlagen (Templates) darstellen.

Muster werden in Katalogen gesammelt, einige Beispiele sind: [GHJV94, Pre94, CNM95, BMR+96, Cop96, Sch96a].

2.1.2.1 Arten objektorientierter Muster

Zur Unterstützung der Software-Entwicklung gibt es unterschiedliche Typen von Mustern. Insgesamt können vier große Gruppen von Mustern für die unterschiedlichen Phasen des Software-Entwicklungsprozesses (siehe Abschnitt 2.3.3.1) unterschieden werden (siehe Abbildung 2.2).

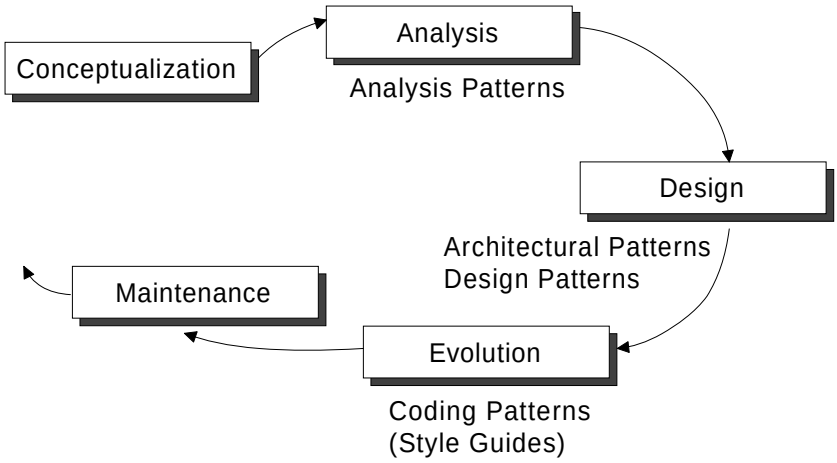


Abbildung 2.2: Verwendung der Muster im objektorientierten Entwicklungsprozeß nach Booch [Boo94]

1. **Analysemuster**:
   Diese Muster werden in der Analysephase des Software-Entwicklungsprozesses verwendet. Sie stellen vorgegebene Modelle dar, mit denen wiederkehrende Anforderungen an zu entwickelnde Systeme dargestellt werden können. In vielen Aufgabenbereichen (Application Domains) sind die konkreten Aufgabenstellungen meist sehr identisch. [Fow97]

   P. Coad, D. North und M. Mayfield definieren Analysemuster wie folgt [CNM95]:

   > "A pattern is a template of objects with stereotypical responsibilities and interactions; the template may be applied again and again, by analogy. Pattern instances are building blocks, very helpful in building effective object models."

   Analysemuster helfen bei der Entwicklung von Analysemodellen (Object Models). Sie werden jedoch nicht zur Beschreibung einer Systemarchitektur verwendet.

   Die in der Analyse mit Hilfe der Analysemuster entworfenen Modelle bilden die Basis für die Auswahl geeigneter Architekturmuster zur Entwicklung einer Systemarchitektur.

2. **Architekturmuster**:
   Der Entwurf von Systemen wird von Architekturmustern und Entwurfsmustern unterstützt. Architekturmuster werden zur Beschreibung der fundamentalen, grobgranularen Strukturen eines Systems verwendet. Dies geschieht meist zu Beginn des Designprozesses. Jede spätere Detaillierung des Entwurfs wird von diesen Strukturen bestimmt. [Dou98b]

   Architekturmuster befassen sich daher meist mit der Anordnung der Subsysteme auf abstrakter Ebene, während die feingranularen Entwurfsmuster sich eher mit dem Design der Subsysteme und den Details der Beziehungen der Subsysteme untereinander befassen.

   Von F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad und M. Stal stammt folgende Definition für Architekturmuster [BMR+96]:

   > "Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them."

3. **Entwurfsmuster**:
   Entwurfsmuster werden für den Entwurf der Details eines Systems verwendet. Sie verfeinern und erweitern die grundlegende Architektur. Beispielsweise können der interne Aufbau von Subsystemen oder die Beziehungen und Kommunikationsmechanismen zwischen Subsystemen beschrieben werden.

   E. Gamma, R. Helm, R. Johnson und J. Vlissides geben folgende Definition für Entwurfsmuster und deren Vorgehen zur Lösungen von Problemen [GHJV94]:

   > "A design pattern systematically names, motivates, and explains a general design that adresses a recurring design problem in object-oriented systems. [...]

> The solution is a general arrangement of objects and classes that solves the problem. The solution is customized and implemented to solve the problem in a particular context."

Durch die Beschränkung auf den detaillierten Bereich des Systemdesigns unterscheiden sich Entwurfsmuster von den Architekturmustern, die Vorgaben zur umfassenden Architektur eines Systems machen. Aufgrund dieser Einschränkung können Entwurfsmuster jedoch wesentlich allgemeiner verwendet werden als umfassendere Typen von Mustern. Während Architekturmuster sich nicht auf ein spezifisches Anwendungsdetail beziehen, sind Entwurfsmuster von einem bestimmten Anwendungsgebiet unabhängig. Eine grobe Architektur eines Systems kann durch mehrere Entwurfsmuster verfeinert werden.

Bei der Auswahl von Entwurfsmustern muß besonders auf die Umgebung der Problemstellung geachtet werden. Nur wenn der Kontext des Musters (siehe Abschnitt 2.1.3) mit der Aufgabenstellung übereinstimmt, ist das Entwurfsmuster optimal für die jeweilige Aufgabe geeignet.

4. **Muster und Richtlinien für die Software-Entwicklung (Idiome)**:
   Muster zur Codierung und Richtlinien für die objektorientierte Software-Entwicklung (beide Bezeichnungen werden hier synonym verwendet) beziehen sich auf eine bestimmte Programmiersprache. Sie werden im Software-Entwicklungsprozeß während der Implementierung verwendet. Dabei unterstützen sie die Umwandlung der Systemarchitektur in Code.

   Im einzelnen werden Muster zur Codierung für folgende Aufgaben verwendet [Pre94]:

   (a) Programmierrichtlinien demonstrieren, wie die grundlegenden Konzepte einer Programmiersprache angewendet werden können sowie wie diese Konzepte miteinander kombiniert werden können.

   (b) Richtlinien bilden die Basis für standardisierten Quellcode und Namensgebung.

   (c) Muster zur Software-Entwicklung helfen, typischen Fehler zu vermeiden und mit den Entwurfsfehlern, bzw. Unzulänglichkeiten, von Programmiersprachen umzugehen.

   Beispiele für Muster und Richtlinien zur Software-Entwicklung sind "Advanced C++ Programming Styles and Idioms" von J.O. Coplien [Cop92] und "Programming in C++, Rules and Recommendations" von M. Henricson und E. Nyquist [HN92].

Neben einer Einteilung der Muster in diese vier Gruppen können Muster auch entsprechend ihrer Verwendungsmöglichkeiten klassifiziert werden, z.B. Muster für verteilte Systeme, interaktive Systeme oder Muster, die die Kommunikation zwischen Komponenten modellieren [BMR+96].

2.1.3 Dokumentation objektorientierter Muster

Der Ansatz, daß Erfahrungen in der Software-Entwicklung als Muster gespeichert werden, ist an sich nicht neu [Knu92]. Untersuchungen haben ergeben, daß Software-Entwickler über Programme nicht auf der Ebene der einzelnen Elemente von Programmiersprachen denken,

sondern auf einer höheren Abstraktionsstufe [SE84, AS85, LC92]. Bisher sind die Erfahrungen meist individuell festgehalten worden. Diese persönlichen Muster wurden nur in geringem Maße ausgetauscht. Durch die systematische Sammlung und Dokumentation von Muster werden nun Problemlösungen für wiederkehrende Aufgabenstellungen der Öffentlichkeit zugänglich. Damit Muster allgemein verständlich in Handbüchern gesammelt werden können, müssen sie einheitlich gemäß eines standardisierten Formats dokumentiert werden.
Die wichtigsten Informationen zur Beschreibung von Mustern sind (Kontext der Muster) [GHJV94]:

- **Name des Musters**
  Durch den Namen kann ein Muster identifiziert werden. Ist ein Muster unter verschiedenen Namen bekannt, so müssen alle Bezeichnungen des Musters angegeben werden.

- **Problemstellung**
  Wichtig für die Auswahl eines Musters ist das Wissen darüber, welche Aufgaben durch die Muster gelöst werden. Daher gehören die Problemstellung, die das Muster behandelt, und deren Kontext zur Dokumentation.

- **Problemlösung durch das Muster**
  Die ausführliche Erklärung der Problemlösung umfaßt eine Übersicht sowohl über die statischen Zusammenhänge der Klassen des Musters als auch über deren dynamisches Verhalten.

- **Konsequenzen des Musters**
  Zum einen werden die Vorteile, die sich aus der Verwendung des Musters ergeben, aufgeführt. Zum anderen müssen auch die Auswirkungen auf das System (Architektur, Details des Designs und Implementierung) für das das Muster verwendet werden soll, beschrieben werden.

Muster können sowohl bei der Entwicklung von Software als auch zu deren Dokumentation verwendet werden. Muster vereinfachen die Beschreibung von Software-Systemen dadurch, daß sie helfen, komplexe Strukturen zu gliedern [OQC97].
Wird Software von vornherein mit Hilfe von objektorientierten Mustern entwickelt, so ergibt sich ein Synergieeffekt. Zuerst erleichtern Muster den Entwurf des Systems. Anschließend werden diese Muster dann in die Dokumentation übernommen.

2.1.4 Entwicklung objektorientierter Muster – Pattern Mining

Objektorientierte Muster leiten sich von realen Systemen ab. Die Erfahrungen, die bei der Entwicklung der Systeme gemacht werden, fließen in das Muster ein [Hel96, Ale99].
Das Vorgehen zur Entwicklung ist folgendes [BMR+96]:

1. Bestimmen eines wiederkehrenden Designs
   Ein bestimmtes Design oder Element eines Designs muß mehrmals erfolgreich bei der Entwicklung einer realen Anwendung verwendet worden sein. Dabei muß der Entwurf möglichst effizient sein.

2. Extrahieren des Lösungsweges
   Nur die abstrakte Lösung dient als Muster. Details werden nicht berücksichtigt.

3. Suche nach Beispielen
   Zusätzliche Anwendungsbeispiele sowie Code-Fragmente machen ein objektorientiertes Muster besser verständlich.

Bei der Entwicklung von Mustern muß folgendes beachtet werden [CNM95]:

- Muster sollen bekannte und erprobte Lösungen wiedergeben.
  Neue und nicht ausreichend getestete Lösungsansätze sollen nicht durch Muster beschrieben werden. Damit ist das Entwickeln von Mustern nicht eine Frage der Intuition, sondern die Suche nach immer wiederkehrenden erfolgreich angewendeten Lösungen.

- Muster müssen einheitlich und eindeutig beschrieben werden.
  Damit Muster allgemein verständlich in Handbüchern gesammelt werden können, müssen sie einheitlich dokumentiert werden (siehe Abschnitt 2.1.3).

2.2 Objektorientierte Frameworks

Im Gegensatz zu objektorientierten Mustern bieten Frameworks einen höheren Grad an Wiederverwendbarkeit, denn durch Frameworks wird die Architektur ebenso wie der Source-Code wiederverwendet.
Frameworks können sowohl prozedural als auch objektorientiert entwickelt werden. Allerdings unterstützt die objektorientierte Vorgehensweise zur Software-Entwicklung durch die Konzepte Modularität und Wiederverwendbarkeit Frameworks in besonderer Weise [FS97, FV98]. Im weiteren sollen in dieser Arbeit nur die objektorientiert entwickelten Frameworks betrachtet werden.
Nach einem kurzen Überblick über die bisherige Verwendung von Frameworks folgt eine genauere Definition sowie eine Beschreibung des Aufbaus von Frameworks. Den Abschluß des Kapitels bildet eine Einführung in ein Vorgehen zur Entwicklung von Frameworks und die Abgrenzung gegenüber objektorientierten Mustern und Bibliotheken.

2.2.1 Entwicklungsgeschichte

Frameworks stellen ebenso wie objektorientierte Software-Entwicklungsmethoden oder Muster ein Vorgehen zur Überwindung der Software-Krise dar. Frameworks sind vorgefertigte Software-Systeme, die entsprechend der konkreten Anwendung angepaßt werden können. Bereits auf der NATO-Konferenz 1968 wurde von D. McIlroy folgendes gefordert [McI76]:

> "Software engineers should take a look at the hardware industry and establish a software component subindustry. The produced software components should be tailored to specific needs but be reusable in many software systems."

Die ersten objektorientierten Frameworks sind für die Programmierung in Smalltalk-80 entwickelt worden [Sha89, Kay93]. Das bekannteste Smalltalk Framework dient zur Entwicklung graphischer Oberflächen (Graphical User Interface, GUI). Dazu verwendet es das Model-View-Controller Muster (siehe Abschnitt 2.1.1).
In der unmittelbaren Folgezeit sind weitere Frameworks vorallem zum Bau von graphisch interaktiven Benutzeroberflächen vorgestellt worden. Beispiele für solche Systeme sind MacApp (Framework zur Entwicklung von GUIs auf dem Apple Macintosh Rechner) [Sch86] oder ET++ (an der ETH Zürich entwickeltes Framework zur Entwicklung GUIs) [WGM89, Gam92, GHJV94]. Heute werden objektorientierte Frameworks für die unterschiedlichsten Aufgaben verwendet, wie z.B. Frameworks als einheitliche Programmierschnittstelle für unterschiedliche Betriebssysteme (ADAPTIVE Communication Environment, ACE [Sch93, Sch96b]), Kommunikations-Frameworks wie RMI (Remote Method Invocation) von JavaSoft oder Implementierungen der CORBA (Common Object Request Broker Architecture) Spezifikation als objektorientierte komponentenbasierte Frameworks.

2.2.2 Definition

Frameworks bestehen aus vorgefertigten, bzw. halbfertigen Software-Komponenten. Durch Anpassung können aus ihnen spezifische Applikationen, bzw. Teilkomponenten von Applikationen, entwickelt werden. Damit legt ein Framework auch implizit die Architektur, d.h. die Komposition der einzelnen Komponenten (bzw. Objekten) und die Interaktion zwischen den Komponenten, des zu entwickelnden Systems fest.
M. E. Fayad und D. C. Schmidt [FS97] definieren objektorientierte Frameworks wie folgt:

> "A framework is a semi-complete application that can be specialized to produce custom applications. In contrast to earlier OO reuse techniques based on class libraries, frameworks are targeted for particular business units (such as data processing or cellular communications) and application domains (such as user interfaces or real-time avionics)."

Kennzeichnend für die Software-Entwicklung mit Frameworks ist [MN96, FS97]:

- geringere Entwicklungs- und Wartungskosten von Applikationen
  Der Aufwand, eine Applikation durch die Anpassung eines Framework zu entwickeln, ist wesentlich geringer, als eine Anwendung neu zu entwickeln.

- hohe Software-Qualität
  Durch die Verwendung von bereits erfolgreich getesteten Software-Komponenten wird eine fehleranfällige Neuentwicklung vermieden.

- portable Systeme
  Da Frameworks zur Lösung verschiedener Varianten einer Problemstellung verwendet werden, müssen sie portabel sein. Systeme, die mittels Frameworks entwickelt werden, sind daher implizit in einem gewissen Maße portabel.

2.2.2.1 Arten von Frameworks

Entsprechend ihres Anwendungsgebiets (Anwendungsdomäne, Application Domain) können Frameworks in drei unterschiedliche Kategorien eingeteilt werden [FS97]:

1. *System Infrastructure Frameworks*
   Diese Frameworks werden zur Entwicklung der Infrastruktur von Systemen (System Infrastructure) verwendet. Beispiele für diese Art von Frameworks sind Betriebssystemzusätze [BMR+96] oder Kommunikations-Frameworks [Sch97b].

2. *Middleware Integration Frameworks*
   Die Integration verteilter Komponenten wird durch *Middleware Integration Frameworks* unterstützt. D.h. sie dienen dazu, modulare und wiederverwendbare Software zu entwickeln, die in einer verteilten Umgebung arbeiten soll. Zu diesen Frameworks gehören beispielsweise Object Request Broker (ORB), die nach dem CORBA-Standard der OMG [COR96, OHE96] entwickelt worden sind, oder transaktionsorientierte objektorientierte Datenbanken.

3. *Enterprise Application Frameworks*
   Diese Art von Frameworks unterstützt die Entwicklung von Applikations-Software, die direkt von einem Endanwender genutzt wird. Damit unterscheiden sie sich von den anderen Typen von Frameworks, die lediglich die Basis für die Applikationen darstellen. Die *Enterprise Application Frameworks* werden unter anderem bei der Entwicklung von Software im kaufmännischen Bereich [Bir93], der Produktionsplanung und -steuerung [Sch95, DH97, OYL97], für die Telekommunikation [Sch97b] oder der Steuerung von Automatisierungsgeräten im industriellen Umfeld [SP00] verwendet.

Die grundlegende Vorgehensweise zur Entwicklung von Frameworks (siehe Abschnitt 2.2.4) ist trotz der obigen Unterscheidung unabhängig von der jeweiligen Anwendung.

2.2.3 Aufbau

Der Aufbau von Frameworks ist orthogonal zu den in Abschnitt 2.2.2.1 beschriebenen Arten von Frameworks. Im folgenden werden die Bestandteile von Frameworks beschrieben. Zunächst wird das grundlegende Konzept, die Einteilung von Frameworks in *Hot Spots* und *Frozen Spots*, vorgestellt. Anschließend werden die *Hot Spots* genauer betrachtet. Den Abschluß bildet eine Unterscheidung zwischen *Black Box* und *White Box* Frameworks.

2.2.3.1 Hot Spots und Frozen Spots

Frameworks bestehen aus *Hot Spots* und *Frozen Spots* [Pre97, FS97, Sch97a, Sch98]. *Hot Spots* sind die Komponenten eines Framework, die flexibel verändert werden können. Die Klassen, die einen *Hot Spot* bilden, werden auch als *Hot Spot Subsystem* bezeichnet [Sch98]. *Frozen Spots* bilden den unveränderlichen Teil eines Framework. In einem Framework sollte der Anteil der festen Software-Komponenten deutlich größer sein als der der flexiblen Komponenten, denn die festen Elemente eines Framework sind diejenigen, die wiederverwendet werden.

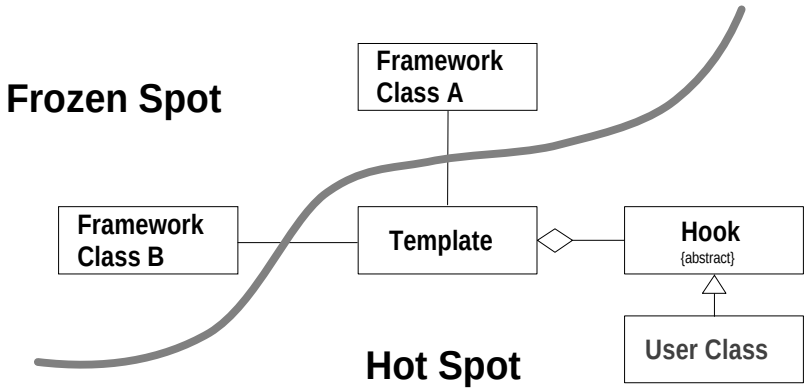


Abbildung 2.3: Framework mit Hot Spot und Frozen Spot

In Abbildung 2.3 wird die applikationsspezifische Klasse *User Class* des *Hot Spot Subsystems* in das Framework eingebunden [Pre97]:

- *Frozen Spot*
  In der Abbildung besteht der unveränderliche Anteil des Framework aus den Klassen A und B.

- *Hot Spot*
  Das *Hot Spot Subsystem besteht aus folgenden Klassen:*

  - *Template* (Schablone)
    Die Schablonenklasse enthält eine Methode (Schablonenmethode), die die *Hot Spot*-Klassen aufrufen. Die Schablonenklasse bestimmt damit den Platz des *Hot Spot* innerhalb des Framework. Diese Klasse bildet den Übergang zu den *Frozen Spots*.
  - *Hook* (Einschub)
    Die Einschubklasse (abstrakte Oberklasse) stellt die Schnittstelle der vom Anwender des Framework geschriebenen Klassen dar. In der Einschubklasse wird die

Schnittstelle, bzw.werden die Methoden dieser applikationsspezifischen Klassen festgelegt.

– Applikationsspezifische Klasse
Die Klasse *User Class* wird vom Anwender entwickelt. Die flexible Adaption des Framework geschieht durch Überschreiben der Methoden der *Hook*-Klasse durch die erbende applikationsspezifische Klasse.

Die *Template*- und die *Hook*-Klasse können zwei getrennte Klassen sein, die durch eine gerichtete Beziehung (z.B. Aggregation) miteinander verbunden sind. In Abbildung 2.3 wird diese Anpassung durch Komposition dargestellt. Alternativ können *Template* und *Hook* die gleiche Klasse sein, wie dies bei der Anpassung des Framework durch Vererbung der Fall ist. Die beiden Arten der Anpassung werden im folgenden Abschnitt beschrieben.
Die Architektur der *Frozen Spots* orientiert sich ausschließlich am jeweiligen Anwendungsgebiet des Framework nicht jedoch an einer konkreten Applikation, die vom Framework abgeleitet werden kann. Zur Entwicklung der *Frozen Spots* ist es sinnvoll, objektorientierte Software-Entwicklungsmethoden anzuwenden. Zum Entwurf können Muster verwendet werden.

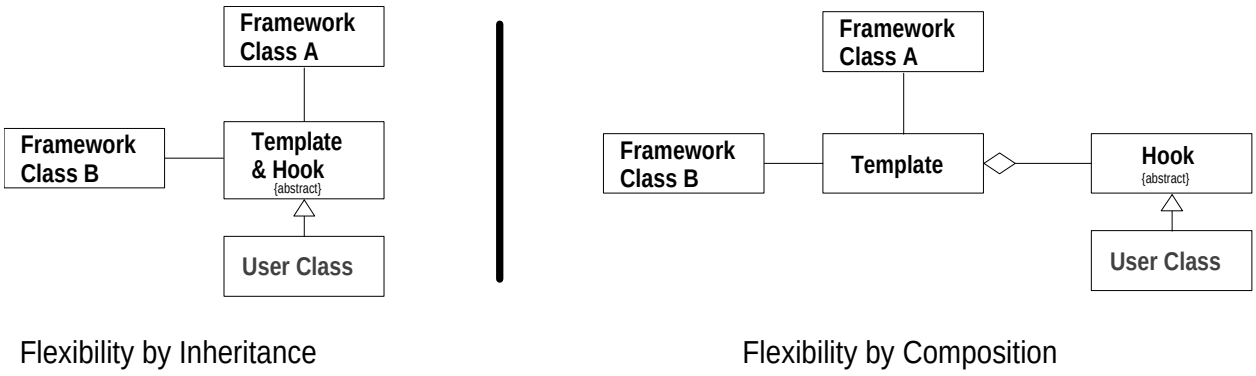


Abbildung 2.4: Anpassung durch Vererbung und durch Komposition

2.2.3.2 Arten von Hot Spots

Ebenso wie *Frozen Spots* können *Hot Spots* nach bestimmten Mustern (z.B. Entwurfsmustern [Sch98]) entwickelt werden. Durch die Beschränkung auf wenige Designvarianten (z.B. auf bestimmte Muster eines Musterkatalogs) können Anwender die *Hot Spots* leichter verstehen und entsprechend ihren Wünschen anpassen. Die grundsätzlichen Varianten von *Hot Spots* sind [Pre97, FS97, Sch97a] (siehe Abbildung 2.4):

1. Anpassung durch Vererbung
Bei dieser Art der Adaption sind *Template* und *Hook* zusammengefaßt.

   Diese Variante der *Hot Spots* kann im Gegensatz zur Anpassung durch Komposition sehr leicht entwickelt werden. Die Modellierung und Implementierung einer Aggregation zwischen *Template* und *Hook* kann entfallen.

2. Anpassung durch Komposition
Diese Adaption besteht aus jeweils getrennten *Template*- und *Hook*-Klassen. Die beiden Klassen sind durch Aggregationsbeziehungen verbunden.

Der Anwender-Code wird durch die Aggregation vom Rest des Framework getrennt. Der Anwender hat dadurch weniger Möglichkeiten, auf das Framework einzuwirken als bei der Anpassung durch Vererbung. Damit wird das Framework vor Fehlern des Anwenders besser geschützt.

In beiden Fällen wird eine vorgegebene Schnittstelle (Einschubklasse) durch eine vom Anwender geschriebene Klasse überschrieben. Die Flexibilität des Framework wird durch abstrakte Vererbung erreicht [Sch97a].

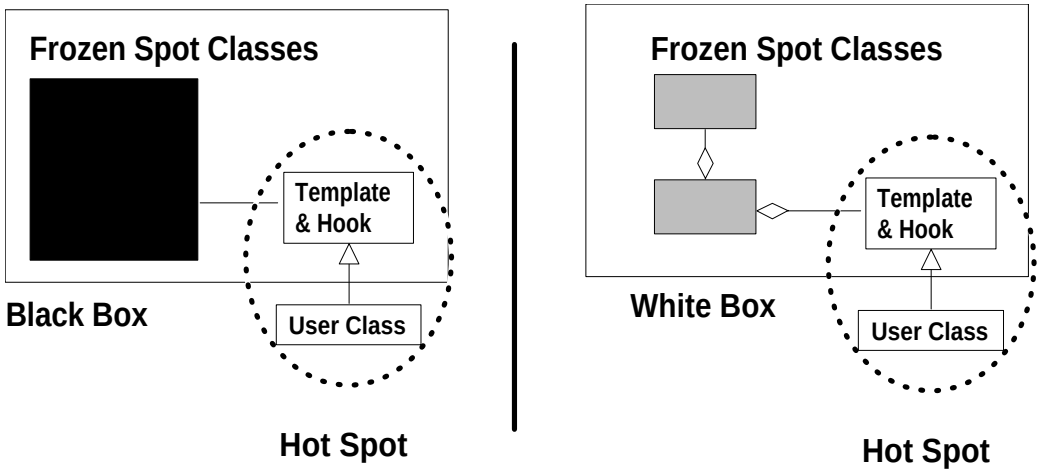


Abbildung 2.5: *Black Box* und *White Box* Frameworks

2.2.3.3 Black Box und White Box Frameworks

Frameworks können grundsätzlich auf zwei unterschiedliche Arten aufgebaut werden [Pre97, RJ98]:

1. *Black Box* Frameworks
   Bei dieser Art kennt der Anwender die interne Struktur des Framework nicht. Er kennt nur die *Hot Spots* des Systems, die flexibel angepaßt werden können und hat nur eine genau definierte Menge von Möglichkeiten zur Adaption. Das Framework ist eine *Black Box*.

   Der Vorteil dieser Vorgehensweise ist, daß der Benutzer eines Framework dessen interne Details nicht kennen muß. Damit wird die Fehlerwahrscheinlichkeit automatisch verringert. *Black Box* Frameworks setzen das Geheimnisprinzip nach D.L. Parnas [Par72] um.

   Die Entwicklung von *Black Box* Frameworks ist jedoch sehr aufwendig. Alle möglichen Alternativen zur Adaption des Framework müssen bei der Framework-Entwicklung bekannt sein und entsprechend berücksichtigt werden.

2. *White Box* Frameworks
   Bei diesen Frameworks ist dem Anwender die Architektur des Framework genau bekannt. Dieses Wissen benötigt der Anwender, damit er das Framework an seine Aufgabe anpassen kann.

Im Vergleich zu einem *Black Box* System kann ein *White Box* Framework sehr schnell entwickelt werden. Die unterschiedlichen Möglichkeiten zur Konfiguration des Framework müssen während dessen Entwicklung nicht vollständig berücksichtigt werden.

Der Nachteil eines *White Box* Framework ist, daß der Anwender ein genaues Verständnis des Framework benötigt. Daher ist die Einarbeitung aufwendiger.

Die meisten Frameworks bestehen aus einer Mischung von *Black Box* und *White Box* Anteilen (*Grey Box*-Wiederverwendung [FV98]). Da Frameworks vorwiegend in einem iterativen Prozeß entwickelt werden, enthalten die ersten Versionen (Releases) eines Framework meist sehr starke *White Box* Anteile. Im weiteren Verlauf werden die Frameworks zu *Black Box* Systemen verbessert [Pre97, FS97, Sch98].
Kennzeichnend für *White Box* Frameworks ist die bevorzugte Verwendung der Anpassung durch Vererbung. Dagegen ist die Anpassung durch Komposition charakteristisch für *Black Box* Frameworks.

2.2.4 Entwicklung objektorientierter Frameworks

Grundlage eines Framework ist immer eine erfolgreiche Architektur. Von dieser wird dann das Framework abgeleitet. In [Sch97a, Sch98] wird ein Vorgehen zur Entwicklung von Frameworks beschrieben. Eine ähnliche Vorgehensweise findet sich auch in [Pre97].

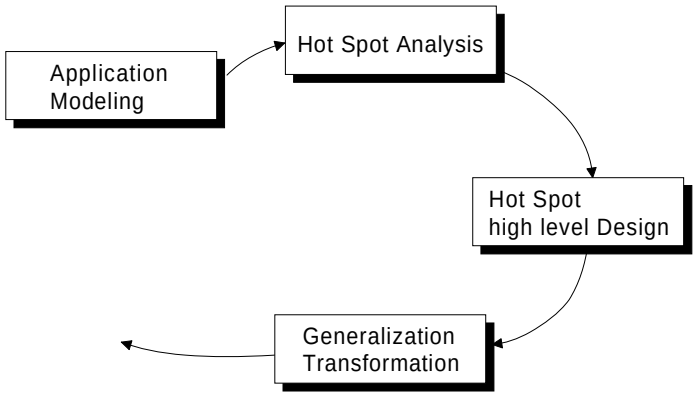


Abbildung 2.6: Entwicklungsprozeß von Frameworks

Die einzelnen Schritte bei der Entwicklung objektorientierter Frameworks sind (siehe Abbildung 2.6):

1. Entwicklung einer objektorientierten Applikation (*Application Modeling*)
   Basis eines Framework ist ein objektorientiertes System. Dieses System findet sich in den *Frozen Spots* des Framework wieder.

2. Analyse der gewünschten flexiblen Aspekte (*Hot Spot Analysis*)
   Zunächst muß die Menge der Applikationen, für die das Framework verwendet werden soll, bestimmt werden. Die *Hot Spots* werden durch eine Analyse der unterschiedlichen potentiellen Anwendungen gefunden. Im wesentlichen ergeben sich die flexiblen Elemente des Framework aus den Unterschieden zwischen den einzelnen Applikationsvarianten.

   Nachdem die einzelnen flexiblen Elemente (*Hot Spots*) des Framework identifiziert sind, werden sie genauer analysiert und spezifiziert.

3. High-level Design der *Hot Spots* (*Hot Spot high level Design*)
   In diesem Schritt wird die Architektur der *Hot Spots* (Anordnung der Klassen) aufgrund ihrer Spezifikation entwickelt. Ein *Hot Spot* kann aus vorgefertigten Klassen bestehen und eine oder mehrere applikationsspezifische Klassen enthalten. Die vom Anwender geschriebenen Klassen können zur Compile-Zeit oder dynamisch zur Laufzeit an das Framework gebunden werden.

4. Generalisierung durch *Hot Spots* (*Generalization Transformation*)
   Entsprechen des gewählten Designs wird das *Hot Spot Subsystem* codiert. Die objektorientierte Applikation aus 1. bildet dafür die Basis. Die Generalisierung geschieht dadurch, daß *Hot Spots* in das System eingefügt werden.

Ähnlich wie das Vorgehen zur Entwicklung objektorientierter Systeme (wie beispielsweise in [Boo94], [You94], [Dod96] oder [BR95] beschrieben) ist auch die Entwicklung von Frameworks ein iterativer und inkrementeller Prozeß.

2.2.5 Abgrenzung von Frameworks gegenüber Mustern und Klassenbibliotheken

Neben Frameworks gibt es zwei weitere Konzepte für die Wiederverwendung der Architektur und Software von Systemen: Muster und Klassenbibliotheken.

2.2.5.1 Frameworks versus Muster

Das Verhältnis zwischen Frameworks und Muster wird in [GHJV94] folgendermaßen beschrieben:

> "[...] frameworks are implemented in a programming language. [...]
> In this sense frameworks are more concrete than design patterns.
> [...] mature frameworks usually reuse several design patterns"

Frameworks unterscheiden sich in zwei Aspekten wesentlich von objektorientierten Mustern:

1. Frameworks beschreiben nicht nur das Design, sondern sie bieten auch vorgefertigte Software-Komponenten mit deren Hilfe ein System entwickelt wird.

2. Da Frameworks aus vorimplementierter Software bestehen, können sie nicht so flexibel an unterschiedliche Problemstellungen angepaßt werden wie Muster. Daher dienen Frameworks zur Lösung einer eng begrenzten Klasse von Aufgaben, während Muster die abstrakte Architektur von Systemen oder deren Teilaspekten beschreiben.

Trotz der Unterschiede können sich objektorientierte Muster und Frameworks gegenseitig ergänzen:

- Muster werden beim Design von Frameworks eingesetzt [LN95, Pre97, FS97, RJ98, Sch98]. Beispielsweise beschreiben viele Entwurfsmuster, wie ein System flexibel entwickelt werden kann [1]. Damit helfen Muster beim Entwurf von *Hot Spot Subsystemen* in Frameworks.

[1] 19 von 23 Entwurfsmustern in [GHJV94] beschreiben die flexible Kooperation von Klassen und Subsystemen.

- Im Gegenzug sind viele bekannte objektorientierte Muster von Frameworks abgeleitet. Das bekannteste Beispiel hierfür ist der Entwurfsmusterkatalog "Design Pattern: Abstractions and Reuse of Object-Oriented Software" [GHJV94], der auf dem Framework ET++ basiert [Gam92].

- Muster können zur Dokumentation von Frameworks verwendet werden [Joh92, Gam92, GHJV94, Pre94, LN95, PPSS95, BMR+96, MCK97]. Muster helfen dadurch, den Aufbau von Frameworks zu beschreiben.

2.2.5.2 Frameworks versus Klassenbibliotheken

Im Gegensatz zu objektorientierten Mustern enthalten sowohl Klassenbibliotheken als auch Frameworks Software. Dieser Code kann vom Anwendungsentwickler (wieder-)verwendet werden. Dadurch können Systeme schneller entwickelt werden.

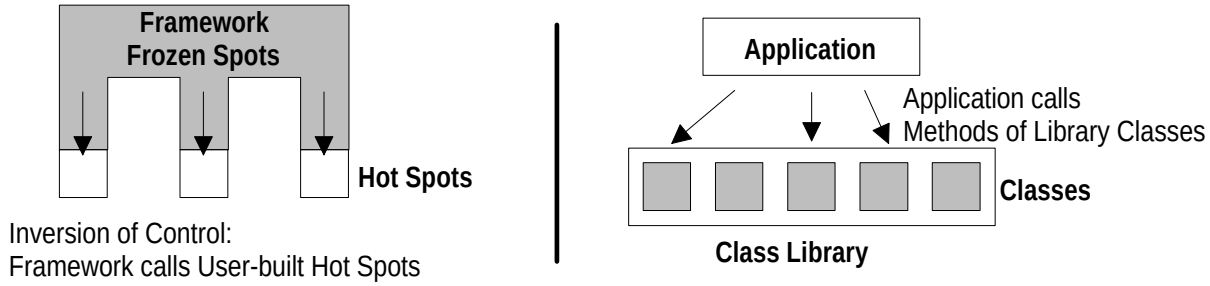


Abbildung 2.7: Frameworks versus Klassenbibliotheken

Die wesentlichen Unterschiede zwischen Frameworks und Klassenbibliotheken sind [Pre94, Pre97]:

- Frameworks haben eine feste Architektur. Diese Struktur kann der Anwender nicht verändern. Bibliotheken liefern keine Anwendungsstrukturen. Der Anwendungsentwickler definiert die gesamte Systemarchitektur selbst. In Bibliotheken muß es daher keine Mechanismen (*Hot Spots*) geben, durch die vorgegebene Strukturen flexibel gehalten werden können.

- Bei der Verwendung von Klassenbibliotheken wird der Kontrollfluß durch die Applikation bestimmt. In Frameworks ist dies umgekehrt. Hier bestimmt das Framework den Kontrollfluß. Die vom Anwender eingefügten Klassen (*Hot Spots*) werden vom Framework aus aktiviert (Call-Back Mechanismus).

Bibliotheken können ebenso wie Frameworks auch nicht objektorientiert sein. Prozedurale Bibliotheken bieten z.B. Prozeduren, die von einem Applikationsprogramm aus aufgerufen werden können. Allerdings können Änderungen in prozeduralen Bibliotheken aufgrund der fehlenden Kapselung sehr weitreichende Folgen haben.

2.2.6 Software-Komponenten

Nach [Szy97] können Komponenten auch als feingranulare Frameworks betrachtet werden. Neben der Wiederverwendbarkeit auf Code-Ebene ist auch die Parametrisierbarkeit eine gemeinsame Eigenschaft. Darüberhinaus können sehr flexible Frameworks aus Komponenten

aufgebaut werden [Pre97] bzw. umgekehrt basieren komponentenbasierte Systeme meist auf Frameworks.
Verglichen mit Frameworks, die nicht auf Komponenten basieren, setzen komponentenbasierte Systeme die Idee von D. McIlroy (siehe Abschnitt 2.2.1, [McI76]) noch besser um. Nach D. McIlroy soll Software analog zum Bau von Maschinen aus einzelnen Komponenten zusammengesetzt werden, wobei diese Komponenten von Zulieferern vorgefertigt zur Verfügung stehen.
Software-Komponenten sind ein vergleichsweise neuer Ansatz (d.h. seit Beginn der 90er Jahre Thema in Forschungsarbeiten), die Architektur von Systemen zu gliedern und in einzelne Komponenten zu kapseln. Komponenten sind eine Möglichkeit, flexible objektorientierte Standard-Software zu realisieren [NGT92].
Bereits heute wird die komponentenbasierte Software-Entwicklung von verschiedenen Programmierumgebungen unterstützt. Die erfolgreichsten Systeme sind CORBA (bzw. die Implementierungen des CORBA-Standards), COM mit ActiveX und Java in Verbindung mit JavaBeans.
Der Vorteil des Einsatzes von Komponenten ist der höhere Grad an Wiederverwendung und damit eine reduzierte Time-to-Market [Szy97].

2.2.6.1 Definition von Komponenten

Ursprünglich sind Software-Komponenten mit Hardware-Komponenten (insbesondere integrierten Schaltungen, ICs) verglichen worden. Dabei werden Begriffe wie *Software IC* für die eigentliche Komponente und *Software Bus* oder *Software Backplane* für die Kommunikationsbeziehungen zwischen den Komponenten verwendet. Allerdings wird bei diesen Bezeichnungen nicht berücksichtigt, daß Komponenten Metaprodukte sind, d.h. analog zu Frameworks angepaßt werden können. [Szy97]
Eine einheitliche Definition von Software-Komponenten existiert bisher noch nicht. Auf dem ECOOP Workshop über komponentenorientierte Programmierung aus dem Jahr 1996 wurde folgende Definition formuliert:

> "A software component is a unit of composition with contractually specific interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

C. Szyperski liefert in [Szy97] eine Erweiterung dieser Definition:

> "A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. [...]
> A component can be deployed independently and is subject to composition by third parties. For the purpose of independent deployment, a component needs to be a binary unit. To distinguish between the deployable unit and the instances it supports, a component is defined to have no mutable persistent state. Technically, a component is a set of atomic components, each of which is a module plus resources."

Diese Definition erfaßt Komponenten nicht vollständig. Nicht beachtet ist der Gültigkeitsbereich von Komponenten, d.h. der Kontext, in dem eine bestimmte Software als eine Komponente betrachtet werden kann.
Dieser Gültigkeitsbereich hängt stark von der Abstraktionsebene ab, auf der die Komponente angesiedelt ist. Im Maschinenbau kann beispielsweise ein Kolben auf der Ebene der Montage

eines Verbrennungsmotors eine sinnvolle und gültige Komponente darstellen. Auf einer übergeordneten Ebene, z.B. der Endmontage des Fahrzeugs, stellt der Kolben allerdings keine sinnvolle Komponente dar, da er für diesen Kontext ein zu feingranularer Bestandteil (der Komponente Motor) ist.
Zum Kontext einer Komponente gehört damit auch deren Verwendung innerhalb des Software-Entwicklungsprozesses.

2.2.6.2 Komposition von Komponenten

Komponenten-Systeme können aus stark kohesiven, aber lose gekoppelten Komponenten, die durch "Plugs" verbunden werden, bestehen [ND95]. Die Menge der Schnittstellen und Regeln (Kontrakte), die die Interaktionen der Komponenten in einem Framework bzw. System definieren, wird als Komponenten-Framework (Component Framework) bezeichnet [Szy97]. Typischerweise beschreibt ein Komponenten-Framework die wichtigsten Regeln der Interaktionen und kapselt damit die benötigten konkreten Interaktionsmechanismen. Konkrete Implementierungen (in den Komponenten) realisieren die Interaktionsregeln.
Konkret können Komponenten unterschiedlich verbunden werden. Ein sehr weit verbreiteter Ansatz ist die flexible Kombination mittels CORBA. Gegenpol hierzu ist die statische Verbindung von Komponenten, wie sie von W. Pree in [Pre97] vorgestellt wird (siehe Abschnitt 2.2.3.2).

2.2.6.3 Abgrenzung gegenüber Objekten, Frameworks und Modulen

Aufgrund der Ähnlichkeiten der Konzepte von Komponenten und Objekten, Frameworks sowie Modulen ist eine Unterscheidung oft schwierig. Diese Schwierigkeit impliziert allerdings nicht, daß sich die Unterschiede allein auf unterschiedliche Begrifflichkeit beschränken [Szy97].

- **Komponenten und Objekte**
  Komponenten setzen sich meist aus mehreren Objekten (bzw. Klassen) zusammen. Zum Teil werden Komponenten und Objekte in der Literatur gleichgesetzt oder begrifflich zu Komponentenobjekten (Component Objects) kombiniert.

  Sowohl Objekte als auch Komponenten bieten ihre Dienste durch eine Menge von definierten Schnittstellen an. Die Interaktion und Kombination von Objekten und Komponenten kann durch Muster beschrieben [Kot98], bzw. durch Frameworks in vorgefertigte Software umgesetzt werden.

  Charakteristisch für Komponenten ist, daß sie von dritter Hand entwickelt, parametrisiert und anschließend in beliebige Applikationen eingebaut werden.

- **Komponenten und Frameworks**
  Konventionelle Frameworks sowie Komponenten unterstützen die Wiederverwendung von Code. Komponenten werden für die Wiederverwendung ebenso wie Frameworks parametrisiert. Sie können daher *Black Box*, *White Box* oder *Grey Box* Ausprägung haben (siehe Abschnitt 2.2.3.3).

  Konventionelle Frameworks sind allerdings vorgefertigte, meist monolithische Schablonen für vollständige Applikationen, die angepaßt werden können. Komponenten bearbeiten demgegenüber meist nur einen Teilbereich einer Applikation. Ein vollständiges System setzt sich dann aus mehreren Komponenten zusammen.

- **Komponenten und Module**
  Komponenten haben Ähnlichkeit mit Modulen in Modula-2 oder Ada. Sie können beispielsweise separat compiliert werden. Nach [Szy97] kann ein Modul als minimale Komponente betrachtet werden. Allerdings bieten Komponenten Schnittstellen an, durch die sie bzw. deren Ressourcen konfiguriert werden können [Szy97].

2.3 Objektorientierte Echtzeitsysteme

Dieses Kapitel stellt eine allgemeine Beschreibung objektorientierter Echtzeitsysteme dar. Dabei wird deren Einsatzzweck nicht näher betrachtet. Industrielle Steuerungssysteme als spezielle Anwendungen finden sich im folgenden Abschnitt 2.4.
Zunächst werden Echtzeitsysteme definiert und klassifiziert. Darauf folgt eine Übersicht über "allgemeine" objektorientierte Vorgehensweisen zur Entwicklung von Echtzeitsystemen und spezielle objektorientierte Echtzeitmethoden. Den Abschluß bildet eine Beschreibung von Echtzeitbetriebssystemen.

2.3.1 Definition

Die meisten nicht-Echtzeitsysteme sind interaktive Systeme, die auf den Dialog mit dem Benutzer optimiert sind. Das Zeiterhalten dieser interaktiven Systeme hängt somit vom Anwender ab und ist insgesamt nicht exakt vorhersehbar. Im Gegensatz dazu garantieren Echtzeitsysteme, daß bestimmte Tasks innerhalb bestimmter zeitlicher Grenzen abgearbeitet werden [Dou98b]. Dabei können Echtzeitsysteme auch nicht echtzeitfähige Komponenten (z.B. Menüs zur Benutzerinteraktion) besitzen.
J. Martin definiert Echtzeitsysteme folgendermaßen [Mar67]:

> "A real-time computer system may be defined as one which controls an environment by receiving data, processing them, and taking action or returning results sufficiently quickly to affect the functioning of the environment at that time."

Weitere charakteristische Eigenschaften von Echtzeitsystemen sind [Ell94, Gal95, Dou98b]:

- Echtzeitsysteme interagieren in erster Linie mit anderen Geräten, den Aktuatoren (z.B. Antriebe eines Roboterarms) und Sensoren (z.B. Inkrementalmeßsyteme zur Bestimmung der Armstellung eines Roboters).

  Echtzeitsysteme interagieren meistens mit einer Menge von Geräten, bzw. mehreren Schnittstellen von Geräten. Die Tasks zur Ansteuerung der Geräte laufen auf Echtzeitsystemen daher meist nebenläufig ab [Hüs95].

- Echtzeitsysteme müssen ein hohes Maß an Robustheit und Zuverlässigkeit aufweisen. Bei Fehlern müssen aussagekräftige Fehlermeldungen ausgegeben werden und das System in einen sicheren Zustand überführt werden.
  In einer Steuerung eines Industrieroboters dürfen beispielsweise keine Fehler wie der von Microsoft im Release Windows 3.1 eingeführte *General Protection Fault* vorkommen [Dou98b].

- Echtzeitsysteme arbeiten entweder periodisch oder aperiodisch. Periodische Systeme arbeiten die Echtzeit-Software zyklisch ab, wie z.B. speicherprogrammierbare Steuerungen, die zyklisch Eingangswerte lesen, die Daten verarbeiten und dann die Ergebnisse wieder ausgeben.

  Aperiodische Systeme reagieren auf sporadische Ereignisse (Interrupts). Rein Interrupt-getriggerte Systeme sind selten. Meist sind Notfallroutinen als Interrupt-Handler realisiert, wobei die Funktionalität für den Normalbetrieb zyklisch abgearbeitet wird. Beispielsweise können derartige Notfallroutinen durch Hardware-Interrupts ausgelöst werden.

- Echtzeitsysteme sind üblicherweise diskrete Steuerungen für komplexe Geräte (z.B. Robotersteuerungen) oder eingebettete Systeme (embedded Systems). Dies sind Steuerungssysteme (meist Einplatinen- oder Single-Chip-Rechner), die in das zu steuernde Gerät (z.B. Haushaltsgerät) oder die Maschine (z.B. Fahrzeug oder Werkzeugmaschine) eingebaut sind.

2.3.2 Klassifikation von Echtzeitsystemen

In diesem Abschnitt werden verschiedene Möglichkeiten zur Einteilung von Echtzeitsystemen vorgestellt. Zuerst werden harte und weiche Echtzeitsysteme unterschieden, darauf folgt eine Klassifikation anhand der Zeitanforderungen an die Systeme.

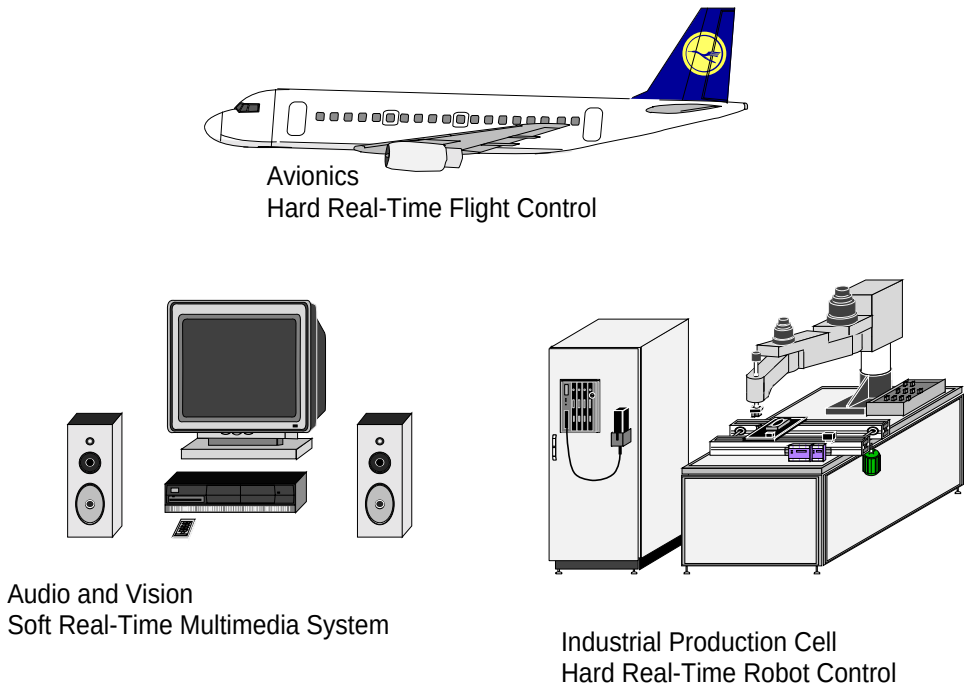


Abbildung 2.8: Beispiele für harte und weiche Echtzeitsysteme

2.3.2.1 Harte und weiche Echtzeitsysteme

Bei Echtzeitsystemen wird zwischen harten (*hard real-time*) und weichen (*soft real-time*) Systemen unterschieden (siehe Abbildung 2.8) [Gal95, Dou98b]:

- Harte Echtzeitsysteme:
  Eine Reaktion ist nur dann korrekt, wenn die Antwort innerhalb eines festen Zeitlimits erfolgt. Eine verspätete Antwort ist als Systemfehler zu bewerten, mit der Folge, daß das System stoppen, bzw. in einen definierten, sicheren Endzustand übergehen muß. Eine Zeitabweichung kann sonst eventuell zur Zerstörung eines Systems führen. Harte Echtzeitsysteme sind meist sehr maschinennah.
  Ein Beispiel für ein solches System ist die Regelungs-Software für Roboterantriebe. Innerhalb einer definierten Zeit muß eine Reaktion auf ein Eingangssignal erfolgen, ansonsten arbeitet die Regelung nicht mehr korrekt.

- Weiche Echtzeitsysteme:
  Um weiche Echtzeitanforderungen zu erfüllen, muß ein System die geforderten zeitlichen Bedingungen im Durchschnitt einhalten. Innerhalb einer gewissen (vorgegebenen) Toleranz kann eine Antwort verspätet erfolgen. Dies bedeutet noch keinen Fehlerzustand.
  Multimedia-Anwendungen sind beispielsweise weiche Echtzeitsysteme. Abweichungen von den Zeitvorgaben mindern zwar die Qualität, aber sie führen zu keinem inkorrekten Zustand oder zur Beschädigung des Systems.

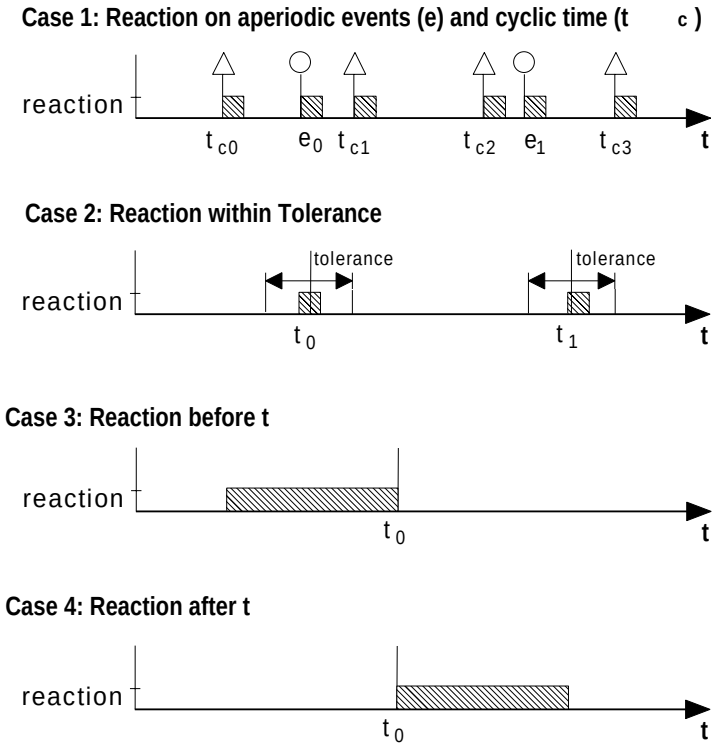


Abbildung 2.9: Klassifikation von Echtzeitbedingungen

2.3.2.2 Klassifizierung von Zeitbedingungen

Die Zeitanforderungen von harten wie weichen Echtzeitsystemen können in unterschiedliche Fälle eingeteilt werden [SSDB95] (siehe Abbildung 2.9):

Fall 1: Eine Funktion wird aperiodisch ausgeführt, d.h. die Funktion wird durch ein definiertes Ereignis getriggert und abgearbeitet, wie die Reaktionen auf die aperiodischen Ereignisse e_0 und e_1. Bei einer industriellen Steuerung können solche aperiodischen Ereignisse Sensormeldungen sein, auf die die Steuerung reagieren muß.
Im Gegensatz dazu können Funktion auch zyklisch ab einem bestimmten Zeitpunkt t_{c0}, t_{c1}, ...) bearbeitet werden. Beispiele hierfür sind das zyklische Versenden eines Synchronisationssignals über einen Feldbus.

Fall 2: Eine Funktion wird zu einem festen Zeitpunkt (t_0, t_1, ...) innerhalb eines Toleranzintervalls ausgeführt. Bei Feldbussystemen können beispielsweise bestimmte Nachrichten innerhalb eines bestimmten Zeitfensters versendet werden.

Fall 3: Eine Funktion wird in einem Zeitintervall bis spätestens zu einem festen Zeitpunkt (t_0) ausgeführt.
Eine Industriesteuerung muß zum Beispiel vor Erreichen eines Zeitpunkts den Fertigungsprozeß beenden.

Fall 4: Eine Funktion wird in einem Zeitintervall frühestens ab einem festen Zeitpunkt (t_0) ausgeführt.

Die Zeitbedingungen können orthogonal zu den oben genannten Fällen in zwei Kategorien eingeteilt werden:

1. Absolutzeit
   Die Absolutzeitbedingungen beziehen sich auf einen absoluten Zeitpunkt (z.B. reagiere zum Zeitpunkt t_1 auf eine bestimmte Weise).

2. Relativzeit
   Die Zeitbedingungen sind relativ zu den (noch bevorstehenden) Ereignissen (z.B. wenn ein Meßwert einen Grenzwert überschreitet, soll nach n Sekunden ein Abschaltsignal ausgegeben werden).

2.3.3 "Allgemeine" objektorientierte Software-Entwicklungsmethoden

Alle "allgemeinen" Methoden zur objektorientierten Software-Entwicklung können grundsätzlich zur Konzipierung von Echtzeitsystemen verwendet werden. Dabei können alle Vorteile der objektorientierten Vorgehensweise genutzt werden [Mey88]. Die wichtigsten dieser Methoden sind: Object-Oriented Analysis and Design (OOAD) [Boo91, Boo94], Object Modelling Technique OMT [RBP+91], Object-Oriented Analysis (OOA) und Object-Oriented Design (OOD) [CY91a, CY91b], Object-Oriented Software Engineering (OOSE) [JCJ92] und Object-Oriented System Design (OOSD) [You94] sowie die Vereinigungen von Methoden Grand Unified Method (GUM) [Dod96] und Unified Modelling Language (UML) [BR95].
Im Vergleich zu objektorientierten Echtzeitmethoden bieten diese "allgemeinen" Vorgehensweisen keine vollständige Notation zur Modellierung aller für Realzeitsysteme wichtigen Eigenschaften (siehe Abschnitt 2.3.4.3). Allerdings können die fehlenden Darstellungsvarianten bei Bedarf ergänzt werden. Viele objektorientierte Methoden zur Entwicklung von Echtzeitsystemen basieren auf den "allgemeinen" Vorgehensweisen, indem sie teilweise die Notation und den Prozeß zur Software-Entwicklung aufgreifen.
Bei einem Vergleich der Methoden ergibt sich eine Konvergenz im Vorgehen (Entwicklungsprozeß) und den verwendeten Modellen (Notation) [HSG96, Dod96]. Diese Übereinstimmung wird in dem Zusammenschluß der Methoden von G. Booch, J. Rumbaugh und I. Jacobson zur Unified Modelling Language unter dem Dach der Rational Software Corporation deutlich [BR95].

2.3.3.1 Objektorientierter Entwicklungsprozeß

Im Gegensatz zu prozeduralen Methoden, die ein streng sequentielles Vorgehensmodell wie z.B. das Wasserfallmodell [Roy70, Boe76] propagieren, wenden objektorientierte Vorgehensweisen einen iterativen und inkrementellen Entwicklungsprozeß an. Bei Bedarf können einzelne Phasen der Entwicklung wiederholt werden.

Zwischen den einzelnen Abschnitten, bzw. Phasen, der Entwicklung gibt es keine strikte Trennung [Dod96]. Der Entwurf von Teilkomponenten eines Systems kann in jeder Phase geändert werden.

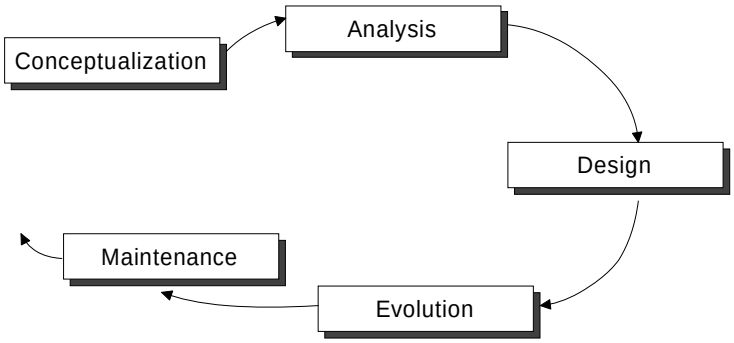


Abbildung 2.10: Objektorientierter Software-Entwicklungsprozeß nach Booch [Boo94]

Bei allen oben genannten Methoden ist der Entwicklungsprozeß in die Abschnitte Konzeptualisierung, Analyse, Design, Implementierung und Wartung (siehe Abbildung 2.10) unterteilt. Allerdings haben die Abschnitte teilweise andere Namen [Boo94, RBP+91, JCJ92, CY91a, CY91b, You94]:

1. Konzeptualisierung des Projekts (Conceptualization)
   In der Konzeptualisierung werden die Ziele des Projekts formuliert und die grundlegenden Anforderungen bestimmt. Die funktionalen und nicht funktionalen Anforderungen (functional and non-functional Requirements) werden festgehalten.

   Durch Prototypen wird in dieser Phase die Durchführbarkeit des Projekts ermittelt [BKKZ92, Boo94].

2. Objektorientierte Analyse (Analysis)
   In der Analyse wird festgestellt, was genau das System tun soll, aber nicht wie. Zunächst wird die in der Konzeptualisierung begonnene Erfassung der Anforderungen überarbeitet und vervollständigt. Dazu dienen Use Case Modelle und dynamische Szenarien.

   Die Anforderungen dienen als Grundlage für den Aufbau des Analyse Modells (Bestimmung der Klassen und Objekten [Weg87]). Dieses Modell besteht aus initialen Klassenmodellen (statische Architektur) und Interaktionsdiagrammen (dynamisches Verhalten), die den Problembereich, bzw die Aufgabenstellung, (Problem Domain) wiedergeben. Szenarien helfen bei der Verifikation der Diagramme.

3. Objektorientiertes Design
   Das Ziel der Designphase ist eine vollständige Beschreibung der Architektur des zu entwickelnden Systems. Die in der Analyse beschriebenen Aufgaben, müssen durch das Architekturmodell erfüllt werden.

   Die strategischen Designüberlegungen werden in Klassenmodellen, Objektdiagrammen und Interaktionsdiagrammen festgehalten. Die Überprüfung der Designmodelle geschieht wiederum mittels Szenarien. Wenn des Designmodell vollständig und verifiziert ist, kann das Design beendet werden. Ein Kriterium für die Qualität des Designs ist die Einfachheit der im Design entwickelten Architektur.

4. Objektorientierte Implementierung (Evolution)
   Im Implementierungsabschnitt werden die Modelle des Designs in Code umgesetzt. Besonders wichtig ist, die Entwicklung der Software genau zu dokumentieren.

   Das Ergebnis der Implementierung sind mehrere lauffähige System-Releases (inklusive dem fertigen Produkt). Die Implementierung wird abgeschlossen, wenn die Qualität und Funktionalität der Software den Wünschen des Kunden entspricht.

5. Wartung (Maintenance)
   Diese Phase umfaßt den Betrieb und die Wartung des implementierten Systems und alle weiteren Arbeiten nach Auslieferung und Inbetriebnahme der Software.

Diagramm	**Beschreibung**	**Verwendung im Entwicklungsprozeß**
initiale Aufgabenbeschreibung	formlose Beschreibung der Problemstellung durch Kunde und Entwickler	Konzeptualisierung
Use Case Modell	funktionale Anforderungen an das System	Konzeptualisierung, Analyse
nichtfunktionale Anforderungen	nichtfunktionale Anforderungen an das System	Konzeptualisierung, Analyse
Klassendiagramm	statischer Aufbau des Systems	Analyse, Design
Zustandsdiagramm und Petri Netze	dynamische Abläufe innerhalb von Klassen und zwischen den Klassen	Analyse, Design
Objektdiagramm (in UML Kollaborations- diagramm)	dynamische Abläufe zwischen Objekten zwischen Objekten	Analyse, Design Design
Interaktionsdiagramm (in UML: Sequenzdiagramm)	dynamische Abläufe zwischen Objekten in Szenarien	Analyse, Design
Moduldiagramm	physikalisches Modell zur Beschreibung der Implementierung des Sytems	Design, Implementierung

Tabelle 2.1: Diagramme und Beschreibungsformen der objektorientierten Entwicklungsmethoden

2.3.3.2 Notation

Die in Tabelle 2.1 aufgeführten Diagramme werden zur Beschreibung objektorientierter Modelle in den "allgemeinen" objektorientierten Software-Entwicklungsmethoden verwendet. Zwischen den einzelnen Methoden gibt es dabei geringe Unterschiede in der verwendeten Symbolik und der Bezeichnung der Modelle, die sich jedoch nicht auf die Aussage der Diagramme auswirken.
Alle diese Diagramme können zur Modellierung von objektorientierten Echtzeitsystemen verwendet werden [CG95].

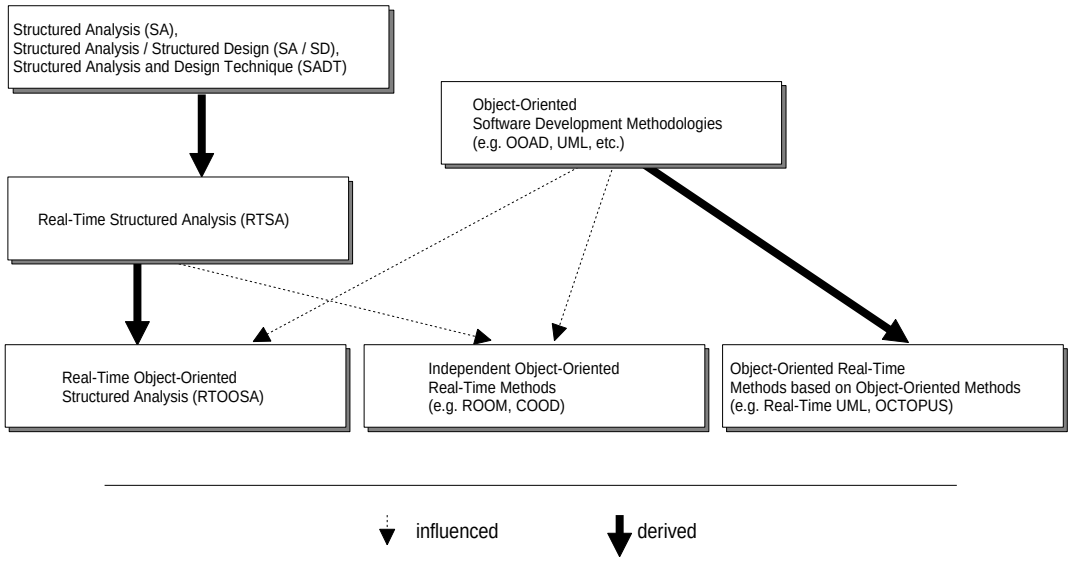


Abbildung 2.11: Ursprünge objektorientierter Methoden zur Entwicklung von Echtzeitsystemen

2.3.4 Spezielle Methoden zur objektorientierten Entwicklung von Echtzeitsystemen

Die unterschiedlichen objektorientierten Methoden zur Entwicklung von Echtzeitsystemen sind von verschiedenen Vorgehensweisen beeinflußt worden (siehe Abbildung 2.11):

- Eine Variante ist, bekannte objektorientierte Vorgehensmethoden (siehe Abschnitt 2.3.3) um Modelle zu erweitern, mit denen für Echtzeitsysteme spezifische Anforderungen dargestellt werden. Beispiele für diesen Ansatz sind Real-Time UML [Dou98b, Sel99] oder OCTOPUS [2] (etwickelt in der Fa. Nokia, Helsinki) [AKZ96]. OCTOPUS basiert auf Fusion [CAB+93] und OMT (Object Modelling Technique) [RBP+91], bzw. UML.

- Andere objektorientierte Methoden für Echtzeitsysteme sind von erfolgreich verwendeten strukturierten Vorgehensweisen zur Entwicklung von Realzeitsystemen [Alf77, RS77, WM85, Sch92, Edw93] abgeleitet worden. Die strukturierten Methoden zur Entwicklung von Echtzeitsystemen leiten sich wiederum von der *Structured Analysis (SA)*, *Structured Analysis / Structured Design (SA / SD)* und *Structured Analysis and Design Technique (SADT)* ab [DeM78, Pet87, Loy90]. Ein Beispiel hierfür ist RTOOSA (Real-Time Object-oriented Structured Analysis) von J.R. Ellis [War89, Ell94].
  RTOOSA leitet sich von RTSA (Real-Time Structured Analysis) [WM85] ab.

- Daneben gibt es spezielle Entwurfsmethoden für Echtzeitsysteme, wie z.B. ROOM (Real-Time Object-Oriented Modelling) [SGW94] oder COOD (Concurrent Object-Oriented Design) [Hüs95].

Zusätzlich zu diesen Methoden gibt es Verfahren, die auf ein bestimmtes Aufgabengebiet beschränkt sind. Ein Beispiel hierfür ist OSDL (Object-Oriented Specification and Description Language) [BDMP87]. OSDL ist eine objektorientierte Erweiterung der Entwurfsmethode SDL (Specification and Description Language) [CCITT88, Hoc89]. Sie unterstützt speziell die

[2] Der Name OCTOPUS hat keine besondere Bedeutung, bzw. ist kein Akronym [Zie99a].

Entwicklung von Telekommunikationsprotokollen. Eine weitere auf ein bestimmtes Einsatzgebiet eingeschränkte Methode ist HOOD (Hierarchical Object Oriented Design) [HOO95]. HOOD unterstützt nur bestimmte Programmiersprachen (prozedurale Sprachen: Ada, C und FORTRAN; objektorientierte Sprachen: Ada95, C++ und Eiffel). Da diese Verfahren nicht für beliebige Echtzeitsysteme verwendet werden können, werden sie in dieser Arbeit nicht betrachtet.
Ebenso wird auf Methoden, die die Hardware- mit der Software-Entwicklung kombinieren (z.B. MOOSE, Model-based Object Oriented Systems [MGGT96]), nicht weiter eingegangen. Bei diesen Methoden liegt der Schwerpunkt auf der Integration von Software und zumeist proprietärer Hardware (Hardware-Software Co-Design). Der Gegenstand dieser Arbeit ist jedoch die Entwicklung von Software-Systemen, die weitestgehend von der Hardware unabhängig sind.
Bei allen objektorientierten Methoden zur Entwicklung von Echtzeitsystemen gilt, daß diese im Gegensatz zu den "allgemeinen" objektorientierten Vorgehensweisen eine erweiterte Menge von Darstellungsformen besitzen. Diese sind für die Modellierung des asynchronen bzw. synchronen, nebenläufigen und verteilten Verhaltens unter Realzeitbedingungen notwendig.

2.3.4.1 Vergleich objektorientierter Software-Entwicklungsmethoden für Echtzeitsysteme

In diesem Abschnitt werden die Entwicklungsmethoden Real-Time UML, OCTOPUS, RTOOSA, ROOM und COOD untersucht und verglichen. Diese Methoden sind sowohl in Büchern als auch in Artikeln veröffentlicht. COOD ist das Ergebnis einer Promotion.

	Real-Time UML	**OCTOPUS**	**RTOOSA**	**ROOM**	**COOD**
1. Entwicklungsprozeß	inkrementell, iterativ	inkrementell, iterativ	inkrementell, iterativ	inkrementell, iterativ	Wasserfallmodell
2. objektorientierte Notation	UML	OMT, UML	RTOOSA spezifisch	ROOM spezifisch	erweiterte Coad/Yourdon Notation
3. spezifische Notation für Echtzeitsysteme	Diagramme für interne und externe Kommunikation sowie zur Modellierung des Zeitverhaltens	zusätzliche Phase zur Modellierung des Zeitverhaltens mit eigener Notation	Diagramme für externe und interne Kommunikation unter Echtzeitbedingungen	Notation für Zeitverhalten und Kommunikation (Event Handling, Aktoren mit Ports)	integriert in Klassen-, Objekt- und Methodenbeschreibung
4. zusätzliche Elemente zur strukturierten Modellierung	nein	nein	nein	ERM (Entity-Relationship Modelle), Datenfluß-diagramm	nein
5. Unterstützung durch CASE Tools	keine spezifischen	KISS	nein	ROOM-CASE Tool	COODT

Tabelle 2.2: Objektorientierte Entwicklungsmethoden für Echtzeitanwendungen

Im folgenden werden die wichtigsten Merkmale der objektorientierten Methoden zur Entwicklung von Echtzeitsystemen verglichen (siehe Tabelle 2.2):

1. Entwicklungsprozeß
   Mit Ausnahme von COOD propagieren alle Methoden einen inkrementellen und iterativen Software-Entwicklungsprozeß wie dies auch bei anderen "allgemeinen" objektorientierten Vorgehensweisen (z.B. OOAD, OMT, UML oder GUM) üblich ist. COOD hält noch am Wasserfallmodell von Royce und B.W. Boehm [Roy70, Boe76] fest.

   In den Methoden OCTOPUS und COOD wird zunächst die Architektur ohne Berücksichtigung der zeitlichen Bedingungen entwickelt. Die Echtzeitanforderungen werden erst in den letzten Abschnitten des Entwicklungsprozesses bearbeitet.

2. objektorientierte Notation
   Real-Time UML, OCTOPUS und COOD lehnen sich an die Notation bekannter Methoden an. RTOOSA und ROOM verwenden eigene Darstellungen. Seit 1998 wird ROOM an UML angenähert [RS98]. Die ROOM-spezifischen Konstrukte Aktoren (Actors, konkurrente logische Maschinen, als einzelne Klassen oder Subsysteme modelliert), Ports (Schnittstellen der Aktoren), Protokollklassen (Protocol Classes, beschreiben das Verhalten der Ports) und Nachrichten (Messages, Mechanismus zur Kommunikation zwischen den Aktoren) werden in UML eingebracht. Dabei werden sie teilweise entsprechend umbenannt, z.B. werden Aktoren (Actors) zu Kapseln (Capsules) [Lyo98].

3. spezifische Notation für Echtzeitsysteme
   Grundsätzlich werden zwei Darstellungsvarianten zur Modellierung von Echtzeitsystemen angeboten (weitere Erläuterungen siehe Abschnitt 2.3.4.3):

   (a) Modelle zur Beschreibung der systeminternen Kommunikation und der Kommunikation mit der Umgebung des Systems.

   (b) Modelle zur Darstellung der Echtzeitanforderungen an Methoden und Tasks, bzw. des Echtzeitverhaltens des Systems.

   Real-Time UML, OCTOPUS und ROOM unterstützen beide Modellierungsvarianten. Dagegen bietet RTOOSA nur Diagramme zur Beschreibung der Kommunikation. Bei COOD sind Möglichkeiten zur Dokumentation des Zeitverhaltens in die Klassen-, Objekt- und Methodenbeschreibung integriert. Eine Modellierung von harten Echtzeitbedingungen ist nicht vorgesehen [Hüs95].

4. zusätzliche Elemente zur strukturierten Modellierung
   Außer ROOM verwendet keine Methode Modelle der strukturierten Vorgehensweise. Allerdings werden in RTOOSA Diagramme verwendet, die von strukturierten Analysemethoden abgeleitet worden sind. Zur Beschreibung der Anforderungen an das Echtzeitsystem werden Object Relationship Diagramme eingeführt, die weitestgehend den Entity Relationship Diagrammen entsprechen (die Entitäten sind dabei durch Objekte ersetzt). Der Datenfluß zwischen den Objekten wird mit den Object Flow Diagrammen modelliert. Diese leiten sich von den Datenflußdiagrammen ab.

5. Unterstützung durch CASE Tools
   Real-Time UML wird durch kein bestimmtes CASE Tool unterstützt. In [Dou98b] wird angemerkt, daß unterschiedliche CASE Tools verwendet werden können. Allerdings wird kein CASE Tool genannt, das die vollständige Notation der Methode darstellen kann.

   OCTOPUS wird durch das CASE Tool KISS von Nokia unterstützt [AKZ96].

Die Methode ROOM ist sehr stark auf die Verwendung des ROOM-CASE Tool ausgerichtet. Die Anwendung der Methode ist nur sinnvoll in Verbindung mit dem CASE Tool.

Im Rahmen der Entwicklung von COOD wurde das CASE Tool COODT (Concurrent Object Oriented Design Tool) entwickelt, das unter UNIX läuft [Hüs95].

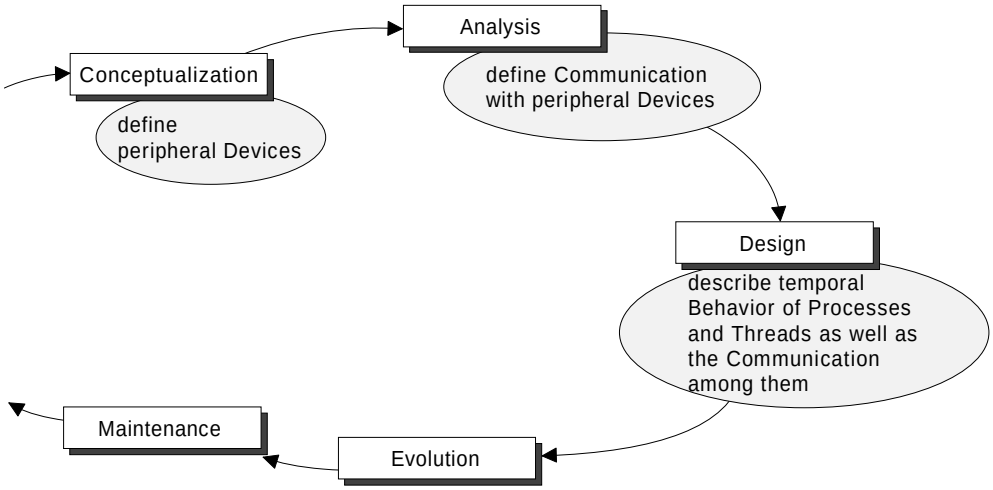


Abbildung 2.12: Entwicklungsprozeß objektorientierter Echtzeitsysteme

2.3.4.2 Prozeß zur Entwicklung von Echtzeitsysteme

Der Prozeß zur objektorientierten Entwicklung von Echtzeitsysteme orientiert sich bei allen in Abschnitt 2.3.4.1 beschriebenen Methoden am "allgemeinen" Vorgehen zur Software-Entwicklung (siehe Abschnitt 2.3.3.1). In diesem Abschnitt wird der in den einzelnen Methoden beschriebenen Vorgehensweisen zu einem Prozeß zusammengefaßt. Daß dies bei objektorientierten Methoden grundsätzlich möglich ist, wird in [Dod96] und [Gra93, HSG96] gezeigt. Die Vereinigung der Methoden OOAD (Booch), OMT (Rumbaugh et al.) und OOSE (Jacobson et al.) zur Unified Modelling Language (UML) [BR95] belegt ebenfalls, daß unterschiedliche Entwicklungsmethoden zusammengefaßt werden können. Daher werden hier die Vorgehensmodelle der verschiedenen Methoden zu einem Software-Entwicklungsprozeß zusammengeführt.
Zusätzlich zum Entwicklungsprozeß der "allgemeinen" Methoden müssen folgende Schritte besonders beachtet werden (siehe Abbildung 2.12):

1. Definition der peripheren Geräte
   In der Konzeptualisierung muß die Peripherie des Echtzeitsystems genau definiert werden. Dies ist bei der Entwicklung von Echtzeitsystemen besonders wichtig, da diese Systeme (z.B. Flugleitsysteme, industrielle Steuerungssysteme oder eingebettete Steuerungen) meist speziell zur Interaktion mit externen Geräten entwickelt werden. Werden von zu Beginn des Entwicklungsprozesses Geräte oder Gruppen von Geräten, mit denen das Echtzeitsystem kommunizieren soll, nicht beachtet, so können diese im weiteren Verlauf des Prozesses nur schwer berücksichtigt werden.

2. Bestimmung der externen und internen Kommunikation
   Nachdem die Peripherie des Systems während der Konzeptualisierung erfaßt worden ist, wird in der Analyse die Kommunikation mit den externen Geräten und die davon abhängige systeminterne Kommunikation beschrieben.

   Die Beschreibung umfaßt externe Ereignisse sowie die Antwortzeiten auf die Ereignisse ebenso wie die Kommunikation zwischen den in Use Case Diagrammen erkannten Aktoren und Use Cases. Im weiteren Verlauf des Entwicklungsprozesses wird auch das Zeitverhalten der Nachrichten zwischen den Subsystemen und Klassen bestimmt. Hierzu können zum Beispiel die Kollaborationsdiagramme (Collaboration Diagrams) von Real-Time UML verwendet werden.

3. Zeitanforderungen an die Prozesse und Tasks des Systems
   Nachdem im Design die Architektur des Systems zunächst ohne Berücksichtigung der zeitlichen Anforderungen an das System entwickelt worden ist, wird nun das System entsprechend den geforderten Zeitlimits optimiert.

   Dazu werden zunächst die Tasks (Prozesse und Threads) ermittelt. Danach wird das Zeitverhalten der Tasks und der Kommunikation und Synchronisation der Tasks untereinander an die in der Analyse spezifizierten Zeitanforderungen angepaßt. Bei Multitaskingsystemen können entsprechende Scheduling Verfahren, wie z.B. das Rate Monotonic Scheduling (RMS), verwendet werden [Gal95].

2.3.4.3 Notationen und Diagramme zur Modellierung von Echtzeitsystemen

Für die Entwicklung von objektorientierten Echtzeitsystemen werden zusätzlich zu den in Abschnitt 2.3.3.2 aufgeführten Diagrammen weitere Darstellungsformen verwendet. Diese Diagramme dienen hauptsächlich zur Modellierung der Systemperipherie, der externen und internen Kommunikation und des Echtzeitverhaltens (siehe Abschnitt 2.3.4.2).

2.3.4.3.1 Definition der peripheren Geräte

Zu Beginn des Entwicklungsprozesses müssen die Geräte, mit denen das Echtzeitsystem zusammenarbeiten soll, bestimmt werden. Daher sollte die verwendete Hardware schon in der initialen Aufgabenbeschreibung (Problem Statement) möglichst vollzählig aufgeführt sein. In den Use Case Modellen sollten die Geräte als Aktoren vorkommen.
Zur Entwicklung von Echtzeitsystemen ist es sehr wichtig, neben einer reinen Aufzählung der verwendeten Hardware auch die Beziehungen und die strukturelle Anordnung der Geräte zu verdeutlichen. Dies kann auf unterschiedliche Weise erfolgen.
Zum einen kann die Peripherie durch formlose Prinzipschaubilder, auf denen alle Geräte erfaßt sind, dargestellt werden [Hüs95]. Sinnvollerweise wird in einem Prinzipschaubild der zu steuernde Prozeß veranschaulicht.
Eine andere Möglichkeit ist, die Geräte hierarchisch zu gliedern. Dazu werden die Geräte in den eigentlichen Steuerungsrechner mit Echtzeit-Software, die Schnittstellen zu den Sensoren und Aktuatoren, die Sensoren und Aktuatoren sowie die Prozeßperipherie eingeteilt. Beispiele für diese Modelle sind das Schalenmodell [Ell94] oder formlose hierarchische Darstellungen [SGW94].

2.3.4.3.2 Bestimmung der externen und internen Kommunikation

Bei Echtzeitsystemen müssen neben der Modellierung der Kommunikationsbeziehungen innerhalb des Systems folgende weitere Abläufe beschrieben werden:

- Kommunikation mit Hardware-Komponenten
  Bestimmung mit welchem externen Gerät oder Bauteil des Rechners, auf dem das System abläuft, Informationen ausgetauscht werden.

- Reaktionen auf Ereignisse
  Definition der Reaktionszeit und wie auf bestimmte externe und interne Ereignisse reagiert werden muß.

Diagramm	**Verwendungszweck**
Kontextdiagramme mit Nachrichten, (auch als External Interface Diagram, EID, bezeichnet)	Definition der Grenze zwischen dem System und der Peripherie; Beschreibung der Nachrichten zwischen den Aktoren und dem System
Kollaborationsdiagramme oder Sequenzdiagramme mit Zeitbedingungen	diese Diagramme zur Beschreibung des dynamischen Verhaltens erhalten zusätzliche Zeitbedingungen
(externe) Ereignisse und Ereignishierarchien (auch als External Event / Response List, EERL, bezeichnet)	Erfassen aller Ereignisse (externe sowie interne) und Gliederung in Hierarchie, Bestimmung der Antwortzeiten und Reaktionen des Systems

Tabelle 2.3: Diagramme zur Beschreibung der Kommunikationsbeziehungen

Die in Tabelle 2.3 aufgeführten Diagramme leiten sich von Darstellungsformen strukturierter und objektorientierter Software-Entwicklungsmethoden ab:
Die Kontextdiagramme stammen von der Methode SA/SD [AKZ96], bzw. strukturierten Echtzeitvorgehensweisen (RTSA) [Edw93] ab. In Real-Time UML, OCTOPUS und RTOOSA werden Kontextdiagramme [3] zur Ergänzung objektorientierter Beschreibungsformen, wie z.B. Use Case Modell verwendet [Ell94, AKZ96, Dou98b].
Die Kollaborationsdiagramme (Objektdiagramme nach [Boo94]) oder Sequenzdiagramme (Interaktionsdiagramme nach [Boo94]) leiten sich von "allgemeinen" objektorientierten Entwicklungsmethoden ab. Sie sind jedoch um Möglichkeiten zur Darstellung von Zeitbedingungen erweitert [AKZ96, Dou98b].
Ereignishierarchien leiten sich sowohl von strukturierten Methoden als auch von objektorientierten Vorgehensweisen ab [Ell94, Dou98b]. Die Hierarchien können als Diagramme, wie z.B. in Real-Time UML, oder als Tabellen (RTOOSA) notiert werden. Durch diese können die Ereignisse klassifiziert werden sowie die Reaktionen auf diese bestimmt werden.
Zusätzlich zu einer speziellen Notation für Echtzeitsysteme bietet Real-Time UML eine Auswahl an objektorientierten Mustern (siehe Abschnitt 2.1), mit denen das Verhalten und die Kommunikation des Systems modelliert werden können [Dou98b].

2.3.4.3.3 Zeitanforderungen an die Prozesse und Tasks des Systems

Die Beschreibung des Zeitverhaltens ist eine zentrale Aufgabe der Entwicklung von Echtzeitsystemen (siehe Klassifikation der Zeitbedingungen in Abschnitt 2.3.2).

[3] Kontextdiagramme werden auch in UML verwendet.

Zum einen muß das zeitliche Verhalten der Prozesse und Tasks und deren Klassen und Objekte (mit Hilfe graphischer Notation) modelliert werden. Zum anderen müssen die Scheduling-Verfahren, nach denen die Tasks den Prozessoren zeitlich zugeordnet werden, definiert werden.

Modellierungsziel	**Diagramme**	**Methoden**	**Verwendungzweck**
Einteilung des Systems in Prozesse und Tasks	Task Diagrams	Real-Time UML	Beziehungen zwischen den Tasks und Zuordnung der Ereignisse zu den Tasks
	Interaktionsdiagramme (Kollaborationsdiagramme) erweitert mit Taskgrenzen	OCTOPUS	Beschreibung der Zusammenhänge zwischen den Objekten der Tasks
Modellierung des Zeitverhaltens	Timing Diagrams	Real-Time UML	zeitliche Abfolge der Zustände einer Task
	Sequential Models, Event Driven Models	ROOM	Zuordnung der Ereignisse und Zeiten zu Prozessen und Tasks
	Interaktionsdiagramme (Kollaborationsdiagramme) erweitert mit Task-Grenzen und Abschätzung des Zeitbedarfs	OCTOPUS	Beschreibung der zeitlichen Zusammenhänge der Objekte in den Tasks
	Worst Case Event Sequences / Performance Charts	OCTOPUS	Ermittlung des max. Zeitbedarfs, Basis für Tuning
Modellierung der Nebenläufigkeit	Concurrent Collaboration Diagrams	Real-Time UML	Interaktionen konkurrenter Objektgruppen
	Concurrent Timing Diagrams	Real-Time UML	zeitliche Abhängigkeiten nebenläufiger Tasks
	Concurrent State Diagrams	Real-Time UML	nebenläufige Zustände in *Active Objects*

Tabelle 2.4: Diagramme zur Beschreibung der Zeitanforderungen

Folgende Möglichkeiten zur zeitabhängigen Modellierung können unterschieden werden (siehe Tabelle 2.4):

- Einteilung des Systems in Prozesse und Tasks
  In Real-Time UML wird darauf verwiesen, daß Tasks (bzw. Prozesse) in Klassen- und Objektdiagrammen implizit vorkommen, d.h. Prozesse und Tasks bestehen aus Gruppen von Klassen oder Objekten. Das Verhalten der Tasks wird von sogenannten *Active Objects* bestimmt, die die *Passive Objects* aufrufen. In COOD wird ebenfalls zwischen aktiven und passiven Klassen unterschieden. Das dynamische Verhalten der Objekte wird in Kollaborations- und Sequenzdiagrammen dargestellt. Die Beziehungen der Tasks untereinander werden in Task Diagrams modelliert. In diesen Diagrammen werden auch die Ereignisse den Tasks zugeordnet.

  OCTOPUS bietet keine speziellen Diagrammtypen zur Aufteilung der Objekte auf bestimmte Tasks (in OCTOPUS Threads genannt) an. Die Zusammenhänge zwischen den Objekten der einzelnen Tasks und das Verhalten der Objekte wird durch Interaktionsdiagramme (Object Interaction Diagrams, entsprechen Kollaborationsdiagrammen in UML) modelliert. Die Bestimmung der Objekte, die zu einer Task gehören,

geschieht durch eine iterative Vorgehensweise. Nach dieser wird von den Objekten (entsprechen den *Active Objects* in Real-Time UML) ausgegangen, die auf Ereignisse reagieren. Threads werden dadurch gebildet, daß zu diesen Objekten die weiteren kollaborierenden Objekte (in Real-Time UML *Passive Objects* genannt) hinzugenommen werden. Diese Zuordnung der Ereignisse zu Objekten wird so lange ausgeführt, bis alle Ereignisse zugeteilt sind.

- Modellierung des Zeitverhaltens
  Die Timing Diagrams von Real-Time UML stellen eine zeitbezogene Variante der Zustandsdiagramme dar [Har87]. Mit einem Zeitstrahl wird die zeitliche Abfolge der Zustände innerhalb einer Task dargestellt. Timing Diagrams leiten sich von den Impulsdiagrammen der Digitaltechnik ab.

  Die Zuordnung der Ereignisse (und der Zeitpunkte der Bearbeitung der Ereignisse) zu den Zuständen eines Prozesses oder einer Task kann mit den Sequential Models und Event Driven Models von ROOM erfolgen.

  In OCTOPUS werden die Grenzen der Threads sowie eine Abschätzung des Zeitbedarfs für die Ausführung von Methoden zusätzlich in Interaktionsdiagramme eingetragen. (Real-Time UML empfiehlt ebenfalls die Verwendung von Kollaborationsdiagrammen für diesen Zweck.) Diese Darstellungsvariante unterstützt auch die Zuordnung der Objekte zu den Prozessen und Tasks. Besondere Berücksichtigung finden Objekte, die von mehreren Threads genutzt werden (Shared Objects List).
  Eine noch deutlichere Darstellung der zeitlichen Abhängigkeiten erfolgt in den Worst Case Event Sequences / Performance Charts. Diese entsprechen im Aussehen den Timing Diagrams von Real-Time UML. Allerdings werden nicht die einzelnen Zustände einer Task, sondern die Aktivitäten mehrerer Prozesse und Tasks dargestellt. Dabei können nebenläufig oder parallel arbeitende Threads modelliert werden. Diese Diagrammform wird zur Ermittlung des maximalen Zeitbedarfs und als Grundlage für Optimierungsmaßnahmen (Tuning) verwendet.

- Modellierung der Nebenläufigkeit
  Die meisten Diagramme zur Darstellung der Nebenläufigkeit leiten sich von zeitabhängigen Modellen ohne Nebenläufigkeit ab. Exemplarisch sollen hier drei Diagrammvarianten aus Real-Time UML vorgestellt werden. Die Concurrent Collaboration Diagrams zeigen die Interaktionen konkurrenter Objektgruppen. Die zeitlichen Abhängigkeiten nebenläufiger Tasks (Task Rendezvous) werden mit Concurrent Timing Diagrams modelliert. Concurrent State Diagrams dienen zur Beschreibung nebenläufiger Zustände in den *Active Objects*.

  Die Diagramme von OCTOPUS zur Darstellung des Zeitverhaltens lassen sich ebenso wie die Real-Time UML Diagramme zur Modellierung nebenläufigen Verhaltens verwenden. In ROOM werden Szenarien (Mischung von Kollaborations- und Zustandsdiagrammen) zur Beschreibung sowohl nichtnebenläufiger als auch nebenläufiger Vorgänge eingesetzt.

Einen wichtigen Einfluß auf das Zeitverhalten eines Systems hat das angewendete Scheduling-Verfahren. In COOD werden unterschiedliche Möglichkeiten diskutiert: Rate-Monotonic Scheduling, Zeitschrankenverfahren (Earliest Deadline First) und Scheduling mit Spielraum (Zeitfenster, Laxity). Entsprechend dieser Verfahren wird die Priorität der Tasks vergeben (statische Priorität bei Rate-Monotonic Scheduling, dynamische Priorität bei Zeitschrankenverfahren und bei Scheduling mit Spielraum).

Real-Time UML schreibt kein besonderes Scheduling-Verfahren vor.
In OCTOPUS wird ein Verfahren zur Bestimmung der Prozeßprioritäten auf der Grundlage des Rate-Monotonic Scheduling beschrieben. Ausgangsbasis sind Tabellen (Preepmtion Tables) zur Festlegung, welche Task welche andere Task unterbrechen darf. Daraus werden dann die genauen Prioritäten der einzelnen Tasks mit Hilfe einer Priority Derivation Table abgeleitet.
Zusätzlich zu den Task-Prioritäten bietet ROOM die Möglichkeit, auch an Nachrichten und Ereignisse Prioritäten zu vergeben.

2.3.5 Echtzeitbetriebssysteme

Die Systemschnittstelle von Betriebssystemen wird durch die POSIX-Standards (Portable Operating System Interface X) normiert. Diese POSIX-Normen werden von Gremien der IEEE entwickelt und von ANSI (American National Standards Institute) und ISO (International Organization for Standardization) standardisiert [SPG91, Tan92, Bac86, LMKQ89]. POSIX konforme Betriebssysteme sind hauptsächlich UNIX Derivate, wie z.B. Solaris von Sun Microsystems oder LINUX. Daneben orientieren sich aber auch weitere Betriebssysteme (z.B. VMS) an dem Standard.

Basis von Echtzeitsystemen sind zumindest teilweise echtzeitfähige Betriebssysteme (mit weichen oder harten Echtzeiteigenschaften). Diese Betriebssysteme bieten Dienste, die die Abarbeitung der Software in Realzeit unterstützen [LA90].

	Funktion	Beschreibung
zwingend nach POSIX.4	Warteschlangen für Echtzeitsignale	Erweiterung der nicht-Echtzeitsignale um einfache Echtzeitsignale mit Prioritäten
optional nach POSIX.4	zusätzliche Funktionen für Echtzeitsignale	erweiterte Möglichkeiten zum Behandeln von Echtzeitsignalen
	synchrone I/O	blockierender synchrone I/O für Echtzeitprozesse
	asynchrone I/O	priorisierter asynchrone I/O (queued)
	Nachrichten mit Wartenschlangen (Message Queues)	Austausch von Nachrichten mit hoher Priorität zwischen Echtzeitprozessen
	Scheduling mit Prioritäten	Steuerung des Scheduling durch Vergabe von Prioritäten
	Speicherallokation	Allokation von Speicher zur Vermeidung von Auslagerung oder Seitentausch
	Semaphore	Semaphore für Prozesse mit Echtzeitpriorität
	Echtzeit Timer	hochpräzise Zeitgeber zur Steuerung von Tasks
	Shared Memory	Zugriff auf gemeinsamen Speicher in Echtzeit

Tabelle 2.5: Zwingende und optionale Bestandteile des POSIX.4 Standards

POSIX.4 [4] ist der Standard für Echtzeitbetriebssysteme [IEE93, Gal95]. Die in POSIX.4 definierten Funktionen sind in Tabelle 2.5 zusammengefaßt. Als besonders wichtig für die

[4] POSIX.4 wurde 1993 in POSIX 1003.1b-1993 umbenannt. Trotzdem wird meist die Bezeichnung PO-

Entwicklung gelten die Bearbeitung von Echtzeitsignalen, Scheduling-Verfahren sowie hochpräzise Zeitgeber [Rze93]. Diese Dienste sind Grundlage der Modellierung der Zeitanforderungen [RS98] (siehe Abschnitt 2.3.4.3.3).
Je nach der garantierten Einhaltung von Zeitgrenzen werden Echtzeitbetriebssysteme in harte und weiche Systeme eingeteilt (siehe Abschnitt 2.3.2.1). An POSIX.4 orientieren sich unter anderem die harten Echtzeitbetriebssysteme VX-Works, LynxOS und QNX und weiche Systeme wie Solaris oder Linux.
Echtzeitfähige Betriebssysteme werden meistens bei größeren Echtzeitsystemen wie industriellen Steuerungen verwendet [Wil91, Nic94, Gal95]. Kleinere Echtzeitsysteme (z.B. einfache eingebettete Systeme, wie digitale oder analoge E/A Module) werden zum Teil ohne Betriebssystem realisiert. Hierbei greift das Echtzeitsystem direkt auf die Hardware zu, ohne daß ein Betriebssystem die Verwaltung der Ressourcen übernimmt.

SIX.4 in der Literatur verwendet.

2.4 Industrielle Steuerungssysteme

Die folgenden Kapitel beschreiben nach einem Überblick über die Geschichte und den aktuellen Stand industrieller Steuerungssysteme die wichtigsten Steuerungstypen [RNS93, Wec96a, Wec96b]: Robotersteuerung (RC, Robot Control), speicherprogrammierbare Steuerung bzw. SPS (PLC, programmable Logic Controller) und Numerische Steuerung (NC, Numerical Control).

Anschließend werden offene Steuerungssysteme, die meist aus Standard-Hardware wie Industrie-PCs bestehen, vorgestellt.

Den Abschluß des Kapitels bildet eine kurze Einführung in industrielle Fertigungssysteme, in denen die vorgestellten Industriesteuerungen zum Einsatz kommen.

2.4.1 Geschichte und aktuelle Entwicklungen

Dieser Abschnitt ist in drei Einheiten aufgeteilt. Zunächst wird die historische Entwicklung der Steuerungssysteme zusammengefaßt. Anschließend folgt eine Darstellung der aktuellen Steuerungssysteme sowie die Einordnung der Steuerungssysteme in die Unternehmenshierarchie.

2.4.1.1 Historische Entwicklung

Kurz nach der Einführung der ersten Computer entstanden die ersten rechnerbasierten Systeme zur automatischen Steuerung von Produktionsprozessen:

Im Jahr 1948 wurden die ersten Konzepte für numerische Werkzeugmaschinensteuerungen im Auftrag der U.S. Air Force erarbeitet. Basierend auf diesen Ideen wurde 1952 die erste numerische Steuerung mit Elektronenröhren und elektromagnetischen Relais am Massachusetts Institute of Technology (MIT) gebaut. Über Lochstreifen konnte eine Fräsmaschine gesteuert werden [Dor85, Wec96a].

Die ersten Prozeßrechner (Process Control Computers) Bourroughs E-101 und Bendix G15 folgten 1955/56. In den 60er Jahren wurde bei Digital Equipment die Serie der Programmable Digital Processor[s] (PDP) als Klein-Prozeßrechner entwickelt. Der bekannteste Rechner dieser Serie, die PDP 8, wurde 1965 vorgestellt.

Der erste kommerzielle Industrieroboter wurde 1959 von der Firma Planet Corperation gebaut. Die Play-back-Steuerungen der Firma Tralfa aus Norwegen aus dem Jahr 1966 waren die ersten in größerem Umfang in der industriellen Fertigung eingesetzten Robotersteuerungen. Diese Systeme wurden für einfache Bewegungsaufgaben (Spritzlackieren oder Bahnschweißen) verwendet. Play-back-Steuerungen werden dadurch programmiert, daß die gewünschte Bahn (Trajektorie) zunächst mit dem Roboterarm manuell geführt abgefahren wird. Die Steuerung speichert sich dabei die Bahnpunkte und kann diese anschließend automatisch abfahren. In der Folgezeit wurden Robotersteuerungen mit speziellen Roboterprogrammiersprachen entwickelt. [DH91]

Die Markteinführung der ersten speicherprogrammierbaren Steuerungen (SPS) erfolgte zu Beginn der 70er Jahre [Wec96a]. Diese freiprogrammierbaren Steuerungen haben aufgrund der flexiblen Einsatzmöglichkeiten die festprogrammierten Schützsteuerungen [5] abgelöst.

[5] Relais werden in der Automatisierungstechnik als Schütze bezeichnet.

2.4.1.2 Aktuelle industrielle Steuerungssysteme

Zur Steuerung von Automatisierungskomponenten werden heute unterschiedliche Steuerungssysteme verwendet. Neben den wichtigsten Typen von Steuerungen, den Robotersteuerungen, speicherprogrammierbaren Steuerungen und numerischen Steuerungen, sind zusätzliche spezielle Systeme im Einsatz (z.B. Steuerungen für Materialtransportsysteme oder für Sondermaschinen).
Steuerungssysteme können auf unterschiedliche Weise realisiert werden. Bis zu Beginn der 90er Jahre dominierten proprietäre Kompaktsysteme den Markt [RNS93, Bar94, McK95]. Diese bestehen aus einer speziell vom Hersteller entwickelten Hardware als Steuerungsrechner und einer meist ebenfalls vom Hersteller entwickelten Steuerungs-Software. Beispiele für solche Systeme sind die Robotersteuerungen rho3 (Robert Bosch AG) und ACR (Siemens AG) oder die speicherprogrammierbaren Steuerungen der S5 oder S7 Serie (Siemens AG) sowie die numerischen Steuerungen Sinumeric (Siemens AG).
Seit Mitte der 90er Jahre werden zunehmend Standardrechner (PCs und Workstations) anstelle proprietärer Hardware verwendet. Die Gründe hierfür sind zum einen die geringen Kosten für Standard-Hardware, zum anderen die sich permanent verbessernde Leistungsfähigkeit von neu entwickelten PCs und Workstations [KGSL97a]. Beispiele für Steuerungen, die auf Standard-Hardware basieren, finden sich in Abschnitt 2.4.5.2.
Außerdem sind in verschiedenen Forschungsprojekten Architekturen und Modelle für offene Steuerungssysteme entwickelt worden. Basis dieser offenen Systeme sind meist Standardrechner. Einige dieser Projekte werden in Abschnitt 2.4.5.1 vorgestellt.

2.4.1.3 Einordnung der Steuerungssysteme in die Unternehmenshierarchie

Die Rechnersysteme in einem Unternehmen können entsprechend der Struktur des Betriebs hierarchisch dargestellt werden [Sch89, Sch91]. Industrielle Steuerungssysteme sind in diesem Modell auf der unteren Ebene eingeordnet. Sie werden also in den operativen Bereichen eines Unternehmens eingesetzt, die dem Produktionsprozeß am nächsten sind.

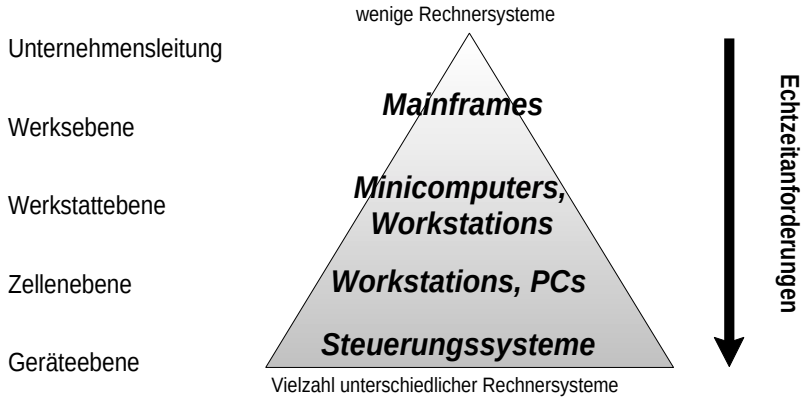


Abbildung 2.13: Unternehmens- und Rechnerhierarchie

Charakteristische Rechnersysteme auf den verschiedenen Ebenen sind [RNS93]:

1. Großrechner mit Terminals oder leistungsfähigen Endgeräten (z.B. PCs)
   Diese Rechner finden sich auf der Ebene der Unternehmens- und der Werksleitung. Mit Hilfe dieser Systeme werden die strategischen Ziele des Unternehmens verfolgt, wie z.B. die Maximierung des Gewinns oder die Stärkung der Wettbewerbsposition.

Auf Werksebene werden Produktionsplanungs- und Steuerungssysteme (PPS) zur Grobdisposition der Auftragsbearbeitung eingesetzt.

2. Minicomputer und Workstations
   Werkstattsteuerungssysteme (WSS) auf der Werkstattebene dienen zur Auftragsverwaltung und Feindisposition sowie der Auswertung erfaßter Daten (z.B. Qualitätsdaten).

3. Workstations und PCs
   Die Aufgabe dieser Zellenrechner ist die Auftragsdurchlaufsteuerung, die Betriebsdatenerfassung (BDE) und die Koordinierung der einzelnen Gerätesteuerungen (DNC, Direct Numerical Control).

4. Industrielle Steuerungssysteme
   Die klassischen Gerätesteuerungen sind Robotersteuerungen (RC), speicherprogrammierbare Steuerungen (SPS) und numerische Steuerungen (NC). Daneben gibt es für spezielle Systeme und Sondermaschinen besonders angepaßte Steuerungen (z.B. Steuerungen auf industrietauglichen Workstations oder PCs).

2.4.2 Robotersteuerungen für Industrieroboter

Industrielle Robotersteuerungen dienen zur Steuerung und Überwachung eines oder mehrerer Roboterarme. Industrieroboter sind universell einsetzbare Bewegungsautomaten. Die Bewegungen werden vom Anwender frei programmiert und anschließend automatisch wiederholt [6]. Industrieroboter haben mindestens vier translatorische (prismatische) oder rotatorische Achsen. Als Endeffektoren besitzen sie Greifer oder Werkzeuge, mit denen sie Handhabungs- oder Fertigungsaufgaben übernehmen können [SB96].
Im Gegensatz zu Einlegegeräten haben Industrieroboter geregelte Antriebe und eine größere Anzahl beweglicher Achsen, mit denen sie Bahnen im Raum (Trajektorien) abfahren können. Einlegegeräte haben meist nur (ungeregelte) pneumatische Antriebe, wobei die Bewegungsendpunkte durch mechanische Endschalter festgelegt werden.
Als ortsfeste Roboter unterscheiden sich Industrieroboter von autonomen Systemen, die sich frei bewegen (zum Teil werden Industrieroboterarme als Greifwerkzeuge an autonomen Systemen angebracht [DEF+96, RNS93]).

Im folgenden werden die Arten von Robotersteuerungen und deren Steuerungskomponenten vorgestellt. Anschließend folgt eine Übersicht über die am häufigsten verwendeten Roboterkinematiken und die Möglichkeiten zur Programmierung von Robotersteuerungen.

2.4.2.1 Arten von Robotersteuerungen

Roboter können auf unterschiedliche Arten gesteuert werden [Hun91, RNS93].

Die einfachste Möglichkeit einen Roboterarm zu kontrollieren, ist die Verwendung von mechanischen Stoppern als feste Endpunkte der Roboterbewegung. Armbewegungen können nur von einem Stopper zu einem anderen Stopper durchgeführt werden. Diese Technik wird

[6] Industrieroboter werden auch als vom Menschen (Master) interaktiv gesteuerte Slave-Systeme (Manipulatoren) verwendet. Dabei werden die Bewegungen nicht von einem Programm gesteuert sondern direkt von einem Bediener. Diese Manipulatoren kommen beispielsweise beim Umgang mit radioaktivem Material oder für schwere und gesundheitsschädliche Arbeiten in der Industrie sowie in der Weltraumforschung zum Einsatz.

bei Robotersteuerungen mittlerweile nicht mehr verwendet, sondern findet lediglich noch bei Einlegegeräten Anwendung [RNS93].

Dagegen können durch Servosteuerungen beliebige Punkte im Raum frei vom Roboter angefahren werden. Diese Steuerungen haben einen geschlossenen Regelkreislauf zur Überwachung der Position des Roboterarms (d.h. der Roboterarm wird durch Rückmeldeinformationen kontrolliert).

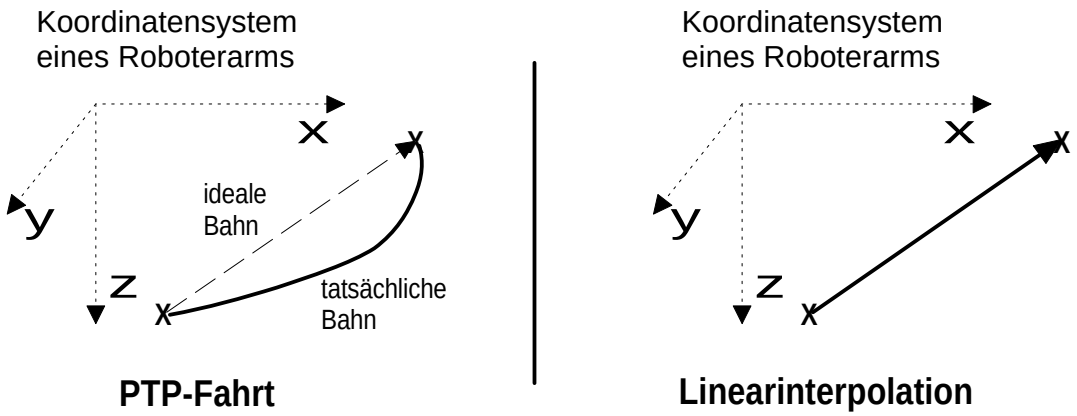


Abbildung 2.14: PTP-Interpolation und Linearinterpolation (Kontinuierliche Bahnsteuerung)

Drei Arten von Servosteuerungen werden unterschieden:

1. Vielpunktsteuerung (Play-back-Steuerung)
   Bei diesem Verfahren geschieht die Programmierung durch Handführung des Roboterarms. In bestimmten Zeitabständen (zyklisch) werden Bahnpunkte dieser Bewegung abgespeichert. Anschließend wird die vorgegebene Bahn automatisch abgefahren (Playback).

   Der Vorteil dieser Art von Programmierung ist, daß das Steuerungssystem einfach zu entwickeln ist.

   Allerdings wird ein großer Programmspeicher für die Zwischenpositionen benötigt. Ebenso ist es nur schwer möglich, die Parameter eines gespeicherten Bewegungsablaufs nachträglich zu modifizieren (z.B. Änderung der Geschwindigkeit). Daher wird dieser Steuerungstyp heute nur noch selten verwendet.

2. Punkt-zu-Punkt Steuerung (Point-to-point, PTP)
   Die Bahnpunkte werden in Achskoordinaten (bzw. Motorkoordinaten), die von der jeweiligen Roboterkinematik abhängen, vorgegeben. Daher bewegt sich der Roboterarm auf einer beliebigen Trajektorie von einer Position zur nächsten. Eine PTP-Bewegung mit fest vorgegebenen Bahnpunkten wird von unterschiedlichen Roboterkinematiken auf jeweils anderen Bahnen abgefahren [Bei90].

   Diese Steuerungsart ist besonders für Pick-and-place Operationen (Transport von Werkstücken von einem Platz zu einem anderen) geeignet.

   Bei einer einfachen PTP-Bewegung fährt jede Achse die maximal erlaubte Geschwindigkeit, wodurch die einzelnen Achsen zeitversetzt an ihrer Zielposition ankommen können. Dagegen beenden bei einer synchronen PTP-Bewegung alle Achsen die Bewegung gleichzeitig.

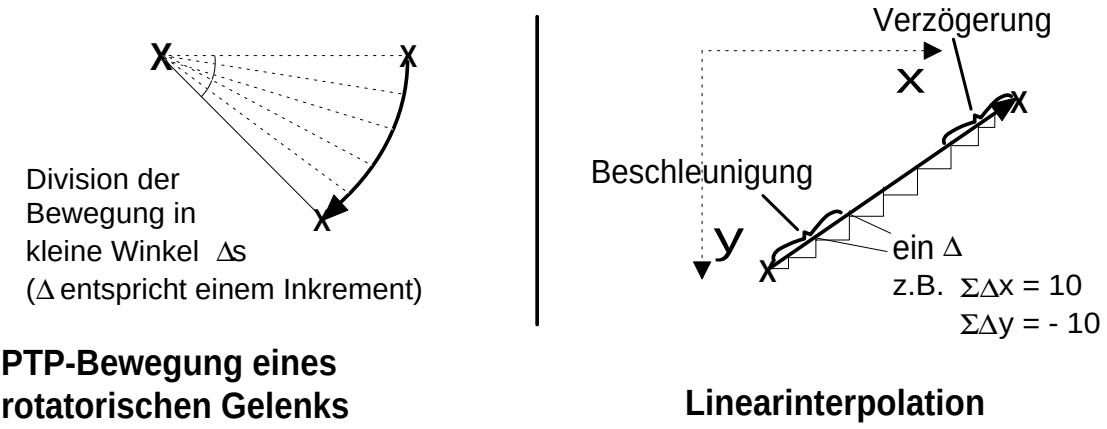


Abbildung 2.15: Bahnstützpunkte bei einer PTP- und einer Linearbewegung (Kontinuierliche Bahnsteuerung)

3. Kontinuierliche Bahnsteuerung (Continuous Path Control)
   Der Endeffektor des Roboters bewegt sich auf einer vordefinierten Bahn im Raum. Diese Trajektorie wird in geräteunabhängigen Raumkoordinaten vorgegeben. Für eine Linearbewegung können beispielsweise der Anfangs- und Endpunkt im Raum, die Geschwindigkeit sowie die Beschleunigung und Verzögerung angegeben werden.

   Die Bahnstützpunkte der Roboterbewegung werden durch Interpolation erzeugt. Unterschiedliche Interpolationsarten (z.B. Linear-, Kreis- und Spline-Interpolation) ermöglichen entsprechende Roboterbewegungen (siehe Abbildung 2.15).

   Diese Bewegungssteuerung eignet sich beispielsweise besonders für Anwendungen wie Schweißen oder Lackieren.

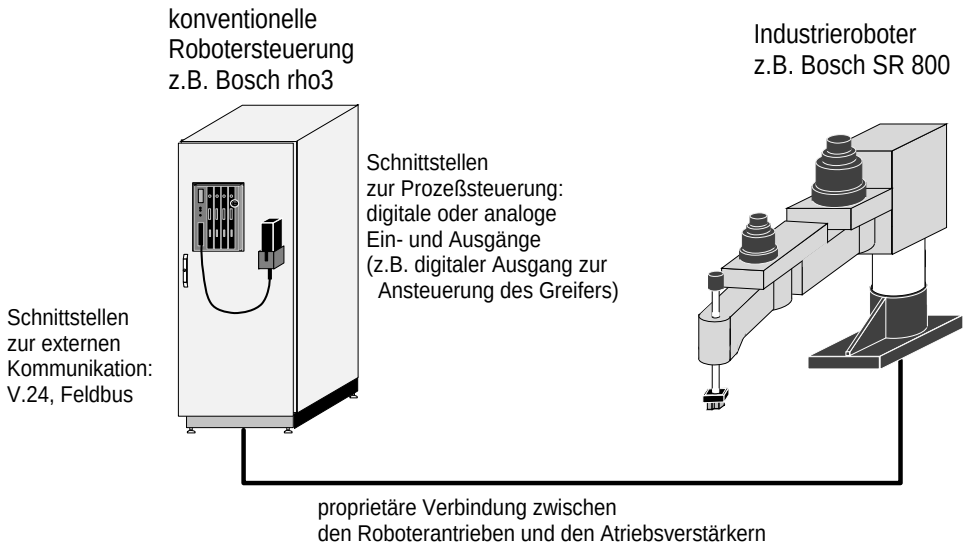


Abbildung 2.16: Konventionelle Robotersteuerung

2.4.2.2 Robotersteuerungskomponenten

In diesem Abschnitt wird zuerst der Aufbau einer Robotersteuerung und anschließend die Bearbeitung einer Bewegungsanweisung vorgestellt. Den Abschluß bildet eine Einführung in die Problematik der Mehrdeutigkeit von Rücktransformationen bei der Interpolation von Bewegungsanweisungen.

2.4.2.2.1 Aufbau

Eine konventionelle Robotersteuerung läuft auf einer proprietären Hardware ab (siehe Abbildung 2.16). Diese besteht aus einer Zentraleinheit, auf der die eigentlichen Roboterprogramme abgearbeitet werden, und den Antriebsverstärkern, die die Bewegungen der Antriebe regeln.

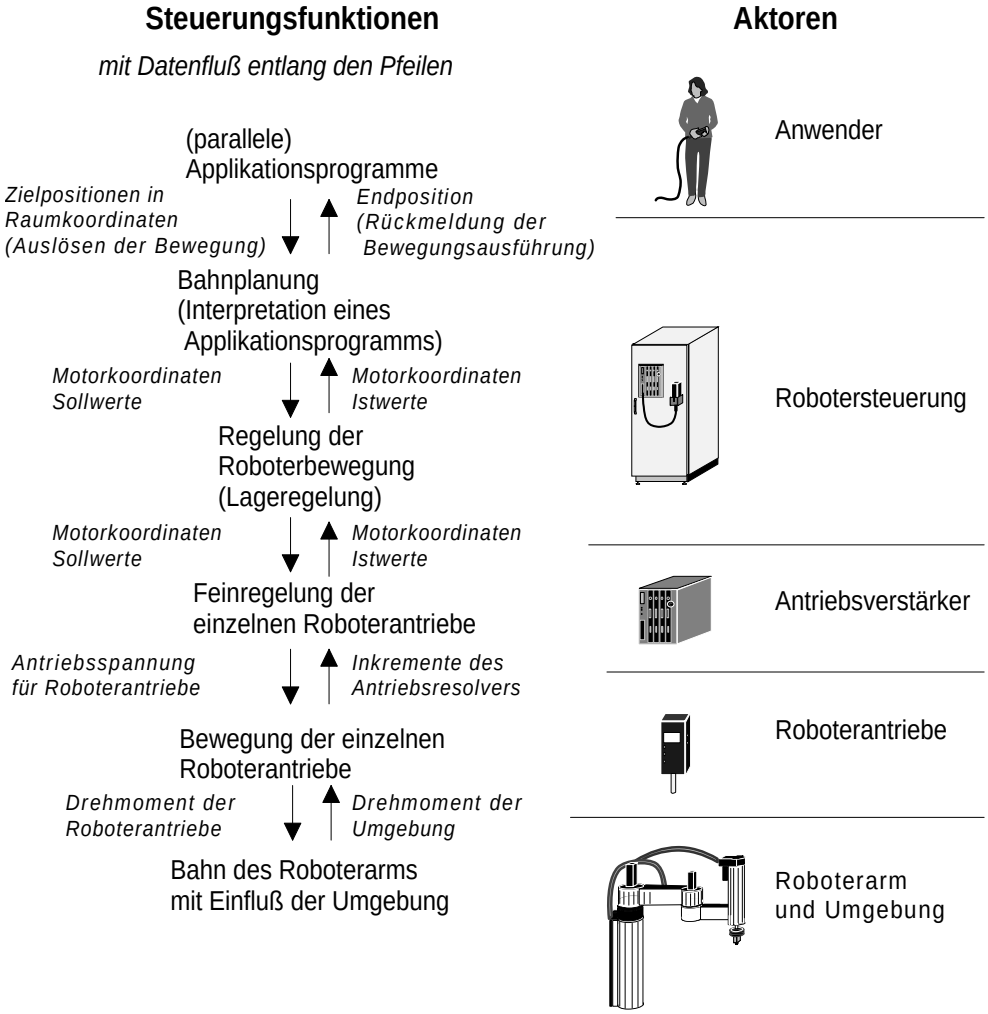


Abbildung 2.17: Hierarchischer Aufbau von Robotersteuerungen

Robotersteuerungen besitzen eine hierarchische Struktur (siehe Abbildung 2.17) [Jar84, BTT87, RNS93, McK95].
Auf oberster Ebene befinden sich die vom Anwender geschriebenen Applikationsprogramme mit einer Folge von Operationen. Diese umfassen Bewegungsoperationen sowie weitere Operationen zur Steuerung der zusätzlichen Schnittstellen der Robotersteuerung [7].

[7] Durch digitale Ein- und Ausgänge können z.B. ein Greifer am Roboterarm oder weitere Sensoren an-

Die Operationen der Applikationsprogramme werden von der darunterliegenden Schicht (Bahnplanung) interpretiert (Bei den heute marktgängigen Steuerungen werden nicht nur Bewegungsoperationen, sondern auch Vorgänge wie digitale Verknüpfungen zeitaufwendig interpretiert.) [RNS93, GW94]. Das Ergebnis der Bewegungsinterpretationen ist eine zeitlich diskrete Folge von Bahnstützpunkten (Interpolation), die die Bewegung des Roboterarms beschreibt. Diese Bahnstützpunkte werden in Motorkoordinaten umgewandelt (Rückwärtstransformation, RT).

Die Regelung der Roboterbewegung sorgt dafür, daß die einzelnen Bahnstützpunkte der Roboterbahn angefahren werden. Dazu vergleicht sie die Istwerte mit den Sollwerten (in Motorkoordinaten). Bei einer Abweichung wird entsprechend gegengeregelt. Da die Lage aller Achsen (bzw. Antriebe) des Roboterarms geregelt wird, heißt dieses Vorgehen Lageregelung. Sie ist meist als einfache P-Regelung (Parallelregelung) realisiert.

Jeder einzelne Antriebsverstärker regelt in einer Feinregelung die Bahn des jeweils zugeordneten Antriebs. Die einzelnen Antriebe (bzw. Antriebsverstärker) werden von der Lageregelung überwacht. Am Roboterarm können Drehmomente entgegen der Armbewegung wirken (z.B. Masseträgheit). Diese Momente müssen durch die Regelung der Roboterantriebe entsprechend kompensiert werden.

Die Antriebe sind meist über Getriebe mit hoher Übersetzung (z.B. Harmonic Drive Getriebe mit einem Übersetzungsverhältnis von 1 : 100) mit den Achsen des Roboterarms gekuppelt. Diese hohe Übersetzung ermöglicht, Bahnen mit großer Genauigkeit nachzufahren.

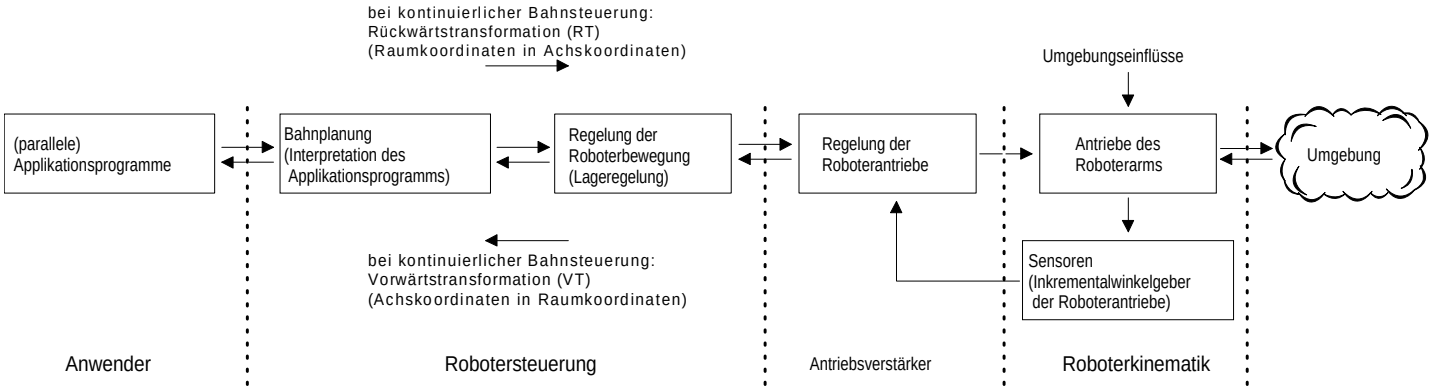


Abbildung 2.18: Interpretative Bearbeitung einer Bewegungsanweisung

2.4.2.2.2 Bearbeitung einer Bewegungsanweisung

Die Roboterbewegung wird in den Applikationsprogrammen initiiert. Die Roboterprogramme werden dabei üblicherweise interpretativ abgearbeitet, d.h. ein Interpreter setzt die einzelnen Bewegungsoperationen in den Programmen nacheinander in Roboterbewegungen um (siehe Abbildung 2.18) [FGL88, McK95].

In der Bahnplanung werden die einzelnen Stützpunkte der Trajektorie berechnet und an die Regelung der Roboterbewegung weitergegeben (Lageregelung). Die Antriebsverstärker sorgen dafür, daß die Antriebe die Bahnpunkte anfahren [Luh85].

gesteuert werden. Die Operationen zur Ansteuerung der Ein- und Ausgänge sind Booleschen (digitale) Verknüpfungen.

Bei der kontinuierlichen Bahnsteuerung müssen dazu die als kartesische Koordinaten berechneten Bahnstützpunkte durch eine Rückwärtstransformation in Achskoordinaten der einzelnen Gelenke umgerechnet werden.
Die aktuelle Position der Roboterarme wird durch Inkrementalwinkelgeber (Resolver) an der Welle der Roboterantriebe erfaßt. Für eine kontinuierliche Bahnsteuerung müssen nun diese Achskoordinaten wieder in Raumkoordination umgewandelt werden. Dies geschieht mit einer Vorwärtstransformation (VT, z.B. durch die Verfahren *Roll, Pitch and Yaw* oder Jacobi-Matrizen) [RNS93, McK95].

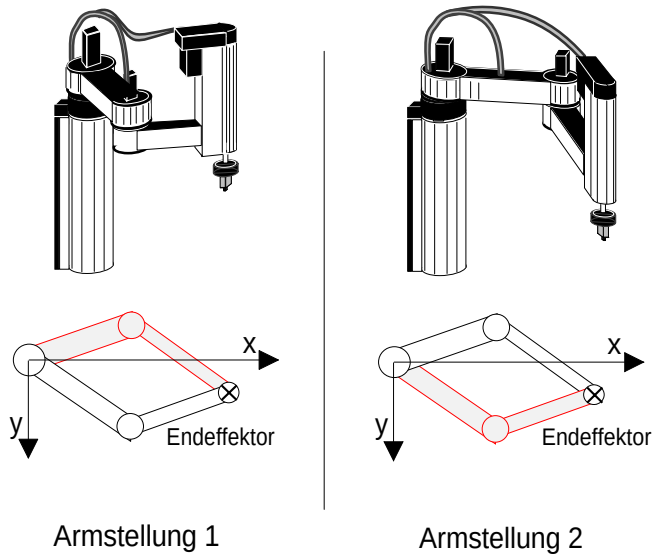


Abbildung 2.19: Mehrdeutigkeiten der Rücktransformation

Zur Umwandlung der Koordinaten von Raumkoordinaten in Achskoordinaten (RT) und umgekehrt (VT) sind unterschiedliche Verfahren entwickelt worden. Für einfache Kinematiken (z.B. SCARA oder Portalroboter, siehe Abschnitt 2.4.2.3) kann die RT leicht berechnet werden. Denavit-Hardenberg-Matrizen erleichtern die Berechnung der RT für komplexere Kinematiken wie Vertikalknickarme [RNS93, McK95].

2.4.2.2.3 Multitasking-Robotersteuerungen

Die meisten Robotersteuerungen bieten die Möglichkeit zur Multikasking-Abarbeitung von Anwendungsprogrammen. Allerdings ist die Anzahl der Tasks beschränkt (z.B. maximal fünf konkurrente Tasks auf der Robotersteuerung rho3 von Bosch). Diese Tasks werden nacheinander in einem festen zyklischen Ablauf bearbeitet (starre Echtzeitschleife). Dieser festgelegte Ablauf der Tasks entspricht jedoch nicht dem dynamischen Scheduling, wie es von POSIX konformen Betriebssystemen geboten wird.
Parallele Robotersteuerungen, die z.B. die im Betriebssystemstandard POSIX definierten Möglichkeiten nutzen oder sogar auf Mehrprozessorrechnern ablaufen, finden sich zur Zeit nur in der Forschung [CFAJ86, Ger90, NZ95, HH97, WHW98, KGSL97a].

2.4.2.2.4 Mehrdeutigkeiten der Rücktransformation

Je nach Roboterkinematik liefern die Algorithmen zur Rückwärtstransformation mehrere Lösungen für die Achsstellung, um eine bestimmte Zielposition mit dem Endeffektor zu

erreichen. Bei SCARA Kinematiken ergeben sich meist zwei mögliche Armstellungen (siehe Abbildung 2.19).
Diese Mehrdeutigkeiten müssen von dem Steuerungssystem durch spezielle Auswahlkriterien geklärt werden [Cra89, Wec96b]. Kriterien für die Auflösung sind:

- mechanische Begrenzung des Achsverfahrbereichs
- Kontinuität der Bahnbewegung
- kürzester Weg zum Zielpunkt

Darüberhinaus können schon im Anwenderprogramm Angaben gemacht werden, wie die Armstellung der Roboterkinematik zu sein hat.

2.4.2.3 Roboterkinematiken

Roboterarme bestehen aus beweglichen Gelenken (Joints). Die Zahl dieser Gelenke bestimmt die Anzahl der Freiheitsgrade der Roboterkinematik und damit die Beweglichkeit und den Arbeitsraum des Roboterarms.
Bei Industrierobotern finden sowohl rotatorische als auch translatorische Gelenke Anwendung [Jar84, SS85].

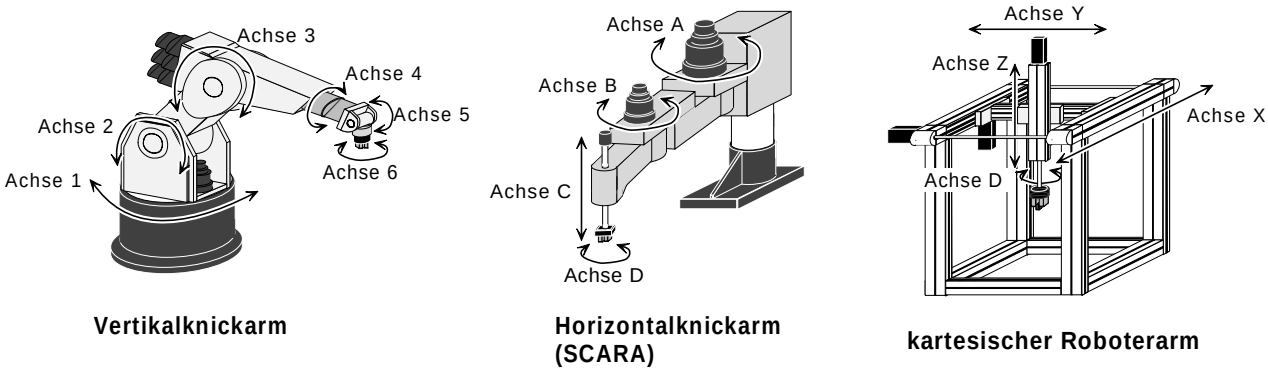


Abbildung 2.20: Vertikalknickarm, Horizontalknickarm und kartesischer Roboter

Die am häufigsten verwendeten Kinematikvarianten bei Industrierobotern sind [WSW85, Cra89, DH91, RNS93]:

- Vertikalknickarm
  Dieser Roboterarm besteht aus insgesamt sechs rotatorischen Achsen (drei Grundachsen und drei Hand- oder Nebenachsen). Damit kann jede Position im Arbeitsraum aus beliebiger Richtung angefahren werden. Jedoch ergibt sich dadurch das Problem, daß eine Position des Endeffektors im Raum durch viele unterschiedliche Armstellungen erreicht werden kann (siehe Abschnitt 2.4.2.2.4).

  Aufgrund der hohen Beweglichkeit wird dieser Robotertyp häufig zum Nachfahren komplexer Trajektorien, z.B. beim Bahnschweißen oder Lackieren eingesetzt.

  Die wichtigsten Hersteller dieser Kinematikvariante sind unter anderen die Firmen GM Fanuc, Kuka, ABB (ASEA-Brown-Boveri) und Reis.

- Horizontalknickarm (SCARA, Selective Compliance Assembly Robot Arm)
  Dieser Roboterarm wurde 1979 an der Yamanashi Universität in Japan entwickelt.

  Er besteht aus drei horizontalen Rotationsachsen und einer vertikalen translatorischen Achse. Dadurch ist die Beweglichkeit im Vergleich zum Vertikalknickarm eingeschränkt. Allerdings gibt es auch weniger mehrdeutige Ergebnisse der Rückwärtstransformation.

  Der Aufbau der SCARA Kinematik ist einfacher (kostengünstiger) als der der Vertikalknickarmroboter. Aufgrund der Anordnung der Achsen (die rotatorischen Achsen bewegen sich in horizontalen Ebenen) zeichnen sich Horizontalknickarme durch hohe Steifigkeit in vertikaler Richtung aus, die sich günstig auf die Positioniergenauigkeit auswirkt. Weitere Vorteile dieser Bauform sind hohe mögliche Verfahregeschwindigkeiten und Beschleunigungen sowie die kompakte Bauform.

  Aufgrund der Wiederholgenauigkeit werden SCARAs bevorzugt in der Kleinteilemontage eingesetzt, bei der Teile präzise gefügt werden müssen (z.B. Leiterplattenbestückung oder Fertigung von Autoradios).

  Gebaut werden Horizontalknickarme unter anderem von den Firmen Sony, Bosch und IBM.

- Kartesischer Roboter oder Portalroboter
  Kartesische Roboterarme bestehen aus drei translatorischen Achsen und einer rotatorischen Handachse. Die Steuerung dieser Systeme ist sehr einfach, da es keine uneindeutige Armstellungen gibt.

  Portalroboter besitzen wie Horizontalknickarme eine hohe Positioniergenauigkeit und werden daher meist für Palettier- und Montageaufgaben eingesetzt

  Portalsysteme werden von sehr vielen Firmen (unter anderen Bosch, Siemens, Festo, Mader, Isel) hergestellt.

Neben diesen drei Arten von Roboterarmen gibt es noch weitere Varianten und Sonderbauformen. Beispiele für solche Sondersysteme sind das Flugzeugreinigungssystem Skywash der Firma Putzmeister, mit dem die Außenhaut von Verkehrsflugzeugen gewaschen werden kann. Roboterarme können mit autonomen Systemen (z.B. mobilen Robotern) kombiniert werden. Dies ermöglicht unter anderem das Greifen von Gegenständen durch diese Mobilroboter. Beispiele für solche Systeme sind Serviceroboter wie ROMAN (TU München) [DEF+96] oder der Mobilroboter KAMRO (TU Karlsruhe) [RNS93].

Außerdem sind Kombinationen von stationären Roboterarmen möglich. Zur Fertigung von sehr großen Einheiten (z.B. Schiffssegmente oder Eisenbahnwaggons) werden an Portalsysteme Vertikalknickarmroboter angebracht. Das Portal positioniert den Vertikalknickarm, worauf dieser dann die eigentliche Aufgabe ausführt (z.B. Punkt- oder Bahnschweißen).

Neben miteinander verbundenen Roboterarmen gibt es Systeme, bei denen zwei getrennte Arme miteinander kooperieren [NSN95]. Der Vorteil dieser Anordnung ist, daß die Arme im Vergleich zu einzelnen Roboterarmen weitaus flexibler sind. Da sich allerdings die Arbeitsräume von beiden Armen überschneiden, müssen aufwendige Methoden zur Kollisionsvermeidung implementiert werden. Solche Systeme sind z.B. auf dem Mobilroboter KAMRO [Dam94] (s.o.) und im Weltraumlabor ROTEX [Hir93] eingesetzt worden.

2.4.2.4 Programmierung von Robotersteuerungen

Die Programmierung von Robotersystemen umfaßt räumliche und zeitliche Aussagen. Zum einen müssen die vom Roboterarm anzufahrenden Punkte im Raum (d.h. die einzelnen Be-

wegungen des Roboters) vorgegeben werden, zum anderen muß im Anwendungsprogramm die zeitliche und logische Abfolge der Bewegungen bestimmt werden.

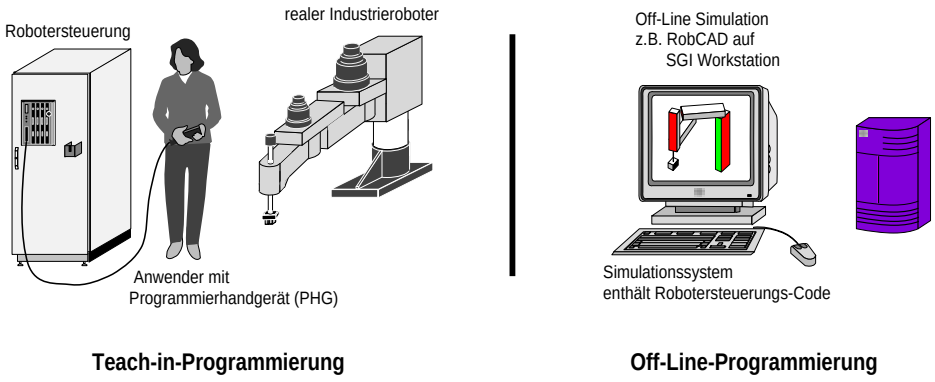


Abbildung 2.21: Teach-in-Programmierung und Off-Line-Programmierung

Zwar können in allen Roboterprogrammiersprachen Bahnpunkte von Roboterbewegungen als Absolutwerte in einem Applikationsprogramm angegeben werden, allerdings lassen sich die exakten Positionsdaten bei den meisten Anwendungen nur schwer vorherbestimmen. Soll ein Roboterarm auf einer Bezierkurve an einem Werkstück entlangfahren (z.B. beim Bahnschweißen einer Fahrzeugkarosserie), so können die einzelnen Zwischenpositionen dieser Bewegung nur sehr aufwendig ermittelt werden. Daher gibt es zwei Vorgehensweisen, die Positionen zu bestimmen und sie in einer Datenbank abzulegen (siehe Abbildung 2.21) [Cra89, RNS93, McK95]:

- Teach-in-Programmierung
  Der Benutzer bestimmt die Bewegungen des Roboters durch Handsteuerung mittels eines Programmierhandgeräts (PHGs). Die gewünschten Positionen des Roboterarms werden in einer Datenbank auf der Robotersteuerung abgelegt und können in das Anwendungsprogramm eingebunden werden.

  Dieses Verfahren wird heute überwiegend angewendet. Der Vorteil ist, daß der Anwender sofort die Bewegung der realen Kinematik überprüfen kann. Allerdings ist diese Methode sehr zeitaufwendig.

- Off-Line-Programmierung
  Bei dieser Vorgehensweise werden die Bewegungen Off-Line, d.h. auf einem speziellen Simulationssystem simuliert [AS94, Bic94b]. Manche Simulationen, wie z.B. RobCAD, enthalten Funktionen der realen Robotersteuerungs-Software.

  Die in der Simulation ermittelten Bewegungsinformationen werden auf eine Robotersteuerung übertragen und können dann nachgefahren werden. Da die reale Robotkinematik aufgrund mechanischer Toleranzen nicht exakt dem simulierten Modell entspricht, muß eine Anpassung der Positionsdaten (durch Kalibrierung des Roboterarms [SAL97]) erfolgen.

  Gegenüber der Teach-in-Programmierung können die Bewegungsdaten jedoch wesentlich leichter ermittelt werden, da die aufwendige Handsteuerung einer realen Kinematik entfällt. Die einzelnen Bewegungsabläufe können zum Teil anhand von CAD-Daten der zu bearbeitenden Werkstücke automatisch berechnet werden.

Neben diesen beiden Vorgehensweisen gibt es die Play-back-Programmierung (siehe Abschnitt 2.4.2.1, Play-back-Steuerung), bei der die Bewegung durch Handführung vorgegeben wird, und die sensorgestützte Programmierung. Hier bestimmt der Roboter seine Bewegung durch Abtasten eines Referenzstücks selbständig. Beide Programmiermethoden haben jedoch keine größere Verbreitung.

Die einzelnen Roboterbewegungsdaten der Positionsdatenbank werden durch Anwendungsprogramme in eine logische Abfolge gebracht. Die dazu verwendeten Roboterprogrammiersprachen können nach drei verschiedenen Ansätzen entwickelt werden [McK95]:

1. Modifikation einer konventionellen Computer-Programmiersprache

2. Modifikation einer NC-Programmiersprache

3. vollständige Neuentwicklung

Meist wird eine neue Sprache dann entworfen, wenn ein spezielles Robotersystem gebaut werden soll, das von den vorhandenen Sprachen nicht ausreichend unterstützt werden kann. Oftmals werden alle drei Ansätze bei der Entwicklung einer Roboterprogrammiersprache gemeinsam verwendet.
Die Mehrheit der kommerziellen Robotersteuerungssysteme bietet prozedurale Programmiersprachen [Hol85, BC85]. Diese basieren hauptsächlich auf BASIC, PASCAL oder C (zum Teil auch Assembler-Dialekte). Die Sprachen werden um Roboter-spezifische Befehle erweitert. Beispiele sind BAPS (Bosch Automatisierung Programmiersprache für Robotersteuerung rho3), SRCL (Siemens Robot Control Language, ACR) oder ARLA (ABB Robot Language, ASEA). Ein allgemeiner Standard für Roboterprogrammiersprachen (Industrial Robot Language, IRL) wurde 1993 als DIN 66312 Teil 1 [DIN95, Hei94] verabschiedet.
In der Forschung entstehen inzwischen die ersten Ansätze für objektorientierte Roboterprogrammiersprachen. Beispiele hierfür sind [Zie97] bzw. die Software-Bibliothek MRROC++ [Zie99b] und ZERO++ [PW97].

2.4.3 Speicherprogrammierbare Steuerungen

Speicherprogrammierbare Steuerungen werden hauptsächlich zur Automatisierung von industriellen Fertigungsprozessen eingesetzt. Sie haben weitestgehend die reinen verbindungsprogrammierten Steuerungen (VPS), die aus elektromechanischen Komponenten bestehen, verdrängt. Allerdings steuern speicherprogrammierbare Steuerungen ihre Peripherie weiterhin über elektrische Bauelemente wie Schütze und Koppelrelais (elektromechanische Schaltelemente) an [DIN85]. Ebenso müssen sicherheitsrelevante Schaltungen (z.B. Nothaltketten) verbindungsprogrammiert ausgeführt werden.
Die Hauptaufgaben der SPS sind die Steuerung und Überwachung von mechanischen Funktionseinheiten. Außerdem werden speicherprogrammierbare Steuerungen zum Datenaustausch mit anderen Systemen und zur Diagnose einer Anlage eingesetzt. Über Eingabeelemente (z.B. Schalter und Taster), Ausgabeelemente (z.B. Leuchtmelder) und spezielle Bedienterminals kommunizieren SPSen mit dem Anwender. [Pet89, Mic90, RNS93, Wec96a]
Wichtige Hersteller von speicherprogrammierbaren Steuerungen sind unter anderen die Firmen Mitsubishi, Siemens, Bosch, Klöckner-Möller und Eberle.
Im weiteren werden nun der Aufbau und die Programmierung von speicherprogrammierbaren Steuerungen vorgestellt.

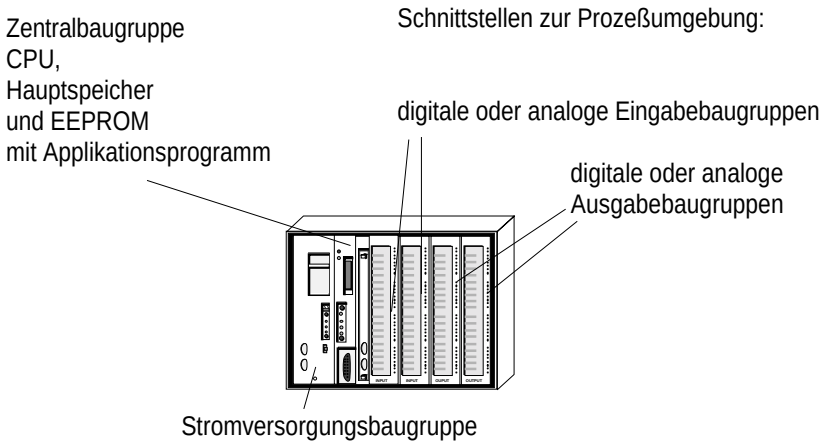


Abbildung 2.22: Bestandteile speicherprogrammierbarer Steuerungen, z.B. Siemens S5

2.4.3.1 Aufbau und Einsatz speicherprogrammierbarer Steuerungen

Speicherprogrammierbare Steuerungen sind speziell auf die Kommunikation mit einer großen Anzahl von Aktuatoren und Sensoren auf Prozeßebene ausgelegt. Daher besitzen sie viele digitale und analoge Ein- und Ausgänge. Im einzelnen besteht eine SPS aus folgenden Bestandteilen (siehe Abbildung 2.22) [Wec96a]:

- Die Zentralbaugruppe besteht aus der CPU, dem Hauptspeicher und dem wiederbeschreibbaren EEPROM, das das Applikationsprogramm enthält. Bei größeren Steuerungssystemen können meist weitere CPU-Baugruppen ergänzt werden.

- Die Schnittstellen zur Prozeßumgebung umfassen die digitalen und analogen Ein- und Ausgänge. Eine bestimmte Anzahl von Ein- und Ausgängen befinden sich auf einer Baugruppe.

- Die Stromversorgungsbaugruppe speist die Baugruppen der Steuerung. Die Ausgänge der Ausgangsbaugruppen werden jedoch nicht mit Spannung versorgt. Diese muß extern angelegt werden.

Speicherprogrammierbare Steuerungen werden zur Ansteuerung von elektrischen Bauteilen (Aktuatoren und Sensoren) eingesetzt (siehe Abbildung 2.23).
Digitale Aktuatoren sind beispielsweise ungeregelte Antriebe, Lampen oder Ventile. Analoge Aktuatoren sind z.B. geregelte Heizungen. Digitale Sensoren sind unter anderem Schalter und Taster. Analoge Sensoren sind beispielsweise Temperatursensoren.
Speicherprogrammierbare Steuerungen werden meist über elektromagnetische Schalter (Schütze und Relais) mit den Aktuatoren verbunden. Dadurch wird die SPS von den zum Teil hohen Strömen der Aktuatoren galvanisch entkoppelt. Sensoren werden meist direkt mit der Steuerung verbunden.

2.4.3.2 Abarbeitung von SPS-Programmen

Die Abarbeitung der Applikationsprogramme geschieht zyklisch (moderne Steuerungen bieten zusätzlich noch eine Interrupt-gesteuerte Bearbeitung). Dadurch wird das gesamte Anwendungsprogramm, unabhängig davon ob sich die Signalzustände am Eingang der SPS

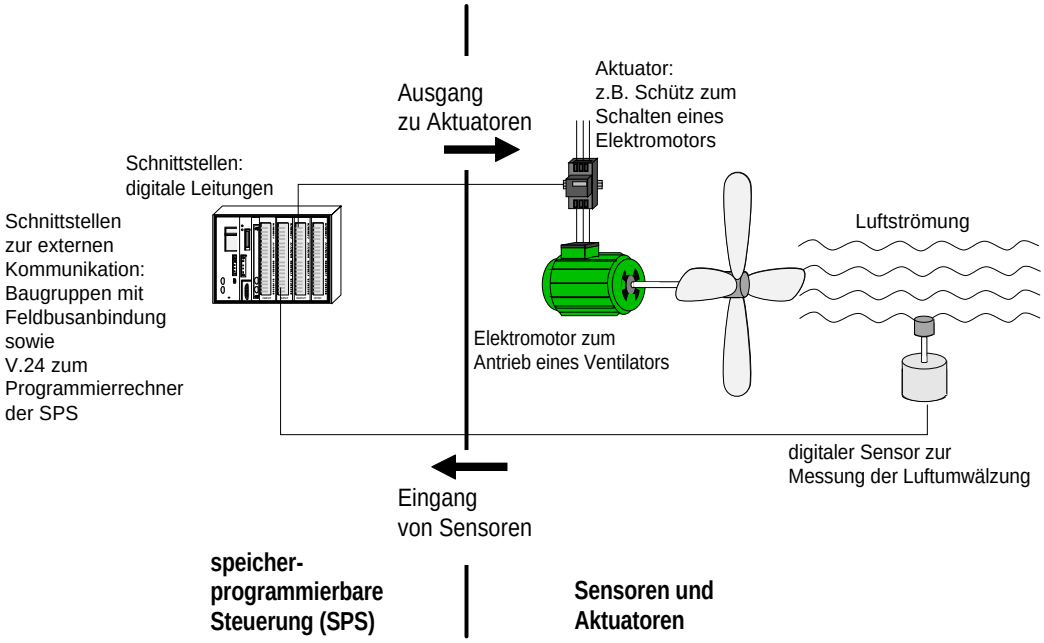


Abbildung 2.23: Anwendung speicherprogrammierbarer Steuerungen (Beispiel)

geändert haben, permanent bearbeitet. Die Programme können in Blöcke (d.h. Module) gegliedert werden. Ebenso kann durch Sprunganweisungen der Programmfluß verändert werden.
Eine genauere Beschreibung der Programmbearbeitung findet sich in Abbildung 2.24:

1. Zu Beginn eines Zyklus wird das Prozeßabbild von den Eingängen in einen Eingangsspeicher eingelesen.

2. Darauf werden die Anweisungen des Applikationsprogramms nacheinander abgearbeitet. Diese Anweisungen sind logische Verknüpfungen von Eingangswerten aus dem Eingangsspeicher und gespeicherten Werten (Merker). Das Ergebnis der Operationen wird in den Ausgangsspeicher geschrieben.

3. Zum Abschluß eines Programmdurchlaufs werden die Werte des Ausgangsspeichers an die Ausgangsbaugruppen weitergegeben.

Ein Zyklus dauert je nach Programmlänge und Leistungsfähigkeit des Geräts wenige Millisekunden (Zykluszeit) [Wec96a]. Durch die in Abhängigkeit zur Programmlänge veränderliche Zykluszeit unterscheidet sich eine SPS von einer Robotersteuerung, bei der die Lageregelung (und alle unterlagerten Regelschleifen) mit fester Zykluszeit erfolgt.

SPS-Programme setzen sich hauptsächlich aus logischen Verknüpfungen (`UND`, `ODER`, `INVERTER`, usw.) zusammen. Zusätzlich können Zeitbedingungen und Sprunganweisungen programmiert werden. In den konventionellen SPS-Programmierverfahren kommen drei verschiedene, gleichwertige Möglichkeiten zur Darstellung von Programmen zum Einsatz (siehe Tabelle 2.6) [Bli88, Wec96a] (Die Beispiele in der Tabelle stellen jeweils eine einzelne Anweisung dar.):

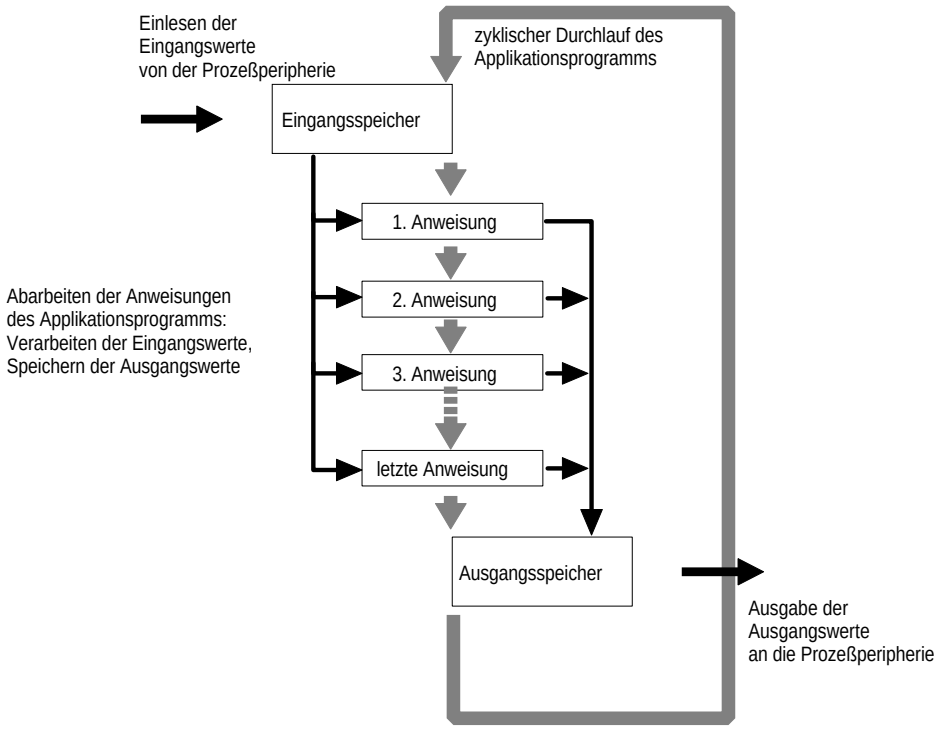


Abbildung 2.24: Zyklische Abarbeitung von SPS-Programmen

- Kontaktplan (KOP)
  Diese Programmdarstellung kommt besonders Anwendern entgegen, die bisher Steuerungen in Relaistechnik (VPS) entwickelt haben. Im Kontaktplan und im Relaisstromlaufplan werden ähnliche Symbole verwendet. Ein Stromlaufplan kann meist direkt in ein entsprechendes SPS-Programm umgewandelt werden.

- Funktionsplan (FUP)
  Ein als Funktionsplan geschriebenes SPS-Programm entspricht einem Schaltplan mit Logikbausteinen.

- Anweisungsliste (AWL)
  Diese Darstellung läßt sich direkt aus den die Steuerungsaufgabe beschreibenden Booleschen Gleichungen ableiten. Dabei gelten die in der Booleschen Algebra verwendeten Regeln (z.B. `UND` vor `ODER`).

In den Programmdarstellungen werden die Eingangswerte als `E` mit einer Nummer (z.B. `E1`) und die Ausgangswerte als `A` mit Nummer (z.B. `A1`) bezeichnet. Zusätzlich gibt es Speicherelemente (Merker, z.B. `M1`).

Seit 1992 liegt die internationale SPS-Programmiernorm IEC 1131 Teil 3 vor [IEC92]. Diese Norm bietet zusätzliche Programmdarstellungen, die ebenso ineinander überführt werden können.

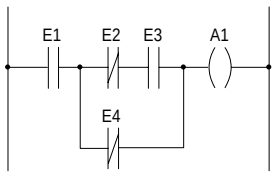
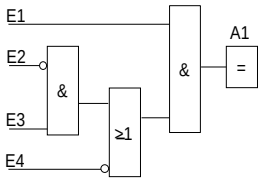
Kontaktplan (KOP)	Funktionsplan (FUP)	Anweisungsliste (AWL)
		U E1 U (U N E2 U E3 O N E4) = A1
graphische Programmierung	graphische Programmierung	mnemotechnische Programmierung

Tabelle 2.6: Konventionelle Programmierverfahren für SPS

2.4.4 Numerische Steuerungen

Numerische Steuerungen (NC, Numerical Control) [8] werden hauptsächlich zur Steuerung von Werkzeugautomaten (z.B. Dreh-, Fräs-, Schleifmaschinen, Schneideanlagen, usw.) eingesetzt [GH70, RNS93, Wec96a].

NCs steuern ebenso wie Robotersteuerungen die Achsen des jeweiligen Geräts. Bei beiden Systemen können Trajektorien nachgefahren werden. Somit besteht eine gewisse Übereinstimmung in den Funktionen zwischen beiden Steuerungstypen [Kor85]. Im Gegensatz zu Robotern bestehen NC-Maschinen jedoch aus einzelnen Achsen, die keine geschlossene Kinematik bilden.

Die weltweit bedeutendsten Hersteller von numerischen Steuerungen sind die Firmen Fanuc, Mitsubishi und Siemens.

Die folgenden Abschnitte geben einen Einblick in die Hardware und den Aufbau numerischer Steuerungen.

2.4.4.1 Hardware

Die numerischen Steuerungen werden über Flachbedientafeln mit einer Tastatur und einem LCD-Bildschirm oder über eine Maschinensteuertafel bedient. Zusätzlich können einzelne Achsen durch ein elektronisches Handrad manuell verfahren werden. Diese NC-Steuerelemente ersetzen die mechanischen Bedienelemente der herkömmlichen Maschinen.

NC-Systeme sind modular aufgebaut (Kartensysteme). Sie bestehen aus einer Master-CPU, die um weitere CPU-Module (z.B. zur Steuerung zusätzlicher Achsen) erweitert werden können. Darüberhinaus können Kommunikationsmodule und sogar SPS-Module als Steckkarten in das Gehäuse des Kartensystems (Rack) eingefügt werden.

Zur Kommunikation nach außen haben numerische Steuerungen V.24-Schnittstellen und Feldbusanbindungen.

[8] Die ersten NC-Systeme waren festverdrahtet. 1972 wurden die ersten Steuerungen mit einem oder mehreren Rechnern eingeführt. Zur Abgrenzung wurden diese Computer-basierten Steuerungen CNC (Computerized Numerical Control) genannt. Da seit Ende der 70er Jahre praktisch alle NCs Mikroprozessoren enthalten, wird inzwischen wieder einheitlich der Begriff NC verwendet.

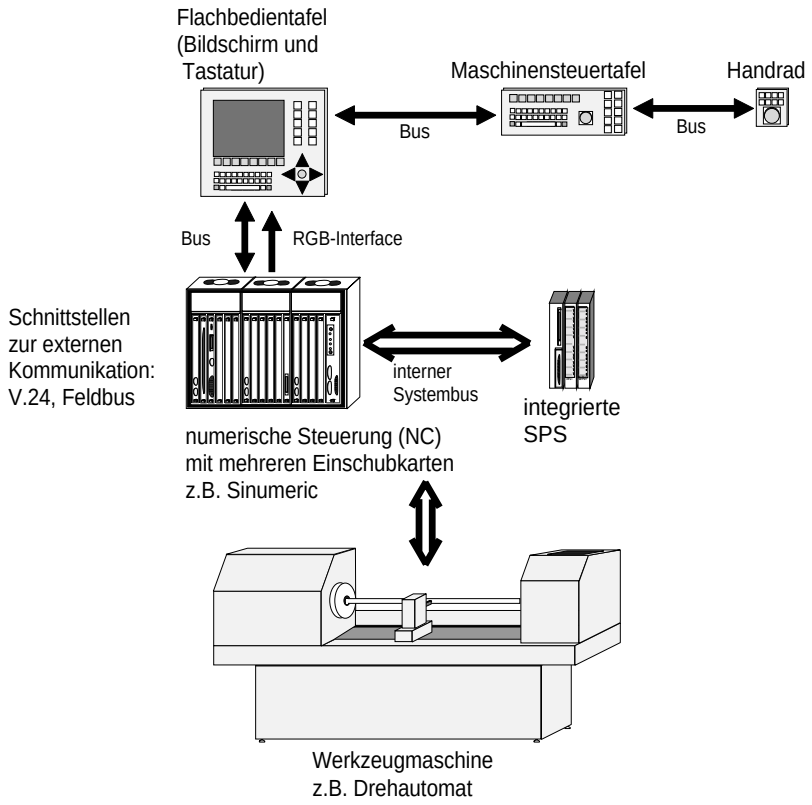


Abbildung 2.25: Numerische Steuerung für Werkzeugmaschinen [Wec96a]

2.4.4.2 Aufbau numerischer Steuerungen

Bei numerischen Steuerungen erfolgt die Steuerung der Maschine durch Tabellen von Zahlen anstelle herkömmlicher Programme. Alle Anweisungen (Geometrie- und Technologiedaten) zur Fertigstellung eines Werkstücks werden numerisch codiert [DIN83]. Der Datensatz, d.h. das NC-Programm, ermöglicht eine exakte Wiederholung von Relativbewegungen zwischen Werkzeug und Werkstück ohne Abweichungen.
Ähnlich den Robotersteuerungen ist der logische Aufbau von numerischen Steuerungen hierarchisch (siehe Abbildung 2.26) [RNS93, Wec96a]:
Auf der obersten Ebene steht die NC-Programmierung durch den Benutzer. Die Programmierung geschieht entweder durch Eingabe von Daten (dies kann lokal oder entfernt geschehen) oder durch direkte Bedienung (z.B. Handfahren).
Über die unterlagerte Satzaufbereitungsebene hat der Benutzer Zugriff auf die aktuellen Daten der Steuerung. Darüberhinaus werden in dieser Ebene die Befehle und Programme entgegengenommen und interpretiert.
Zur weiteren Verarbeitung werden die Daten des NC-Programms abgelegt. Dabei wird zwischen Geometriedaten (Bewegung, bzw. Bahnen, der Maschinenachsen entsprechend der Geometrie des zu bearbeitenden Werkstücks) und Technologiedaten (Informationen über den Bearbeitungsprozeß, wie z.B. Spindeldrehzahl oder Vorschub) sowie Schaltfunktionen unterschieden. Die Schaltfunktionen beziehen sich auf die Ansteuerung der Maschinenstellglieder (z.B. Kühlmittelpumpe), die Verarbeitung von Sensordaten (z.B. Temperaturüberwachung des Werkzeugs) und die Sicherheitsschaltungen (z.B. Notaus). Zur Bearbeitung der Schaltfunktionen wird meist eine SPS als externer Rechner zusätzlich zur NC herangezogen. Diese SPS wird durch einen internen Systembus mit der Master-CPU verbunden.
In der Verarbeitungsebene werden die Geometrie- und Technologiedaten bearbeitet. Die

Interpolation berechnet die Bewegungen der einzelnen Achsen. Wie bei Robotersteuerungen ist der Interpolation eine Lageregelung unterlagert. Über diese Lageregelung werden die einzelnen Feinregelungen der Achsen angesteuert.

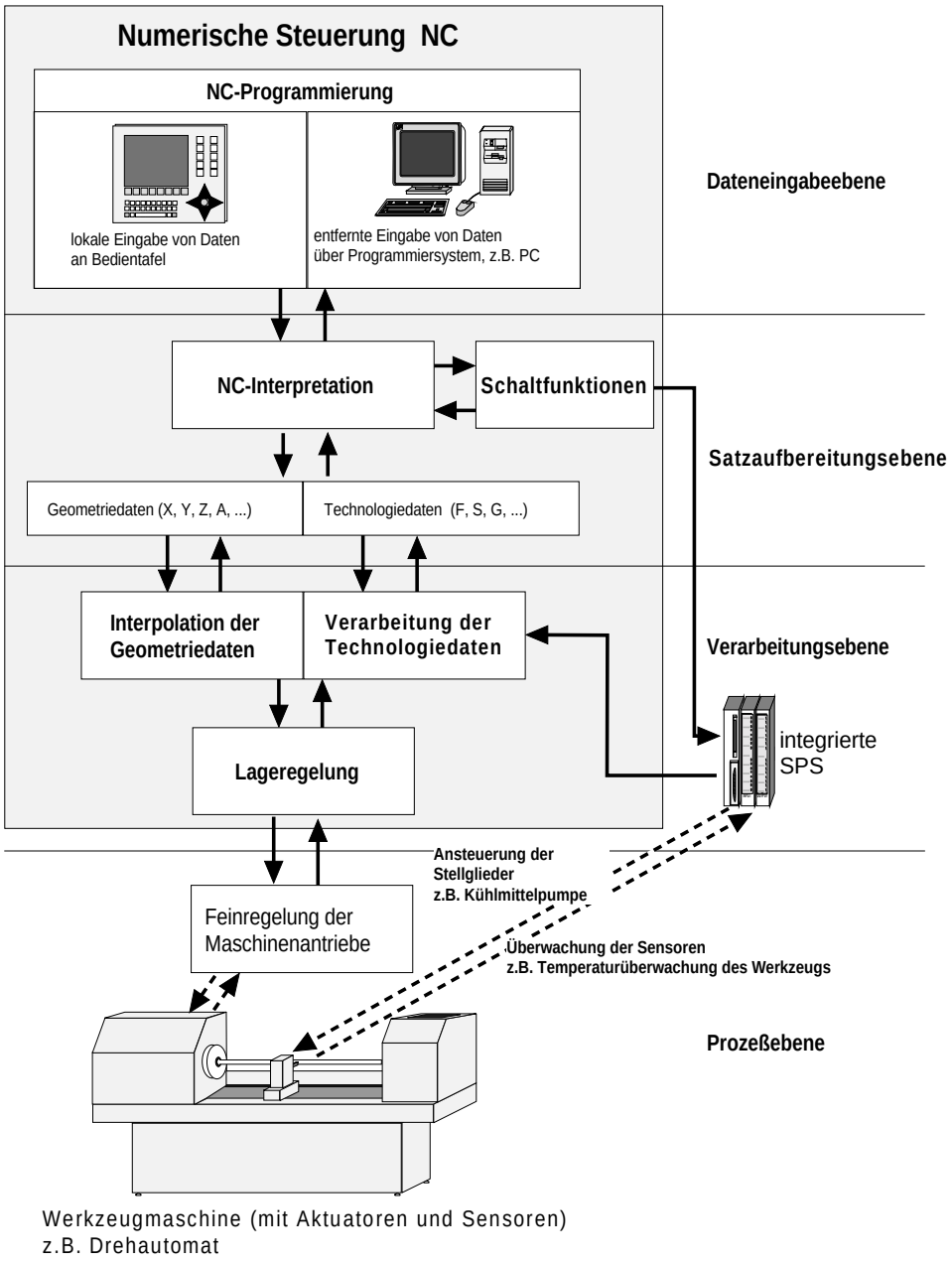


Abbildung 2.26: Aufbau numerischer Steuerungen (NC)

In der Prozeßebene befinden sich die Aktuatoren, wie z.B. die Antriebe und Antriebsregler sowie die Sensoren der Werkzeugautomaten.

Die möglichen Interpolationsarten sind denen der Robotersteuerungen ähnlich (siehe Abschnitt 2.4.2.1) [RNS93, Wec96a]:

- Punktsteuerungen werden für einfache Positioniervorgänge verwendet. Dabei wird nur eine Achse verfahren. Das Werkzeug darf bei dieser Interpolationsvariante nicht im Eingriff sein (d.h. das Werkstück nicht bearbeiten).

- Die Streckensteuerung ermöglicht, einen Endpunkt von einer oder mehreren Achsen auf geradem Weg anzufahren. Bei dieser Interpolationsart können einfache Fräs- oder Dreharbeiten ausgeführt werden.

- Bahnsteuerungen bieten die Möglichkeit, Bahnkurven wie Geraden oder Kreise, mit definierter Geschwindigkeit zu durchfahren. Die meisten Werkzeugmaschinentypen nutzen diese Interpolation zur Bearbeitung von Werkstücken.

2.4.5 Offene Steuerungssysteme

Die marktgängigen industriellen Steuerungssysteme haben den Nachteil, daß sie proprietäre Entwicklungen sind. Komponenten unterschiedlicher Hersteller können nur selten untereinander ausgetauscht werden.
Offene Steuerungssysteme mit standardisierten Schnittstellen und offenen Architekturen sollen daher den freien Austausch von Komponenten ermöglichen. Dadurch soll die Entwicklung von Steuerungssystemen effizienter gestaltet werden (d.h. Reduktion der Kosten und Entwicklungszeit).
Im Gegensatz zu den bisherigen proprietären Steuerungen sollte ein offenes Steuerungssystem also auf unterschiedlichen Plattformen ablaufen und dabei um beliebige Steuerungskomponenten (z.B. Robotersteuerung, SPS oder numerische Steuerung) erweitert werden können.

Eine komplett plattformunabhängige Steuerung kann nur schwer entwickelt werden. Allerdings erleichtert die Verwendung von Standard-Hardware [9] mit POSIX konformen Betriebssystemen die Portierung von Steuerungssystemen. Daher basieren offene Systeme in den meisten Fällen auf Standardrechnern.
Weitere Steuerungskomponenten sind am einfachsten dadurch in ein System einzubinden, daß die Architektur offengelegt wird und entsprechende Schnittstellen für die Erweiterung vorgesehen sind. In diesem Sinn können reine Referenzarchitekturen als Muster (siehe Abschnitt 2.1) und bereits bestehende erweiterbare Steuerungen als Frameworks (siehe Abschnitt 2.2) betrachtet werden.
Im folgenden werden einige beispielhafte Projekte zur Entwicklung offener Steuerungssysteme vorgestellt. Anschließend folgt eine Beschreibung von aktuell eingesetzten Standardrechnern als Steuerungs-Hardware.

2.4.5.1 Projekte zur Entwicklung offener Steuerungssysteme

Im Laufe der letzten zehn Jahre sind eine Reihe von Forschungsprojekten zur Entwicklung von Referenzarchitekturen für offene Systeme durchgeführt worden. Einige Beispiele für derartige Projekte sind:

- OSACA (Open System Architecture for Controls within Automation Systems)
  Dieses Verbundprojekt (ESPRIT Projekt) von verschiedenen europäischen Forschungseinrichtungen sowie Steuerungsherstellern und -nutzern hatte eine Laufzeit von 1992 bis 1996 [OSA96].

  In OSACA ist eine Spezifikation und Referenzarchitektur erstellt worden. Das Steuerungssystem soll portabel, erweiterbar (bzw. skalierbar), austauschbar und interoperabel mit anderen Systemen sein. Die Architektur ist hierarchisch in Ebenen gegliedert

[9] Standard-Hardware soll in dieser Arbeit als in großen Mengen hergestellte Rechnersysteme (off-the-shelf Systeme) definiert werden. Beispiele dafür sind PCs und Workstations.

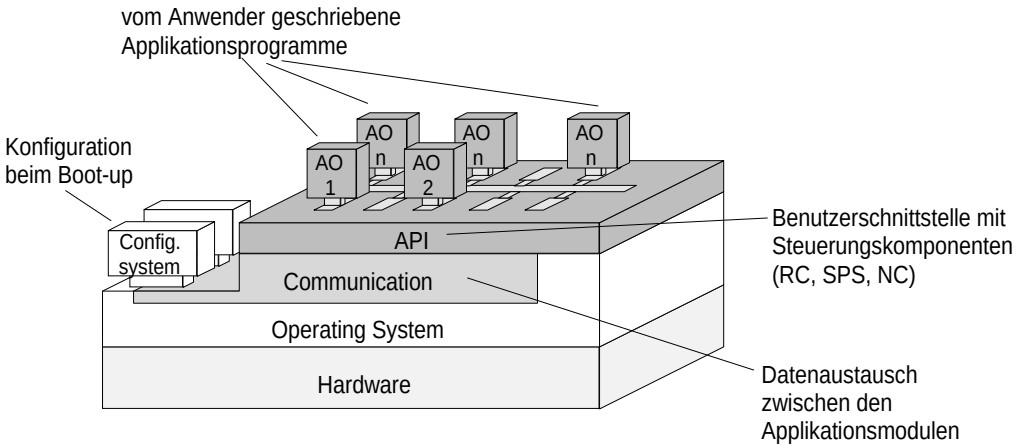


Abbildung 2.27: Referenzarchitektur von OSACA

(siehe Abbildung 2.27):
Hardware, Operating System, Communication, Application Programming Interface (API) und Applikationsprogramme.
Diese Ebenen bestehen jeweils aus verschiedenen Modulen, die innerhalb ihrer Ebene ausgetauscht werden können. Die Steuerungskomponenten (Module der Applikationsprogrammierungsebene, API) von OSACA sind Robotersteuerung (RC), SPS und numerische Steuerung (NC).

Im Rahmen von OSACA ist am ISW (Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen, Universität Stuttgart) u. a. ein offenes Konzept für eine Multitasking-Robotersteuerung (ISWRC) entwickelt worden [PUD94].

- OMAC (Open Modular Architecture Controllers)
  1994 haben die amerikanischen Automobilhersteller General Motors, Ford und Chrysler eine Spezifikation für eine offene und modulare Steuerungsarchitektur begonnen [Chr94]. Die Funktionalität von OMAC umfaßt im wesentlichen SPS und NC.

  Ergebnis des Projekts ist neben der Spezifikation eine 1998 veröffentlichte Applikationsprogrammierungsschnittstelle (API) [OMA98].

- CEROS (Cost Efficient Robot Control System)
  In diesem ESPRIT Projekt von europäischen Roboterherstellern, Steuerungsherstellern und Forschungseinrichtungen wurde eine Spezifikation und Referenzarchitektur für eine offene Robotersteuerung erstellt. Die Basis hierfür ist eine numerischen Steuerung [BSW96].

Neben diesen Arbeiten an offenen Architekturen gibt es Konzepte, nach denen unterschiedliche Steuerungskomponenten auf proprietären Plattformen modular ablaufen. Ein bekannter Ansatz ist die Open Control Architektur für die proprietäre Zielplattform Windows NT [Ope98]. Allerdings ist fraglich inwieweit PCs mit Windows NT tatsächlich als Steuerungsrechner eingesetzt werden können. Eigene Messungen haben beispielsweise ergeben, daß auf einem PC (Intel Pentium 166 MHz und 32 MB Hauptspeicher) mit Windows NT bei einer Schleife mit einer Zykluszeit von 20 ms bis zu 25 ms Abweichungen (Jitter) auftreten können (siehe Abschnitt 6.3).

2.4.5.2 Standard-Hardware als industrielle Steuerungsrechner

Die industriellen Varianten der Standardrechner (z.B. Industrie-PC oder Industrie-Workstation) unterscheiden sich von denen für Bürosysteme hauptsächlich dadurch, daß sie unter wesentlich ungünstigeren Umweltbedingungen (z.B. Temperatur und Verschmutzung) noch zuverlässig arbeiten und sie mehr Schnittstellen für die Kommunikation mit peripheren Geräten unterstützen.
Meist werden Industrierechner als Einschubkarten für Kartensysteme realisiert (z.B. Industrie-PC SMP 16 von Siemens oder industrielle SPARC Einschubkarten von den Firmen FORCE oder MEN als Sun Clone Systeme).
Das Anwendungsspektrum von industriellen Standardrechnern ist sehr weit gefächert. Beispiele für charakteristische Anwendungsfälle sind:

- Spezialsteuerungen
  Die Anforderungen an diese Spezialsteuerungen übersteigen die Möglichkeiten von konventionellen Steuerungssystemen (RC, SPS und NC). Daher übernehmen leistungsfähige und flexible Standardrechner die Steuerungsaufgaben.

  Ein Beispiel aus dem wissenschaftlichen Umfeld ist die Steuerung des mobilen Roboters AMBLER (Autonomous MoBiLE Robot) [BHK+89]. Dieses Steuerungssystem besteht aus einer Workstation von Sun Microsystems. AMBLER ist ein autonomer Laufroboter, der für die Erkundung von fremden Planeten entwickelt worden ist.

  Zur Steuerung einer Fertigungszelle im industriellen Einsatz verwendet die Fa. Seagate beispielsweise ein PC-basiertes System [SC95]. In dieser Zelle werden Festplatten hergestellt. Das flexible Multitasking-System steuert ein Dreiachseinlegegerät, den Materialfluß, die Montage und beinhaltet zusätzlich Diagnose- und Testfunktionen.

  Insgesamt haben diese Spezialsteuerungen den Nachteil, daß sie proprietäre Lösungen sind und nicht universell eingesetzt werden können.

- Überlagerte Steuerungssysteme für verschiedene konventionelle Steuerungen
  Hier werden Standardsysteme nur zur Produktionsplanung und -überwachung eingesetzt. Sie koordinieren lediglich andere Steuerungen.

  In [VHK+92] wird ein derartiges System, bei dem PCs Automatisierungsgeräte überwachen, beschrieben. Allerdings ersetzen die PCs die "klassischen" proprietären Steuerungssysteme wie Robotersteuerungen und speicherprogrammierbare Steuerungen nicht.

- Neue Steuerungssysteme
  Seit 1995 bietet die Firma Kuka die PC-basierte Robotersteuerung KR C1 an. Diese kann mehrere Typen von Roboterkinematiken (meist unterschiedliche Größen von Vertikalknickarmrobotern) steuern. Allerdings ist die KR C1 keine universelle Steuerung. Sie kann nur in eingeschränktem Umfang andere Steuerungsaufgaben (wie Operationen von speicherprogrammierbaren Steuerungen) übernehmen.

  Der Geschäftsbereich Automation & Drives der Siemens AG hat eine Modellreihe von unterschiedlich leistungsfähigen Industrie-PCs (Sicomp) entwickelt. Je nach Ausbaustufe der Rechner können diese konkurrent numerische Steuerungsapplikationen und Anwendungen für speicherprogrammierbare Steuerungen abarbeiten. Eine Robotersteuerung gibt es für diese PCs allerdings nicht.
  Hauptsächlich werden die Sicomp PCs für Meß- und Regelungsaufgaben verwendet.

Moderne Standard-Hardware-basierte Steuerungssysteme eröffnen neue Einsatzgebiete. So können diese Steuerungen ebenso einfach wie Bürosysteme vernetzt werden. Dadurch erschließen sich Einsatzfelder wie Teleworking, Fernwirken und Fernwartung [STY94, KGL+97].

2.4.6 Industrielle Fertigungssysteme

Industrielle Fertigungssysteme bestehen meist aus einer Vielzahl kooperierender Geräte (Steuerungen und Maschinen).
Im folgenden sollen nun die Fertigungssysteme in der Produktion und die Kommunikation zwischen einzelnen Steuerungssystemen vorgestellt werden.

2.4.6.1 Typische Fertigungssysteme (Fertigungszellen)

In automatischen Fertigungszellen arbeiten üblicherweise unterschiedliche Arten von Geräten mit den entsprechenden Steuerungen (RC, SPS und NC) zusammen.
Am Beispiel von roboterbasierten Fertigungssystemen werden charakteristische Konfigurationsvarianten vorgestellt. In den beschriebenen Systemen arbeiten ein oder mehrere Roboterarme mit mehreren peripheren Geräten zusammen [KPK95].

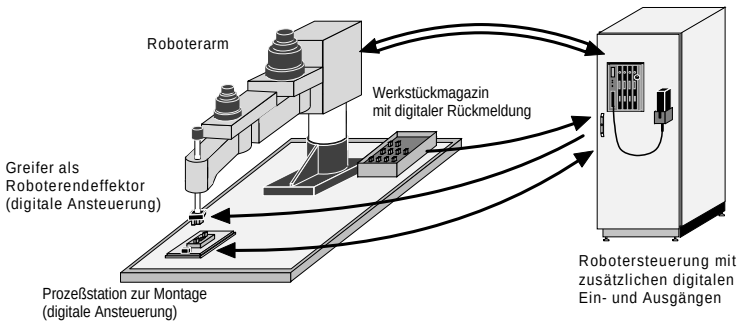


Abbildung 2.28: Robotersteuerung als alleinige Steuerung

- Robotersteuerung als alleiniges Steuerungssystem
  Die Robotersteuerung steuert sowohl den Roboterarm als auch wenige weitere Geräte. In den meisten Robotersteuerungen können einige digitale und analoge Ein- und Ausgangsbaugruppen eingebaut sein. Über diese Baugruppen werden neben einem einfachen Endeffektor des Roboterarms (z.B. Greifer oder Sauger) auch einige wenige periphere Geräte (z.B. Prozeßstation oder Werkstückmagazin) angesteuert.

  Durch diese Konfiguration können kleinere Anlagen oder Roboterzellen gesteuert werden. Die Kommunikation nach außen geschieht über die Robotersteuerung.

- Robotersteuerung mit zusätzlichen Steuerungssystemen
  Für viele Anwendungen reichen die Möglichkeiten (Leistungsfähigkeit) der Robotersteuerung zur Kontrolle der peripheren Geräte nicht aus. Daher werden zusätzliche Steuerungen (meist speicherprogrammierbare Steuerungen) verwendet. Deren typische Aufgaben sind die Steuerung von Transportsystemen (für den Materialfluß innerhalb der Zelle) oder komplexe Roboterendeffektoren (wie z.B. Schweißzangen, die spezielle Schweißsteuerungen benötigen).

  Die Robotersteuerung koordiniert die Arbeiten in der Fertigungszelle und kommuniziert mit Systemen außerhalb der Zelle. Die Kommunikation zwischen der Robotersteuerung und der Zusatzsteuerung erfolgt über digitale Signale, serielle Schnittstellen (V.24 oder 20mA Schnittstelle) oder Feldbusse.

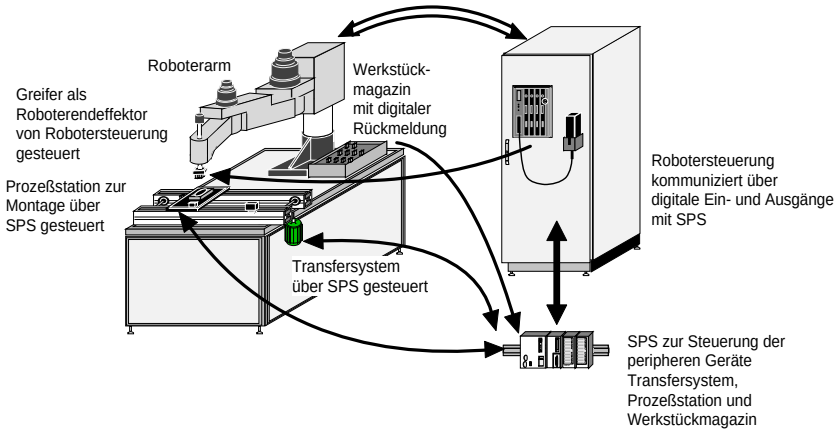


Abbildung 2.29: Robotersteuerung und SPS

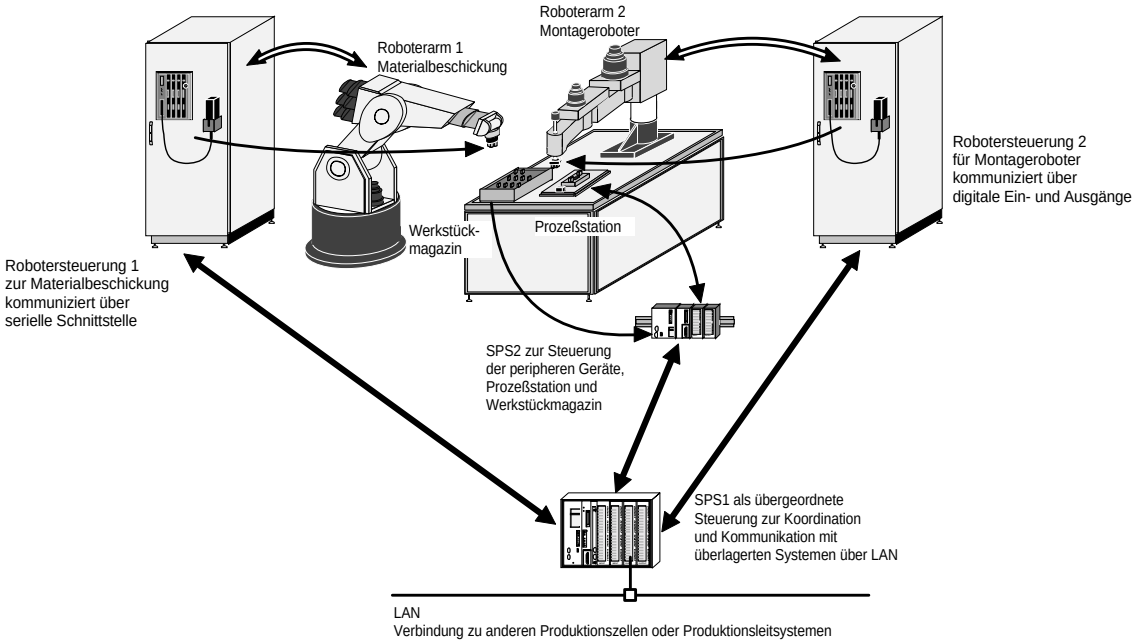


Abbildung 2.30: Komplexes Fertigungssystem mit mehreren Steuerungen

- Kombination mehrerer Steuerungstypen
  Sehr komplexe Montagezellen oder Fertigungsstraßen benötigen mehrere Steuerungen (zum Teil sogar mehrere Robotersteuerungen [FB93]). Die meisten dieser Systeme verwenden eine Steuerung (meist eine SPS) als überlagerte Steuerung (Leitrechner) [RK92, Bra96]. In diesem hierarchischen Steuerungsaufbau [Cas83] koordiniert ein (oder mehrere) Leitrechner die verschiedenen Steuerungssysteme und kommuniziert mit anderen Fertigungseinheiten oder den Fertigungssystemen überlagerten Produktionsplanungssystemen.

  Ein Beispiel für eine besonders aufwendige Kombination von unterschiedlichen industriellen Steuerungssystemen in einer Fertigungszelle findet sich in [RK92]. Die Aufgabe dieser Produktionszelle besteht aus der Beschichtung von Windschutzscheiben für Fahrzeuge. Der Fertigungsprozeß besteht aus mehreren Schritten, wie z.B. Reinigen der Scheiben und Beschichten des Glas.
  Insgesamt werden in diesem System zwei speicherprogrammierbare Steuerungen (eine

Prozeßsteuerung und eine überlagerte Steuerung zur Gesamtkoordination), drei Robotersteuerungen und eine Förderbandsteuerung benötigt. Die Prozeßvisualisierung läuft auf einem separaten PC ab. Zur Kommunikation der Steuerungen untereinander werden serielle Verbindungen (V.24, bzw. RS323), digitale Einzelverbindungen sowie analoge Schnittstellen verwendet. Nach außen kommuniziert die Produktionszelle mit einem Produktionsplanungssystem, von dem sie Arbeitsaufträge erhält (DNC, Direct Numerical Control).

Neben diesen Beispielen gibt es auch beliebige Kombinationen von numerischen und speicherprogrammierbaren Steuerungen (wie z.B. in flexiblen Bearbeitungszentren [RNS93]).

2.4.6.2 Feldbusse zur Kommunikation

In diesem Abschnitt werden Feldbusse in die Kommunikationshierarchie von Unternehmen eingeordnet. Daran schließt sich der Vergleich von Feldbussen mit dem OSI-Referenzmodell, eine Beschreibung eines Feldbuskommunikationsmodells am Beispiel des CANopen Protokolls und ein Überblick über charakteristische Anwendungen dieser Bussysteme an.

2.4.6.2.1 Einsatz von Feldbussen in Unternehmen

Die Kommunikationssysteme innerhalb von Fertigungsbetrieben sind hierarchisch aufgebaut. Die Anordnung orientiert sich an der Hierarchie der Rechnersysteme in Firmen (siehe Abschnitt 2.4.1.3).
Feldbusse sind in der unteren Ebene der Kommunikationshierarchie eingeordnet (siehe Abbildung 2.31). Sie ermöglichen die Kommunikation zwischen Steuerungen der Zellenebene und den Aktuatoren und Sensoren der Geräteebene. Dabei ersetzen diese Busse die Einzelverdrahtung. Außerdem können Zellenrechner untereinander über Feldbusse Daten austauschen.

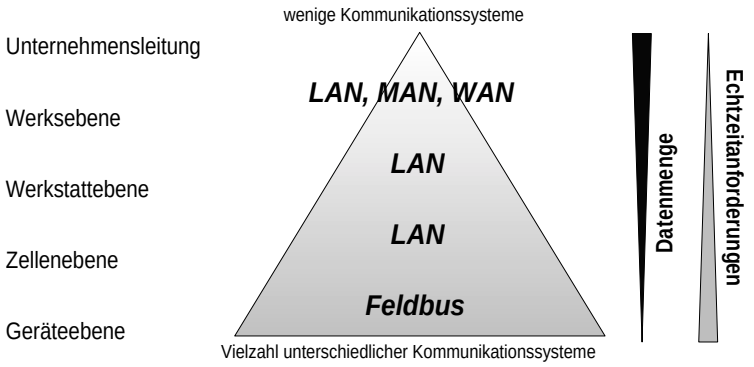


Abbildung 2.31: Kommunikationshierarchie von Unternehmen

Durch den Einsatz von Feldbussen in der prozeßnahen Kommunikation der Fertigungs- und Verfahrenstechnik ergeben sich folgende Anforderungen an Feldbusse [RNS93]:

- Echtzeitverhalten, Antwortzeit von maximal 1 ... 50 ms
- hohe Übertragungssicherheit
- hohe Verfügbarkeit
- kurze Nachrichtenlänge, günstiges Brutto- / Netto-Datenverhältnis

- geringe Kosten
- einfache Protokolle (kurzer Protokoll-Stack)
- große Anzahl von Teilnehmern
- galvanische Entkopplung der Busknoten
- geringe Busausdehnung (10 m ... 1000 m)

Die Kommunikationssysteme auf den einzelnen Ebenen werden in Abbildung 2.31 dargestellt. Für die Rechnerkommunikation in Unternehmen gilt, daß je niedriger die Ebene:

- desto mehr Geräte kommunizieren
- desto geringer ist die ausgetauschte Datenmenge pro Übertragung
- desto zeitkritischer ist die Kommunikation.

2.4.6.2.2 Einordnung in das OSI-Referenzmodell

Das OSI-Referenzmodell (Open System Interconnection) der ISO [Tan89, Tan92] teilt die einzelnen Funktionen der digitalen Datenübertragung in sieben Schichten ein. Allerdings ist es nicht unbedingt erforderlich, daß ein Kommunikationssystem alle Ebene verwendet. Je mehr Schichten realisiert werden, desto komplexer wird das System und desto schlechter wird das Zeitverhalten.
Bei den meisten Feldbussen (z.B. Profibus, Interbus-S [**?**] oder CAN) sind nur die Schichten 1 und 2 sowie die Applikationsschicht (Schicht 7) implementiert [RNS93].

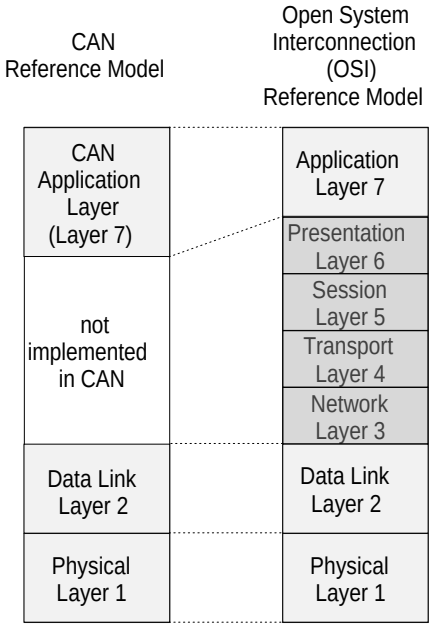


Abbildung 2.32: Schichten des Feldbus CAN [CAL94]

Am Beispiel des Feldbus CAN (Controller Area Network) [CAL94, CAN95a] soll der Aufbau von Feldbussystemen beschrieben werden:

- Physikalische Schicht – die Schicht 1 (Physical Layer)
  Die unterste Schicht stellt die Dienste zur physikalischen Datenübertragung bereit.

- Datenübertragungsschicht – die Schicht 2 (Data Link Layer)
  Die Aufgabe dieser Ebene ist, eine sichere Datenübertragung zwischen den Busknoten zu gewährleisten.

  Das CAN-Protokoll spezifiziert nachrichtenorientierte Telegramme. Da die Telegramme beim CAN-Bus per Broadcasting übertragen werden, prüft jeder Knoten, ob diese Nachricht für ihn relevant ist.

- Anwendungsschicht – die Schicht 7 (Application Layer)
  Die Anwendungsschicht setzt direkt auf den Schichten 1 und 2 des CAN-Referenzmodells auf. Sie stellt die Schnittstelle für die Anwender des Feldbus dar. Über diese Schnittstelle können Feldbustelegramme versendet oder empfangen werden.

Die Dienste der Schichten 3 – 6 entfallen entweder oder sind in den Schichten 1, 2 und 7 implementiert (siehe Abbildung 2.32):

- Netzwerkschicht (Network Layer)
  Die Netzwerkebene kann entfallen, da es keine Vermittlungs- oder Routing-Funktion in einem CAN-Netzwerk gibt. Alle Nachrichten werden per Broadcasting an alle Netzknoten verschickt.

- Transportschicht (Transport Layer)
  Eine wichtige Aufgabe der OSI-Transportschicht ist das Erkennen von fehlerhaften oder duplizierten Nachrichten. Diese Aufgabe übernimmt beim CAN-Bus die Datenübertragungsschicht.

- Sitzungsschicht (Session Layer)
  Jede CAN-Nachricht wird sofort nach Erhalt der Nutzdaten vom Empfänger bestätigt. Dadurch wird das Konzept von Sitzungen mit Synchronisationspunkten und Roll-back Mechanismen überflüssig.

- Präsentationsschicht (Presentation Layer)
  Aufgabe dieser Schicht ist die Aufbereitung der über das Netz versendeten Applikationsdaten. Bei CAN wird das Format und die Struktur der Daten in der Schicht 7 festgelegt.

2.4.6.2.3 Kommunikationsmodelle

In Anlehnung an die Kommunikationsprotokolle anderer Feldbusse (z.B. Profibus) unterstützt das CANopen Protokoll des CAN-Bus verschiedene Nachrichtentypen (Communication Objects, COBs) sowie unterschiedliche Möglichkeiten zum Versenden dieser Telegramme.
Die Typen von Nachrichten sind [CAN95a, CAN95b, CAN97]:

- Nachrichten mit Servicedaten (*Service Data Objects*, SDOs)

- Nachrichten mit Prozeßdaten (*Process Data Objects*, PDOs)

- administrative Nachrichten (*administrative Messages*)

- vordefinierte Nachrichten (*pre-defined Communication Objects*)

Nur die *Service Data Objects* und die *Process Data Objects* werden zur Übertragung von Nutzdaten verwendet. Durch Servicedatennachrichten ist ein Zugriff auf alle Daten eines Geräts (diese werden in einem sogenannten Objektverzeichnis gespeichert) möglich. Daher

wird dieser Nachrichtentyp hauptsächlich zur Konfiguration von CANopen Modulen verwendet. PDOs werden dagegen zur Echtzeitübertragung von Prozeßdaten eingesetzt. Mit ihnen werden nur bestimmte vordefinierte Daten ausgetauscht.

Service Data Objects (SDO)	**Process Data Objects (PDO)**
niedere Priorität für Datenaustausch	Echtzeitdatenaustausch, hohe Priorität
asynchrone Übertragung	synchrone und asynchrone Übertragung zyklische und azyklische Übertragung
flexibler Zugriff auf alle Daten des Geräts (Multiplexed Domain Protocol)	Inhalt des Telegramms durch Mapping festgelegt
Datenlänge unbegrenzt	Datenmenge auf 8 Bytes beschränkt
Initiative vom Kommunikations-Master	Initiative von Slave (Gerät) oder Master
Bestätigung durch Slave	nur implizite Bestätigung

Tabelle 2.7: Vergleich von Service Data Objects (SDOs) und Process Data Objects (PDOs)

Administrative Nachrichten werden zum Versenden schichtspezifischer Parameter (Layer Management LMT), zur Initialisierung und Konfiguration (Network Management NMT und Identifier Distributor DBT), sowie zur Überwachung des Netzwerks verwendet.
Die vordefinierten Nachrichten dienen zur Synchronisation der Netzwerkknoten, zur Übertragung von Zeitmarken und zum Erkennen von Notfällen (Emergency Notification).

2.4.6.2.4 Anwendungen

Feldbusse werden weitverbreitet eingesetzt. Gründe für die Verwendung von Feldbussen sind zum einen die hohe Zuverlässigkeit der Datenübertragung und die zumeist niedrigen Kosten von Feldbussen (z.B. CAN-Bus).
Sie werden unter anderem zur Kommunikation innerhalb von Fertigungszellen oder zwischen der Zellensteuerung und überlagerten Planungssystemen [Shi91, SC96] eingesetzt.
Bei der Kommunikation innerhalb von Montagezellen gibt es zwei allgemeine Anwendungsfälle: Zum einen werden Daten über Feldbusse zwischen einzelnen Steuerungsrechnern ausgetauscht (siehe Abschnitt "Kombination mehrerer Steuerungstypen" in Abschnitt 2.4.6.1). Zum anderen kommunizieren Steuerungsrechner mit den gesteuerten Geräten über einen Feldbus, wobei unterschiedlichste Gerätetypen (z.B. einfache E/A Baugruppen oder auch komplexe Roboterantriebe [DFP95, CDSM97, KGSL97a, KGSL97b]) angesprochen werden. Speziell für die Kommunikation zwischen Steuerung und Automatisierungsgeräten sind bei verschiedenen Feldbussen sehr ähnliche Konzepte entwickelt worden. Diese basieren auf der Definition von Geräteprofilen. Die Profile legen das Verhalten und die Attribute einer bestimmten Gerätegruppe fest. Beispiele für solche Profile sind das CANopen Geräteprofil für E/A Module (CAL based Device Profile for I/O Modules, DS-401 [CAN95b]) oder das CANopen Geräteprofil für Antriebe und Antriebsregler (CAL based Device Profile for Drives and Motion Control, DS-402 [CAN97]).

Kapitel 3

Bau einer objektorientierten universellen Robotersteuerung

In diesem Kapitel wird die Entwicklung und die Architektur des ersten lauffähigen Steuerungssystems, das im Rahmen dieser Arbeit entstanden ist, vorgestellt. Dieses erste System wird als universelle Robotersteuerung HighRobot bezeichnet, da es unterschiedliche Steuerungstypen integriert (Robotersteuerung, SPS, Transfersystemsteuerung) [1].
Zum Bau der universellen Robotersteuerung wird das Vorgehen von G. Booch [Boo94] verwendet. Die einzelnen Schritte sind hierbei Konzeptualisierung, Analyse, Design und Implementierung. In Kapitel 6 wird ein Überblick über Implementierungsdetails aller entwickelter Systeme gegeben. Eine Beschreibung der Wartung entfällt, da anstelle der Wartung die Weiterentwicklung wie in Kapitel 4 beschrieben erfolgt. An dieser Phaseneinteilung orientieren sich die meisten objektorientierten Entwicklungsmethoden für Echtzeitsysteme (siehe Abschnitt 2.3.4.2). Daher findet sie auch in dieser Arbeit Anwendung.

3.1 Konzeptualisierung

Im wesentlichen werden in der Phase der Konzepterstellung zunächst die Aufgabenstellung (Problem Statement, siehe Abschnitt 1.1) sowie die Rahmenbedingungen der Entwicklung festgelegt. Im Anschluß werden die möglichen alternativen Konfigurationen und deren Nachteile beschrieben. Darauf folgt eine erste, grobe Gliederung der im Rahmen der Arbeit verwendeten Systemkomponenten. Abschließend werden die zu erwartenden Probleme (Risiken) abgeschätzt und entsprechende Prototypen mit Lösungsansätzen entwickelt.

3.1.1 Rahmenbedingungen der Entwicklung

Zusätzlich zu den allgemeinen Anforderungen an das Steuerungssystem müssen beim konkreten Bau eines ersten Systems weitere Bedingungen beachtet werden. Diese beziehen sich hauptsächlich auf die tatsächlich verwendete Hardware und Software, d.h. das Steuerungssystem und die angesteuerten Geräte.

3.1.1.1 Steuerungssystem

Als Steuerungsrechner wird eine SPARC Workstation (SPARC Clone HAMstation 19 von Hamilton, die einer SPARCstation 20 von Sun Microsystems leistungsmäßig entspricht) ver-

[1] Eine numerische Steuerung wird durch dieses Steuerungssystem nicht unterstützt, da zum Zeitpunkt der Entwicklung noch keine geeigneten zu kontrollierenden Geräte zur Verfügung gestanden haben.

wendet. Dieser Rechner besitzt zwei SuperSPARC Prozessoren mit einer Taktrate von 50 MHz und 64 MByte Hauptspeicher. Ähnliche Konfigurationen werden von verschiedenen Firmen (z.B. Tekelec, Force oder MEN) als industrietaugliche Rechner angeboten.

Der eigentliche Steuerungsrechner ist im Verlauf der Arbeiten durch eine weitere Workstation (Sun MicroSPARC) ergänzt worden. Dieser zweite Rechner ist bei der Software-Entwicklung und als Terminal benutzt worden.

Als Betriebssystem werden die zum Zeitpunkt der Arbeiten aktuellen Releases des am POSIX-Standard orientierten Betriebssystems Solaris eingesetzt (Solaris 2.4 und Solaris 2.5 von Sun Microsystems) [GGM93, KSS96].

Zur Programmierung wird C++ [Lip91, Str93, Str94] verwendet (GNU Compiler 2.7.1).

Der Steuerungsrechner und die gesteuerten Geräte sind über den Feldbus CAN verbunden. Das Kommunikationsprotokoll ist CANopen. Die Workstation wird über eine Schnittstellenkarte (SB01 der Fa. Stock Microcomputersysteme) an den CAN-Bus angebunden.

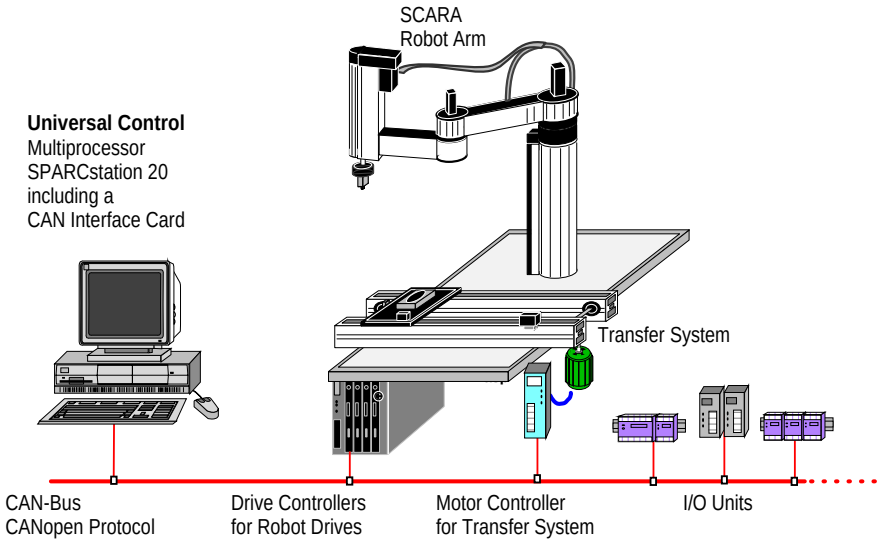


Abbildung 3.1: Gesteuerte Geräte

3.1.1.2 Gesteuerte Geräte

Durch die universelle Robotersteuerung werden nur Geräte mit einer Schnittstelle zum CAN-Bus angesteuert. Im wesentlichen handelt es sich um einen SCARA Roboterarm (SR60 von Fa. Bosch) mit CAN-fähigen Servoantrieben der Fa. MOOG. Zusätzlich werden eine größere Anzahl von digitalen E/A Modulen unterschiedlicher Hersteller und ein ungeregelter Antrieb der Fa. Lenze angesteuert.

Diese Geräte sind in einer Roboterzelle montiert. Diese Zelle ist eine zu Demonstrationszwecken umgebaute marktgängige Roboterzelle von Bosch. Alternativ zu der hier beschriebenen Steuerung kann die Roboterzelle auch von drei miteinander kommunizierenden traditionellen Steuerungssystemen gesteuert werden.

3.1.2 Alternative Konfigurationen

Neben der realisierten Variante der Anbindung der Prozeßperipherie (Roboterkinematik und weitere Geräte) an den Steuerrechner über den CAN-Bus mittels Schnittstellenkarte, sind noch weitere Alternativen untersucht worden:

- Entwicklung einer Einschubkarte, die die Funktionen eines oder mehrerer Antriebsverstärker übernimmt.
  Dadurch können die Servoantriebe eines Roboterarms direkt von dieser Karte aus angesteuert werden. Die Regelung der Antriebe findet auf der Karte statt. Die PC-basierte Robotersteuerung KR C1 von Kuka stellt ein derartiges System dar. Diese Steuerung verwendet speziell für den Steuerungs-PC entwickelte Karten zur Regelung der Antriebe. Die Leistungsverstärker der Antriebe befinden sich in einer externen Baugruppe, die mit den Reglerkarten über eine proprietäre Verbindung kommuniziert.

  Die Nachteile dieses Ansatzes bestehen darin, daß derartige Einschubkarten nur eine stark begrenzte Anzahl von Geräten ansteuern können. Zudem ist die Entwicklung dieser Karten sehr teuer, da sie keine Standardkomponenten sind.

- Einsatz eines VME-Kartensystems
  Aufgrund der hohen Flexibilität eines derartigen Rückwandsystems können hier unterschiedliche Typen von Standardrechnern (Workstations oder PCs), verschiedene Antriebsverstärkerkarten und Feldbusanbindungen integriert werden.

  Der Vorteil von VME-Kartensystemen liegt darin, daß von sehr vielen Firmen Komponenten angeboten werden. Allerdings sind derartige Baugruppen sehr teuer.

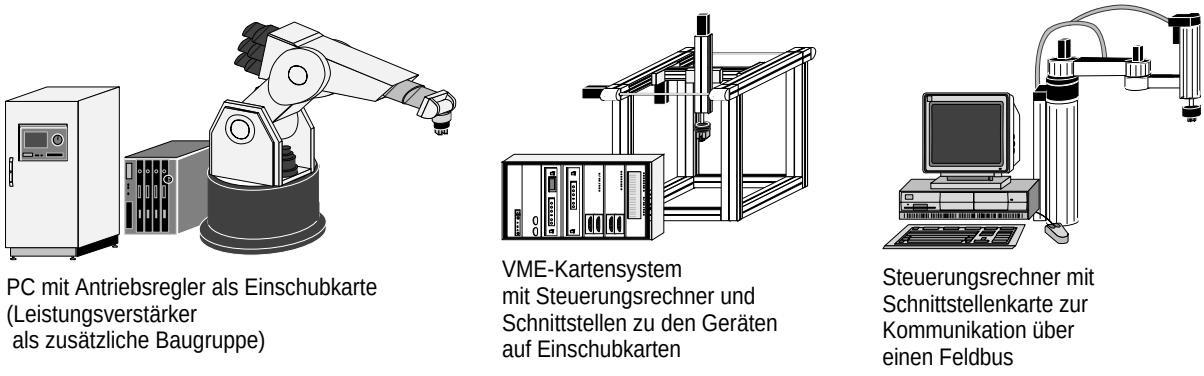


Abbildung 3.2: Konfigurationsvarianten des Steuerungssystems

Im Gegensatz zu den anderen Varianten bietet die tatsächlich realisierte Konfiguration (Steuerung kommuniziert mittels Schnittstellenkarte über einen Feldbus) mehrere Vorteile. Zum einen kann sie sehr kostengünstig umgesetzt werden, da auf in großem Umfang produzierte Standard-Hardware zurückgegriffen werden kann. Zum anderen ist sie sehr skalierbar, d.h. über den Feldbus kann eine stark variierende Menge an Geräten angesprochen werden. Der verwendete Steuerungsrechner kann leicht durch eine andere Hardware ersetzt werden (z.B. Austausch der Workstation gegen einen PC). Dies schließt auch einen Umbau auf ein VME-System mit ein.

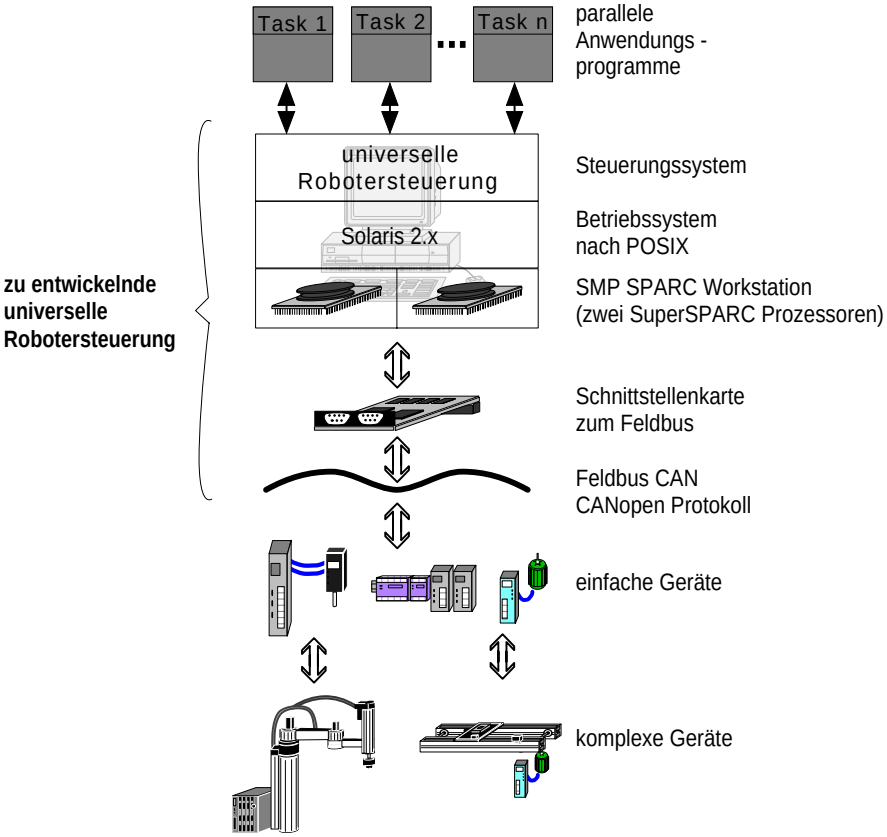


Abbildung 3.3: Systemkomponenten

3.1.3 Gliederung der Systemkomponenten

Entsprechend der an die Steuerung gestellten Anforderungen und der gewählten Konfigurationsvariante können die einzelnen grobgranularen Komponenten (Hardware und Software) hierarchisch gegliedert werden (siehe Abbildung 3.3).

An oberster Stelle stehen die parallelen Anwendungsprogramme. Diese sind vom Benutzer geschriebene Steuerungsprogramme und enthalten Anweisungen, wie sich die Geräte verhalten sollen. Ausgeführt werden diese Applikationsprogramme auf der universellen Robotersteuerung.

Die eingesetzten Betriebssysteme (Solaris 2.4 und 2.5) unterstützen die parallele Verarbeitung von Prozessen auf der SMP (Shared Memory Multiprocessor) Workstation. Die Geräte sind mit dem Steuerungsrechner über den Feldbus CAN verbunden. Zur Feldbusanbindung besitzt die Workstation eine Schnittstellenkarte.

Die Geräte werden in zwei Gruppen eingeteilt: Die einfachen Geräte bilden die Basis für die komplexen Geräte. Einfache Geräte sind beispielsweise geregelte und nichtgeregelte Antriebe oder E/A Module. Aus diesen sind die komplexen Geräte zusammengesetzt. Ein SCARA Roboter besteht z.B. aus vier positionierfähigen geregelten Servoantrieben (siehe Abschnitt 2.4.2.3).

Die Schichten *parallele Anwendungsprogramme* sowie *einfache* und *komplexe Geräte* sind nicht Teil der entwickelten Robotersteuerung.

Die Anwendungsprogramme müssen (wie bei jeder Steuerung) vom Benutzer selbst implementiert werden. Die Steuerung bietet nur die Programmierschnittstelle und die Applikationsklassenbibliothek, mit deren Hilfe Anwendungen geschrieben werden.

Die geräteinterne Software ist fester Bestandteil der gesteuerten Automatisierungsgeräte. Daher kann sie nicht durch die Robotersteuerung verändert werden. Diese geräteinterne Software bietet die Schnittstelle, über die die Geräte von der Robotersteuerung angesprochen werden.

3.1.4 Risiken und Prototypen

Die beiden zentralen Probleme bei der Entwicklung der universellen Steuerung sind das zeitliche Verhalten des Steuerungssystems sowie die Kommunikation über den Feldbus CAN. Das Zeitverhalten des Betriebsystems betrifft unter anderem das Scheduling (bzw. die Task-Wechselzeiten), die Genauigkeit eines periodischen, betriebssysteminternen Zeitgebers und die Abweichung von diesem festen Zeittakt. Hierbei hat sich gezeigt, daß ein Kontextwechsel von hochprioren Prozessen (oder Threads) in durchschnittlich 0,7 ms erfolgt. Bei einer Periodendauer von 20 ms ist die Abweichung (Jitter) geringer als 0,5 ms. Zusätzlich kann der hochpräzise, ($< 0{,}1$ ms) von der CAN-Schnittstellenkarte per Hardware Timer erzeugte Takt von der Steuerung genutzt werden.
Ebenso müssen im Vorfeld die Möglichkeiten zur Kommunikation und Synchronisation zwischen Prozessen und Threads unter Echtzeitbedingungen untersucht werden. Durch Versuche sind folgende Mechanismen als verwendbar verifiziert worden: Echtzeitsignale nach POSIX.4 [Gal95] und System V Semaphore [Bac86, Tan92] als Synchronisationsmechanismen zwischen Prozessen sowie Shared Memory nach POSIX.1 [GGM93] für den Austausch von Daten. Zusätzlich zu den Prozessen können die proprietären Threads von Solaris und deren Synchronisations- und Kommunikationsmechanismen verwendet werden [MTP95, KSS96].
Die Untersuchung der CAN-Schnittstellenkarte hat ergeben, daß alle für das CANopen Protokoll notwendigen Telegrammtypen rechtzeitig über die Karte versendet und empfangen werden können.

3.2 Analyse

In der Analyse werden zunächst die externen Objekte (z.B. Geräte) und die Kommunikationsbeziehungen zwischen diesen Objekten und der Steuerung vorgestellt. Anschließend folgt die Beschreibung der funktionalen und nichtfunktionalen Anforderungen. Den Abschluß des Kapitels bilden die statischen und dynamischen Analysemodelle.

3.2.1 Externe Objekte

Zu Beginn der Analyse werden zunächst die externen Objekte (z.B. zu steuernde Geräte oder Benutzer) und die Kommunikationsbeziehungen zwischen den externen Objekten und dem zu entwickelnden System bestimmt (siehe Abschnitt 2.3.4.2).

3.2.1.1 Schalenmodell

Im Schalenmodell werden die externen Objekte in verschiedene Schalen eingeordnet [Ell94]. Diese externen Objekte werden aus der Gliederung der Systemkomponenten in Abschnitt 3.1.3 abgeleitet.

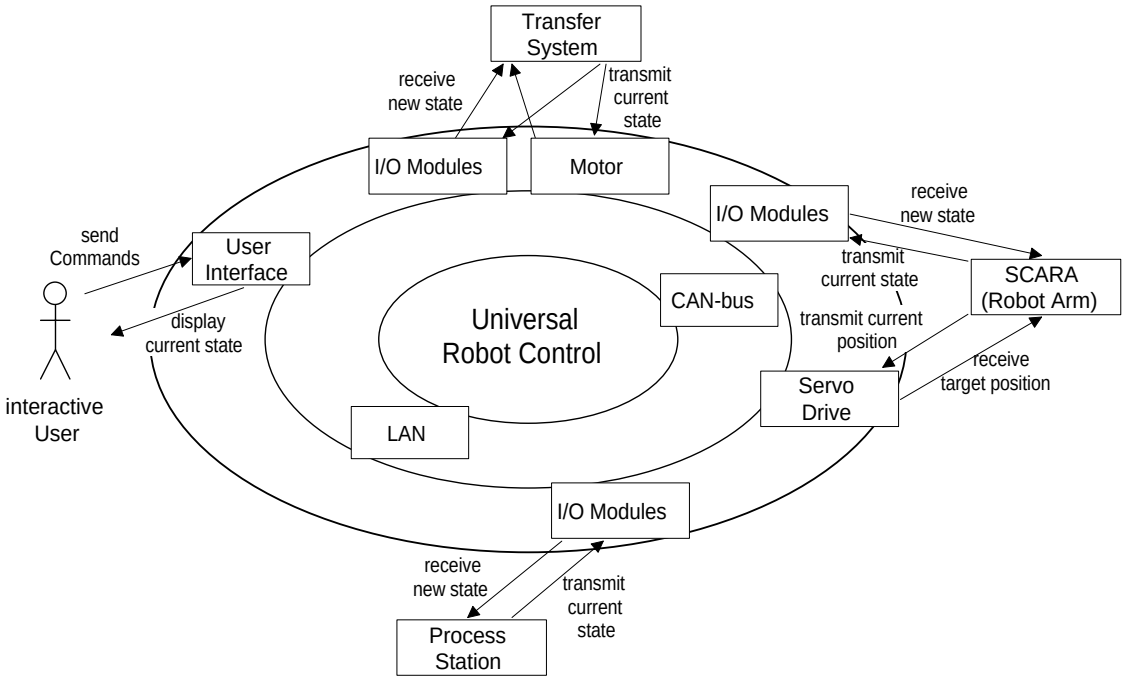


Abbildung 3.4: Schalenmodell nach [Ell94]

In der innersten Schale befindet sich das zu entwickelnde System: die universelle Robotersteuerung (*Universal Robot Control*). In der nächsten Schale (Schnittstellenschale, *Interfaces*) sind die Kommunikationselemente *CAN-bus* und *LAN*. Durch den CAN-Bus werden die einzelnen Geräte angesteuert. Das lokale Netzwerk ermöglicht den entfernten Zugriff auf die Steuerung. Da die Workstation als Steuerungsrechner standardmäßig immer eine LAN-Schnittstelle besitzt, soll diese auch durch die Steuerung genutzt werden. Die einfachen Gerätetypen (*I/O Modules*, *Motor* und *Servo Drives*) sowie das *User Interface* sind Teil der *Sensors / Actuators* Schicht. Die einfachen Geräte sind Bestandteil der komplexen Geräte (*Transfer System*, *Process Station* und *SCARA*). Diese finden sich zusammen mit dem Benutzer in der äußersten Schale.

Da sich die externen Objekte im Design der zu entwickelnden Software wiederfinden, gibt diese erste Einordnung Aufschluß über die Architektur des Steuerungssystems.

3.2.1.2 Kontextdiagramm

Im Kontextdiagramm werden die Kommunikationsbeziehungen zwischen den im Schalenmodell beschriebenen externen Objekten näher spezifiziert.

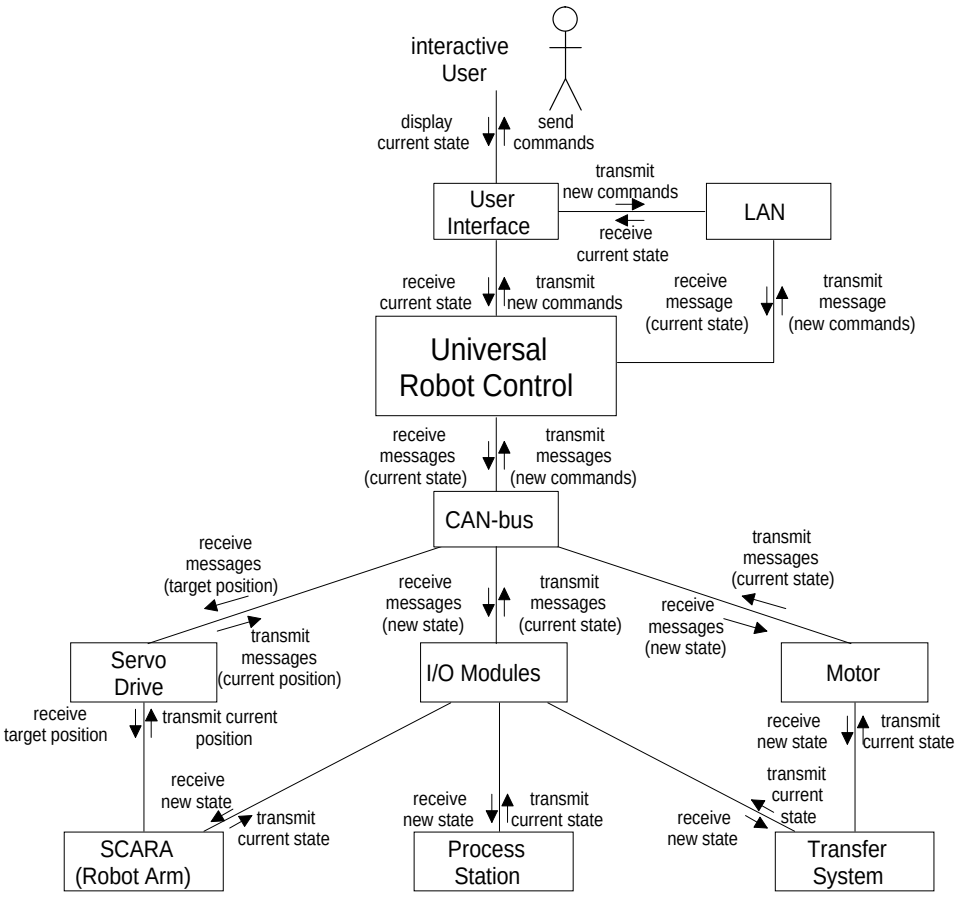


Abbildung 3.5: Kontextdiagramm

Die jeweils äußersten Objekte kommunizieren über die Zwischenschichten (im Schalenmodell) mit der universellen Robotersteuerung.

- interaktiver Benutzer (*interactive User*):
  Das Benutzermenü zeigt dem Anwender den jeweils aktuellen Status der Geräte an. Er kann interaktiv Befehle an die einzelnen Geräte verschicken oder auch Applikationsprogramme starten.

  Auf die Anwenderschnittstelle kann entweder lokal oder entfernt zugegriffen werden. Ein entfernter Zugriff soll zunächst nur über das Telnet-Protokoll möglich sein.

- CAN-Bus
  Mit diesem Feldbus werden alle Geräte an den Steuerungsrechner verbunden. Als Protokoll wird CANopen verwendet. Daher muß dieses Protokoll in der Steuerung implementiert werden.

- Geräte
  Die Geräte sind entsprechend der Gliederung der Systemkomponenten (siehe Abschnitt 3.1.3) angeordnet.

3.2.2 Funktionale Anforderungen

Bei fast allen objektorientierten Software-Entwicklungsmethoden werden Use Case Modelle [JC95] zur Beschreibung der funktionalen Anforderungen verwendet (siehe Abschnitt 2.3.3 und 2.3.4). Diese werden durch Szenarien, die durch bestimmte Ereignisse ausgelöst werden, genauer spezifiziert. Am Ende des Abschnitts werden die zu realisierenden Steuerungsfunktionen vorgestellt.

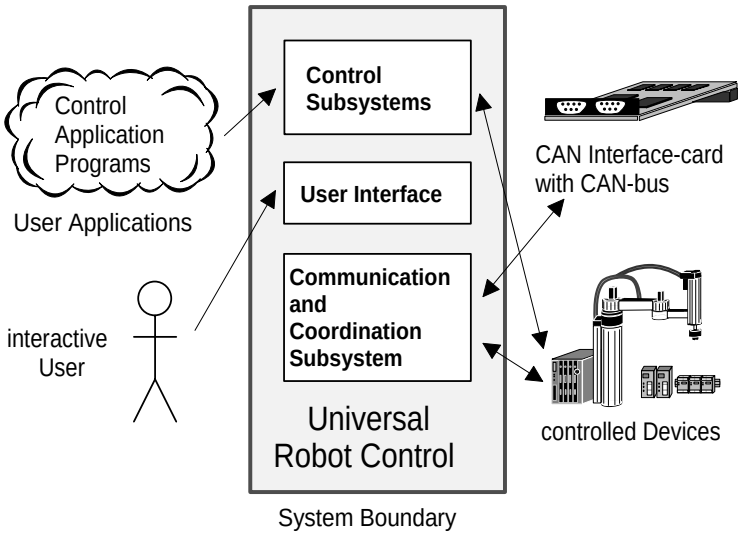


Abbildung 3.6: Use Case Diagramm

3.2.2.1 Use Case Modell

Use Case Modelle ermöglichen eine erste Gliederung der funktionellen Elemente eines Systems.
In einem ersten Use Case Modell (siehe Abbildung 3.6) können folgende Use Cases identifiziert werden:

- Die vom Benutzer geschriebenen Anwendungsprogramme greifen auf die *Control Subsystems* zu. Diese beinhalten die einzelnen Steuerungsfunktionen. Sie umfassen unter anderem Funktionen zur Steuerung eines Roboters und eines Transfersystems sowie eine speicherprogrammierbare Steuerung. Eine weitere Beschreibung folgt in Abschnitt 3.2.2.3.

- Der Anwender kann über das *User Interface* auf die Steuerung direkt einwirken. Es können Anwendungsprogramme gestartet oder terminiert werden. Ebenso kann sich der Benutzer die aktuellen Zustände der Geräte anzeigen lassen und direkt Kommandos an die Geräte verschicken. Das *User Interface* ist eine interaktive Schnittstelle.

- Die Kommunikation mit den Geräten sowie die Koordination der nebenläufigen Applikationsprogramme (z.B. Robotersteuerungsapplikationen oder SPS-Programme). geschieht durch das *Communication and Coordination Subsystem*. Es bildet die zentrale Kontrollinstanz innerhalb des Gesamtsystems. Zur Kommunikation der *Application Tasks* mit den von ihnen gesteuerten Geräten bietet das *Communication and Coordination Subsystem* eine virtuelle Geräteschnittstelle (siehe Abbildung 3.7). Diese nimmt die Nachrichten von den *Application Tasks* zu den Geräten entgegen und liefert die Informationen von den Geräten an die *Application Tasks* zurück.

Die Use Cases beschreiben die Interaktionen mit den Aktoren, die nicht Teil des zu entwickelnden Systems sind. Der Benutzer und die vom Benutzer geschriebenen Anwendungsprogramme stellen Aktoren dar. Die Applikationsprogramme verwenden die Steuerungsfunktionen der *Control Subsystems*. Die weiteren Aktoren sind die zu steuernden Geräte und die Kommunikationsverbindungen zwischen den Geräten und dem Steuerungsrechner. Diese beiden Typen von Aktoren kommunizieren mit der Robotersteuerung über das *Communication and Coordination Subsystem*.

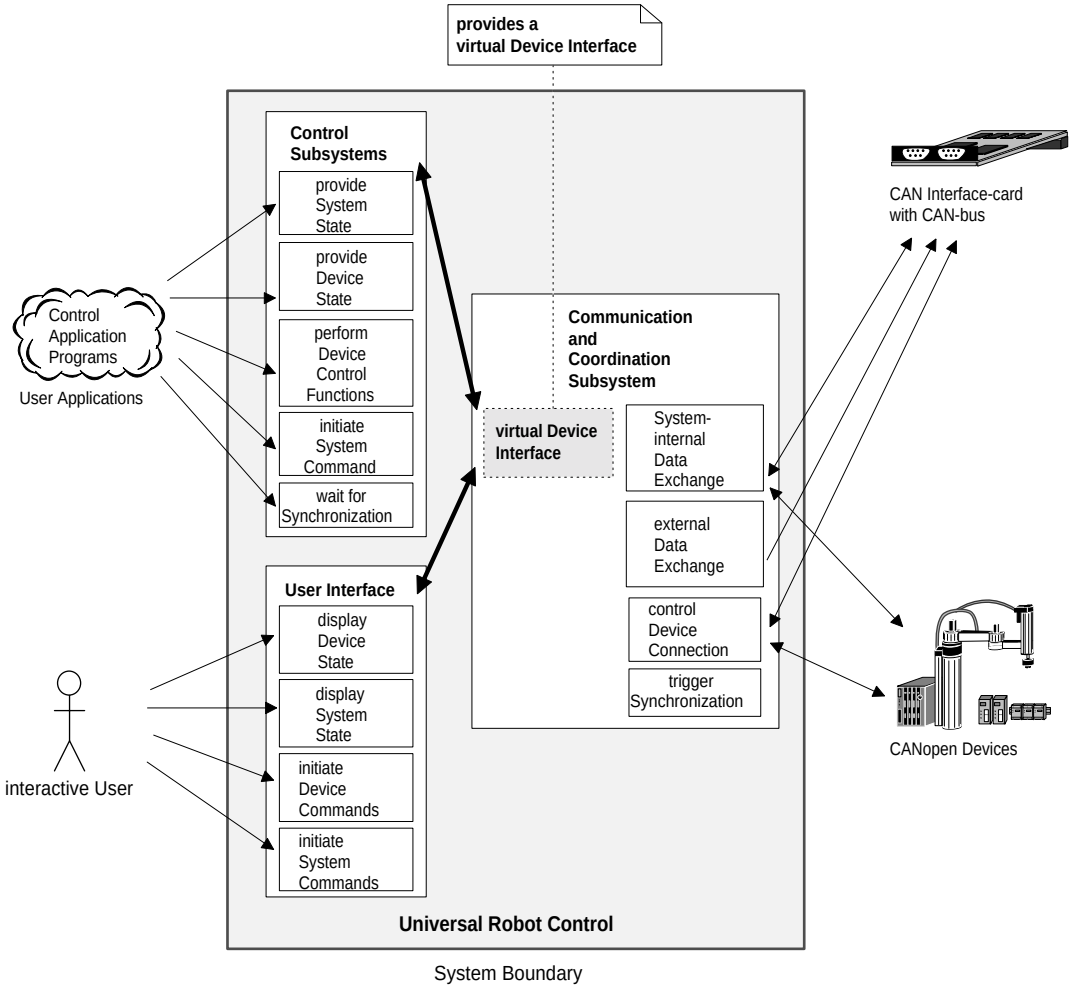


Abbildung 3.7: Erweitertes Use Case Diagramm

Zur detaillierten Beschreibung wird das initiale Use Case Modell erweitert (siehe Abbildung 3.7). Die Aktoren werden unverändert übernommen. Es ergeben sich folgende weitere

detaillierte Use Cases innerhalb der im ersten Diagramm gefundenen:

- Use Cases innerhalb der *Control Subsystems*:
  In den Anwendungsfällen *provide System State* und *provide Device State* wird Auskunft über den aktuellen Zustand der Geräte oder des Steuerungssystems gegeben.

  Den Zugriff aus den Applikationsprogrammen auf die gesteuerten Geräte beschreibt der Use Case *perform Device Control Functions*. Der Anwendungsfall *initiate System Command* zeigt, wie das Steuerungssystem beeinflußt wird.

  Wait for Synchronization wartet auf die Synchronisationssignale des *Communication and Coordination Subsystem*. Eine derartige Synchronisation ist notwendig, da das Protokoll zur Kommunikation (Feldbusprotokoll CANopen) mit den Geräten einen synchronisierten Datenaustausch erfordert.

- Use Cases innerhalb des *User Interface*:
  Display Device State und *display System State* zeigen dem Anwender den jeweiligen Zustand der Steuerung und Geräte an.

  Im Gegenzug kann der Anwender in den Anwendungsfällen *initiate System Command* und *initiate Device Command* entsprechende Manipulationen vornehmen.

- Use Cases innerhalb des *Communication and Coordination Subsystem*:
  Die Anwendungsfälle *System-internal Data Exchange* und *external Data Exchange* beschreiben die Nachrichteninhalte der Kommunikation innerhalb des Steuerungssystems und mit den gesteuerten Geräten. Daneben bestimmt der Use Case *control Device Connection* das Kommunikationsprotokoll zwischen der Steuerung und den Geräten.

  Der Anwendungsfall *trigger Synchronization* erläutert die Synchronisation und Koordination der *Application Tasks* und der Kommunikation mit den Geräten.

Szenario	**Use Cases**		
	Control Subsystems	*User Interface*	*Communication and Coordination Subsystem*
1. Lesen des Steuerungszustands	*provide System State*	*display System State*	*System-internal Data Exchange, control Device Connection*
2. Lesen des Zustands der Geräte	*provide Device State*	*display Device State*	*System-internal Data Exchange, external Data Exchange*
3. Senden von Kommandos an das Steuerungssystem	*perform System Command*	*initiate System Command*	*System-internal Data Exchange, control Device Connection*
4. Senden von Kommandos an die gesteuerten Geräte	*perform Device Command*	*initiate Device Command*	*System-internal Data Exchange, external Data Exchange*
5. Synchronisation	*wait for Synchronization*	—	*trigger Synchronization*

Tabelle 3.1: Zuordnung der Anforderungsszenarien zu den Use Cases

3.2.2.2 Szenarien und Ereignisse

Szenarien beschreiben die dynamischen Handlungsabläufe zwischen den Use Cases und Aktoren, die über die Beziehungen ablaufen. Die einzelnen Szenarien leiten sich von dem detaillierten Use Case Modell ab. Sie werden von Ereignissen ausgelöst, die daher ebenfalls hier vorgestellt werden.

In Tabelle 3.1 sind die Szenarien und die an den Szenarien beteiligten Use Cases aufgelistet. Die Ereignisse, die diese Szenarien auslösen, finden sich in Tabelle 3.2.

Auf eine hierarchische Strukturierung der Ereignisse, wie sie bei vielen Echtzeitsystemen üblich ist [Dou98b], kann bei dem universellen Steuerungssystem verzichtet werden, da der Datenaustausch zwischen den Geräten und der Steuerung immer streng zyklisch abläuft, d.h. die Ereignisse treffen sequentiell und werden in der Reihenfolge ihres Eintreffens abgearbeitet.

Bei dem entwickelten Steuerungssystem werden alle ohne Zeitverzögerung zu bearbeitenden Ereignisse von den Geräten selbst entsprechend bearbeitet (wie z.B. der Not-Halt der Roboterantriebe durch elektrische Signale zwischen den Antriebsverstärkern).

Bei Notfallereignissen protokolliert die Steuerung lediglich, daß ein Notfall eingetreten ist und wie das betroffene Gerät (oder die Gerätegruppe) darauf reagiert hat.

Ereignis	**Szenario**
cyclic Synchronization Signal	Synchronisation
Read Event System State	Lesen des Steuerungszustands
Read Event Device State	Lesen des Zustands der Geräte
Write Event to System	Senden von Kommandos an das Steuerungssystem
Write Event to Devices	Senden von Kommandos an die gesteuerten Geräte

Tabelle 3.2: Zuordnung der Ereignisse zu den Szenarien

Alle Vorgänge in der Steuerung entsprechend dem üblichen Verfahren bei Robotersteuerungen, SPSen oder NCs zyklisch ab (siehe Abschnitte 2.4.2.2.2, 2.4.3.2 und 2.4.4.2) und orientieren sich an einem Synchronisationssignal. Die übrigen Ereignisse können nur innerhalb bestimmter Zeitfenster im Steuerungszyklus stattfinden (siehe Abbildung 3.8). Die Ereignisse *Read Event System State* und *Read Event Device State* sind nur zu Beginn des Vorgangs *process Application* möglich. Dagegen treten *Write Event to System* und *Write Event to Devices* nur zum Abschluß von *process Application* ein.

Die Ereignisse, die vom Benutzer (über das *User Interface*) ausgelöst werden, scheinen zunächst von der zyklischen Synchronisation unabhängig zu sein.

Der Benutzer kann asynchron Daten über das *User Interface* abfragen, bzw. eingeben. Allerdings erfolgt der Datenaustausch zwischen dem *User Interface* und dem *Communication and Coordination Subsystem* ebenso synchron wie der Datentransfer zwischen den *Control Subsystems* und dem *Communication and Coordination Subsystem*. Der Anwender kann also nur die jeweils in einem bestimmten Zyklus aktuellen Daten auslesen und seine Anweisungen ebenfalls nur synchron innerhalb des aktuellen Zyklus verschicken. Damit verhält sich das *User Interface* ähnlich wie die *Control Subsystems*.

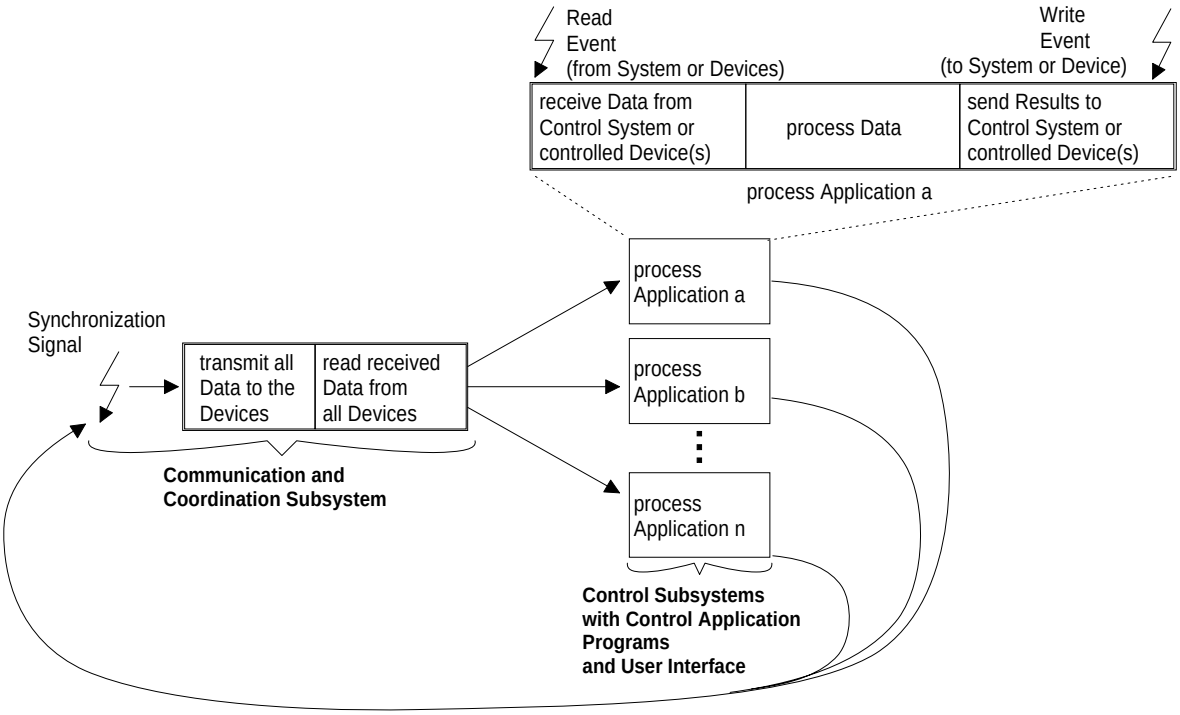


Abbildung 3.8: Ereignisse im dynamischen Verhalten der Steuerung

3.2.2.3 Steuerungsfunktionalität

Die Steuerungssubsysteme (*Control Subsystems* aus Abbildung 3.9) enthalten die Methoden zur Steuerung der Geräte. Entsprechend der in der Konzeptualisierung (siehe Abschnitt 3.1.3) beschriebenen Einteilung werden Steuerungsfunktionen für einfache und komplexe Geräte entwickelt.

- Komplexe Steuerungssubsysteme:
  - Das Subsystem zur Steuerung des SCARA Roboterarms (*SCARA Control*) bietet die Möglichkeit zur PTP- und Linearfahrt mit wahlfreier Geschwindigkeit. Aufgrund der Roboterantriebe sind kreisförmige Bewegungen nicht möglich.
    Das Roboterarmsteuerungssubsystem greift auf vier einfache Subsysteme zur Ansteuerung von Antrieben zu.
  - Das speicherprogrammierbare Steuerungssubsystem (*Soft-PLC*) behandelt die zentrale Funktionalität einer SPS, wie z.B. binäre Verknüpfungen, Zähler oder Zeitgeber. Dabei können die Anwendungsprogramme zyklisch (d.h. in der Art von SPS-Programmen, siehe Abschnitt 2.4.3.2) oder in Einzelschritten (d.h. nach Bearbeitung einer einzelnen Befehlszeile wird das Ergebnis direkt an die Geräte weitergeleitet) abgearbeitet werden. Das SPS-Subsystem nutzt beliebig viele einfache E/A Subsysteme.
  - Das Subsystem zur Kontrolle des Transfersystems (*Transfer System*) ermöglicht das Verfahren und Festhalten von Transportschlitten auf dem Transfersystem. Dazu greift es auf das Subsystem *Motor* und ein E/A Subsystem zu.

- Einfache Steuerungssubsysteme:
  - Durch das Subsytem *Drive Control* kann auf geregelte Antriebe wie die Roboterantriebe zugegriffen werden.
  - Die E/A Subsysteme (*I/O Modules*) steuern die einzelnen E/A Baugruppen an.
  - Ungeregelte Elektromotoren werden über das Subsystem *Motor* angesprochen.

Alle Steuerungsfunktionen werden als Klassen, die in Applikationsprogramme eingebunden werden können, realisiert (siehe nichtfunktionale Anforderungen in Abschnitt 3.2.3).
Die hier beschriebenen Steuerungssubsysteme sollen innerhalb der Gliederung in einfache und komplexe Systeme beliebig ausgetauscht und modifiziert werden können (siehe Abschnitt 3.2.3). Beispielsweise kann das Robotersteuerungssubsystem durch ein Subsystem zur numerischen Steuerung (NC Subsystem) ausgetauscht werden.
Die aufgeführten Steuerungssubsysteme werden zum Bau der universellen Robotersteuerung verwendet. Im Rahmen der Wiederverwendung (siehe Abschnitt 4) erfolgt zum Teil der Tausch der Subsysteme durch hier nicht aufgelistete Systeme.

3.2.3 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen [Som92] ergeben sich zum einen aus der Anforderungsbeschreibung (siehe Abschnitt 1.1) zum anderen leiten sie sich von den weiteren, in der Konzeptualisierung (siehe Abschnitt 3.1) gewonnenen, Erkenntnissen ab.
Diese Anforderungen an die universelle Robotersteuerung sind:

- Verwendung von C++
  C++ kann als einzige objektorientierte Sprache auf allen Plattformen eingesetzt werden. Im Gegensatz zu anderen plattformunabhängigen Sprachen wie z.B. Java unterstützt C++ die Echtzeitanforderungen an das System.

  Die vom Anwender entwickelten Applikationsprogramme werden ebenso in C++ geschrieben und binden die Steuerungsfunktionen des Steuerungssystems als Klassen ein.

- Orientierung an Software-Standards und Verwendung von Standard-Hardware
  Bei der Entwicklung muß darauf geachtet werden, daß Standards eingehalten und verwendet werden. In dieser Arbeit sollten nach Möglichkeit nur POSIX konforme Systemaufrufe verwendet werden. Dadurch kann die universelle Robotersteuerung auf unterschiedliche Systemplattformen, wie z.B. Sun SPARCstation, PCs mit Windows NT oder Linux PCs, übertragen werden.

- Austauschbarkeit von Steuerungssubsystemen
  Das Konzept der universellen Robotersteuerung soll nicht nur auf die in Abschnitt 3.1.1.2 aufgeführten Geräte beschränkt bleiben. Zusätzlich soll die Steuerung von weiteren Automatisierungsgeräten vorgesehen werden. (Die Realisierung der Steuerung von weiteren Gerätetypen ist in Kapitel 4 beschrieben.)

- Unterstützung konkurrenter Anwendungen
  Mehrere konkurrente *Application Tasks* (als Prozesse oder Threads realisiert) sollen auf der Steuerung ablaufen können.

  Diese Steuerungsprogramme der universellen Steuerung müssen in Echtzeit ausgeführt werden können, bzw. müssen Automatisierungsgeräte in Echtzeit ansteuern können.

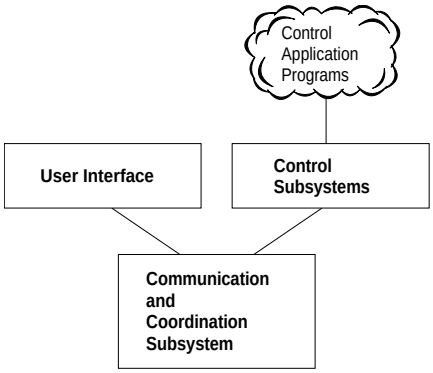


Abbildung 3.9: Initiales Klassendiagramm der Analyse

3.2.4 Analysemodell

In diesem Abschnitt sollen die Anforderungen in Form von Klassendiagrammen und dynamischen Beschreibungen modelliert werden. Diese Modelle geben einen ersten Überblick über die statische Hierarchie der Klassen und deren dynamisches Verhalten.

3.2.4.1 Klassenmodell

Im Klassenmodell der Analyse wird die statische Struktur des Problembereichs entsprechend den Anforderungen modelliert. In einer ersten Näherung besteht das Klassenmodell in Abbildung 3.9 aus *User Interface*, *Control Subsystems* und *Communication and Coordination Subsystem*. Diese Klassen werden direkt aus dem Use Case Modell (siehe Abschnitt 3.2.2.1) abgeleitet.
Zusätzlich ist *Control Application Programs* Teil des Klassenmodells. Diese Steuerungsprogramme werden vom Benutzer auf der Steuerung zur Ausführung gebracht und sind daher nicht Teil des eigentlichen Steuerungssystems. Zur vollständigen Modellierung der Anforderungen werden im Diagramm in Abbildung 3.9 und weiteren Darstellungen die Beziehungen zwischen diesen Programmen und dem universellen Robotersteuerungssystem dennoch gezeigt. Die *Control Application Programs* werden entweder prozedural oder objektorientiert entwickelt.
Die Subsysteme des initialen Klassenmodells werden in Abbildung 3.10 erweitert. Das *Communication and Coordination Subsystem* enthält die Klassen *Interprocess Communication*, *Coordination* und *Device Connection*. Die Aufgaben, die von diesen Klassen übernommen werden, sind aus dem erweiterten Use Case Modell (siehe Abbildung 3.7) abgeleitet:

- *Interprocess Communication* übernimmt *System-internal Data Exchange*
- Die Klasse *Device Connection* ist für die Kommunikation mit den gesteuerten Geräten sowie für die Kommunikationsverbindung verantwortlich. Daher fallen die Use Cases *external Data Exchange* und *control Device Connection* in ihren Aufgabenbereich.
- Die Koordination des dynamischen Systemverhaltens geschieht durch die Klasse *Coordination*. Sie steuert die Aktivitäten aller anderen Klassen im System und behandelt unter anderem den Use Case *trigger Synchronization*.

Die *Control Subsystems* werden entsprechend der Gliederung der Geräte in der Beschreibung der Systemkomponenten (siehe Abschnitt 3.1.3) in eine Klasse zur Kontrolle der einfachen

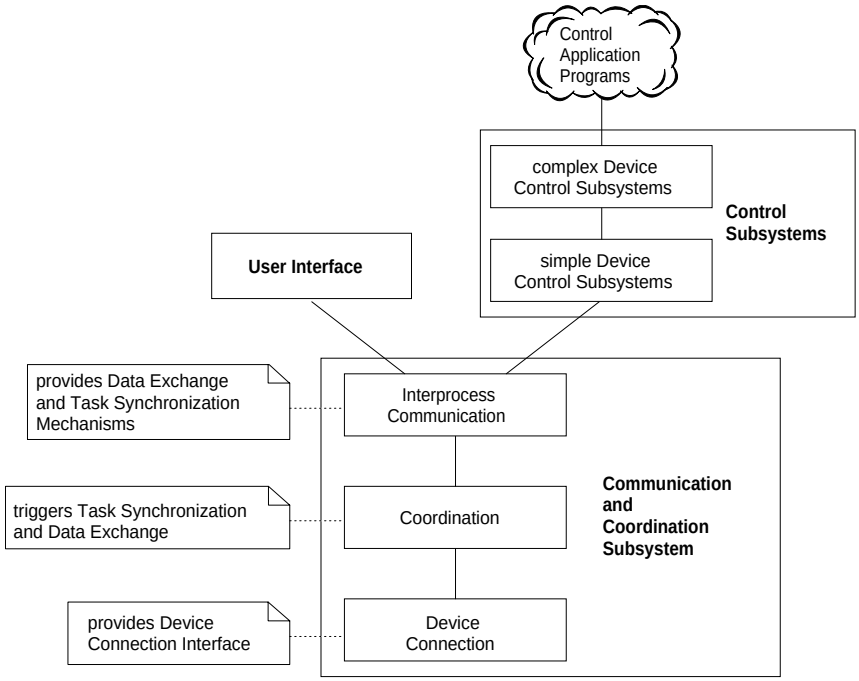


Abbildung 3.10: Erweitertes Klassendiagramm der Analyse

Geräte (*simple Device Control Subsystems*) und eine Klasse für komplexe Geräte (*complex Device Control Subsystems*) unterteilt.
Im ersten Ansatz wird das *User Interface* nicht weiter verfeinert, da vorläufig ein sehr einfaches ASCII-Terminal basiertes Menü den Anforderungen genügt.
Eine genauere Spezifikation der Beziehungen zwischen den Klassen erfolgt erst im Design.

3.2.4.2 Dynamisches Verhalten

Das dynamische Verhalten wird vorallem durch den internen Datenaustausch bestimmt (siehe Abbildung 3.11).
Die Klasse *Interprocess Communication* realisiert die Datenübertragung. Die Daten werden für die asynchrone Übertragung zwischengespeichert. Dadurch separiert diese Klasse die Kontrollsubsysteme und das *User Interface* von der zentralen Klasse *Coordination*, die die dynamischen Abläufe auslöst. Diese Aktionen laufen dann in einem starren zyklischen Zeitraster ab. Die einzelnen Aktivitäten sind:

1. Lesen und Versenden der Kommandos an die gesteuerten Geräte
   Coordination liest die an die Geräte zu sendenden Kommandos von der Klasse *Interprocess Communication*. Diese Kommandos sind im vorhergehenden Zyklus in *Interprocess Communication* gespeichert worden (siehe Aktion 4.). Die zu versendenden Daten werden anschließend über *Device Connection* an die Geräte weitergeleitet.

   Auf diese Weise werden auch Anweisungen an die zentrale Klasse des Steuerungssystems (*Coordination*) weitergegeben.

2. Einlesen des Zustands der Geräte
   Coordination liest den aktuellen Zustand der Geräte (z.B. aktuelle Position eines Roboterarms oder Eingangswerte eines E/A Moduls) über die Klasse *Device Connection* und speichern sie im Puffer von *Interprocess Communication*.

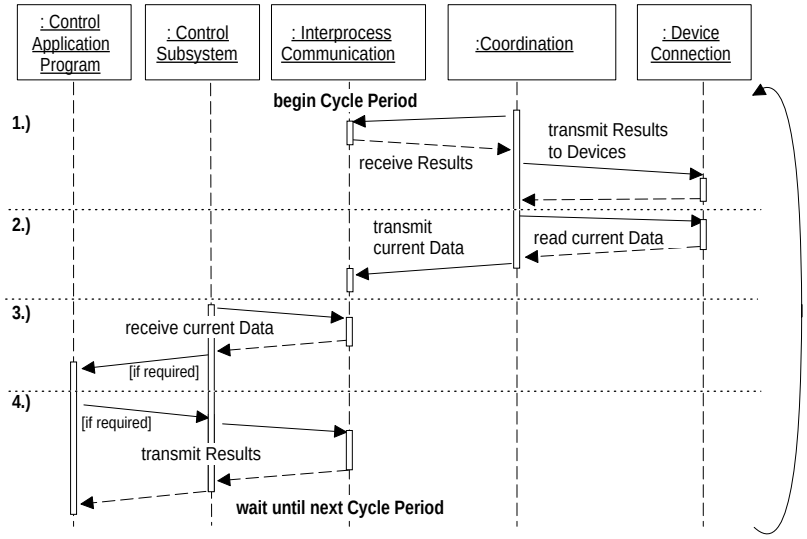


Abbildung 3.11: Szenario des Datenaustauschs zwischen den Steuerungssubsystemen (bzw. Klassen) und den gesteuerten Geräten

3. Lesen der Geräteinformationen durch eine Steuerungsapplikation
   Die *Control Subsystems* lesen aus *Interprocess Communication* den aktuellen Zustand der Geräte ein und verarbeiten die Daten. Bei Bedarf geben sie die Daten an die vom Anwender geschriebenen *Control Application Programs* weiter.

   Ebenso wie die Klasse *Coordination* liest das *User Interface* die aktuellen Informationen aus der Klasse *Interprocess Communication* und zeigt sie daraufhin dem Benutzer an.

4. Zurückschreiben der Ergebnisse der *Control Subsystems*
   Die Ergebnisse der Kontrollfunktionalität (z.B. neue Zielpunkte für einen Roboterarm, Ausgangszustände einer E/A Baugruppe oder auch Befehle an *Coordination*) werden in *Interprocess Communication* abgelegt.

   Neben dem beschriebenen Ablauf besteht die Möglichkeit, daß ein *Control Application Program* Befehle direkt weiterleitet, d.h. die *Control Subsystems* umgeht.

   Das gleiche gilt für die vom Benutzer im *User Interface* eingegebenen Befehle.

3.3 Design

In diesem Kapitel wird zunächst der Aufbau der Subsysteme vorgestellt. Danach folgt eine Beschreibung der einzelnen Subsysteme, bzw. Packages *Interprocess Communication*, *Communication and Coordination*, *Device Connection*, *Control Packages* und *User Interface*. Den Abschluß bildet eine Einteilung der Klassen des Steuerungssystems in einzelne Tasks.

3.3.1 Architektur der Subsysteme

Die Architektur des Robotersteuerungssystems ist in hierarchischen Schichten aufgebaut [KGSL97a]. Dies ist typisch für objektorientierte Echtzeitsysteme [SGW94]. Die in dieser Arbeit vorgestellte Schichtung entspricht der Interpretation von B. P. Douglass [Dou98b] des *Microkernel* Architekturmusters.
Die Modellierung des Steuerungssystems in Schichten hat folgende Vorteile:

- Die Einteilung in Schichten gibt implizit die Grobstruktur des Systems wieder.
- Das Design der einzelnen Schichten kann unabhängig voneinander entwickelt werden.
- Die einzelnen Schichten können ausgetauscht und wiederverwendet werden.

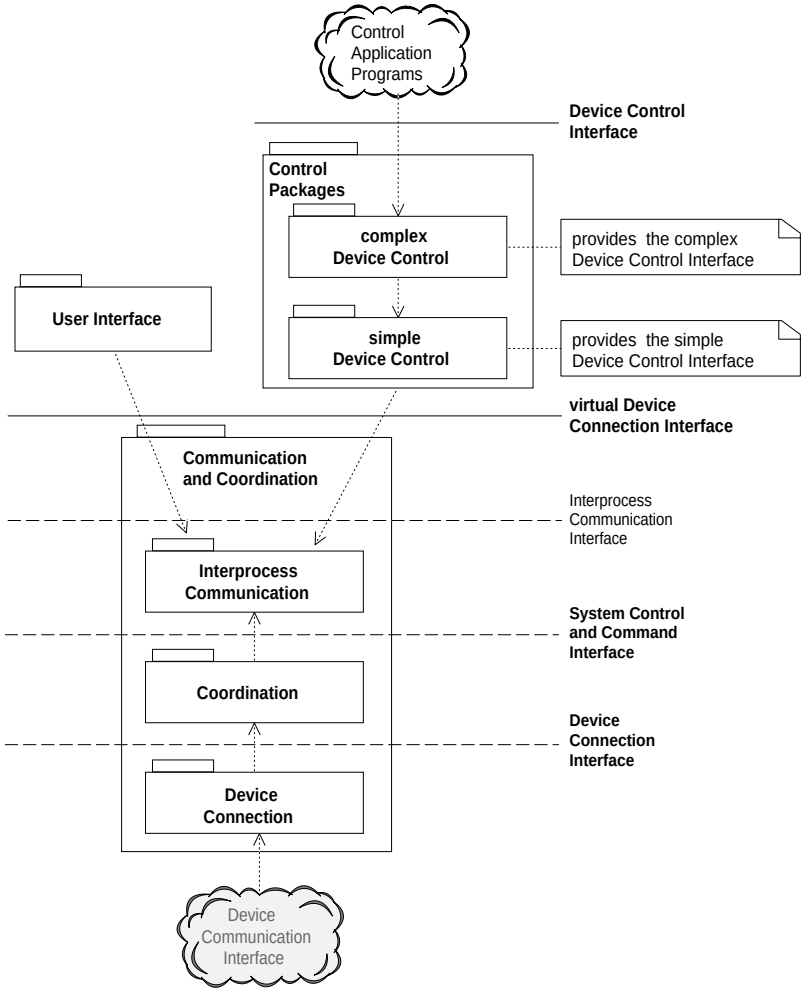


Abbildung 3.12: Einteilung in Packages (Subsysteme)

Die Schichten (siehe Abbildung 3.12) bearbeiten jeweils eine logisch zusammengehörige Aufgabenstellung. Diese Ebenen der Steuerung werden im Design als Packages dargestellt. Subsysteme nach [RBP+91, CY91a, Boo94] entsprechen weitestgehend den Packages in UML [BRI99, RIB99]. Allerdings sind UML Packages stärker auf das Design und die Implementierung von Systemen ausgerichtet [BRI99]. Die hier im Design verwendeten Packages entsprechen nicht dem Konzept der Java Packages, da die Implementierung in C++ erfolgt. Die Packages des Entwurfs und der Implementierung leiten sich aus den Subsystemen der Analyse ab (siehe Abschnitt 3.2.4.1).
Zwischen den einzelnen Packages ergeben sich folgende Schnittstellen:

1. *Device Control Interface* der *Control Packages*
   Diese Schnittstelle besteht aus zwei Schichten, die von zwei Arten von Packages (*complex Device Control* Packages und *simple Device Control* Packages) realisiert werden [2] :

   (a) *Complex Device Control Interface*
   Dieser Teil der Schnittstelle ermöglicht den Zugriff auf komplexe Geräte und wird durch die komplexen Steuerungs-Packages (*complex Device Control*) implementiert. Die Steuerungsapplikationen (*Control Application Programs*) greifen über Aggregationsbeziehungen auf die *complex Device Control* Packages zu.

   (b) *Simple Device Control Interface*
   Durch diese Schnittstelle können die einfachen Geräte angesteuert werden. Zur Erfüllung ihrer Steuerungsaufgaben benutzen die komplexen Steuerungs-Packages die einfachen Packages. Dies wird durch die Aggregationsbeziehung zwischen den Packages ausgedrückt.

   Denkbar ist auch, daß Steuerungsapplikationen zur Ansteuerung einfacher Geräte direkt auf die entsprechenden *simple Device Control* Packages zugreifen (diese Möglichkeit ist in Abbildung 3.12 nicht dargestellt).

   Die beiden Package-Typen, die diese beiden Schnittstellen realisieren, spiegeln die Gliederung der Systemkomponenten (siehe Anordnung der Geräte in Abschnitt 3.1.3) wider.

2. Schnittstellen des *Communication and Coordination Package*
   Dieses Package besteht intern aus drei Schichten von Packages. Die Schnittstelle des obersten Package (*Interprocess Communication Interface*) stellt zugleich die Schnittstelle des gesamten *Communication and Coordination Package* dar (entsprechend dem *Facade* Entwurfsmuster aus [GHJV94]).

   (a) *Virtual Device Connection Interface*
   Diese Schnittstelle bietet den konkurrenten *Application Tasks* (siehe Beschreibung der Tasks in Abschnitt 3.3.7.1) und der *User Interface Task* eine virtuelle Zugriffsmöglichkeit auf die realen Geräte, d.h. die *Application Tasks* verhalten sich so, als hätten sie den exklusiven Zugriff über die *Device Connection*.

[2] Die Beziehungen zwischen den Packages in Abbildung 3.12 werden durch Aggregationen von Klassen realisiert. UML definiert zwischen Packages keine Aggregationsbeziehungen sondern nur Abhängigkeiten [Dou98b] oder *Import*-Beziehungen [BRI99]. Die dennoch in dieser Arbeit erwähnten Aggregationen zwischen Packages signalisieren, daß es sich bei diesen Beziehungen tatsächlich um Aggregationen zwischen Klassen handelt und nicht um Assoziationen, die ebenfalls als Abhängigkeiten oder *Import*-Beziehungen zwischen Packages modelliert werden.

Über das *Virtual Device Connection Interface* werden alle Daten zwischen den *Application Tasks* zur Gerätesteuerung und den Geräten ausgetauscht.

(b) *System Control and Command Interface*
Diese Schnittstelle stellt eine logische Trennung zwischen den Packages (mit unterschiedlicher Funktionalität) des Robotersteuerungssystems dar. Während das *Interprocess Communication* Package die Datenkommunikation zwischen den *Application Tasks* übernimmt, beinhaltet das *Coordination* Package die zentralen Funktionen wie die Synchronisation und Koordination der *Application Tasks* und steuert dadurch implizit deren Kommunikation. Zur Bearbeitung dieser Aufgaben benutzt (in Form einer Aggregationsbeziehung) das *Coordination* Package die Packages *Interprocess Communication* und die *Device Connection*.

(c) *Device Connection Interface*
Da die *Device Connection* von der verwendeten Kommunikationsschnittstelle zu den Geräten (z.B. Ansteuerung der Geräte über einen Feldbus) abhängt, wird die *Device Connection* als eigenständiges Package modelliert.

In den folgenden Abschnitten werden die einzelnen Packages ausführlicher vorgestellt.

3.3.2 Interprocess Communication

Das *Interprocess Communication* Package ermöglicht den Austausch von Daten zwischen einzelnen Prozessen (oder Threads) und bietet Mechanismen zur Synchronisation von Tasks. Zunächst werden die statischen Zusammenhänge zwischen den Klassen vorgestellt, anschließend folgt eine exemplarische Darstellung des dynamischen Verhaltens.

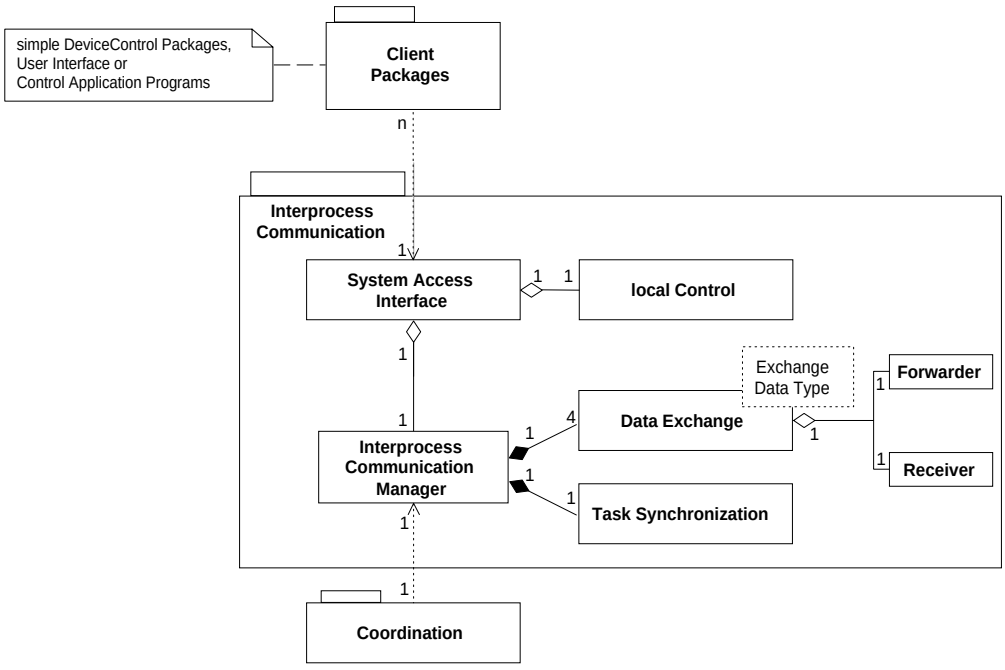


Abbildung 3.13: Klassendiagramm des *Interprocess Communication* Package

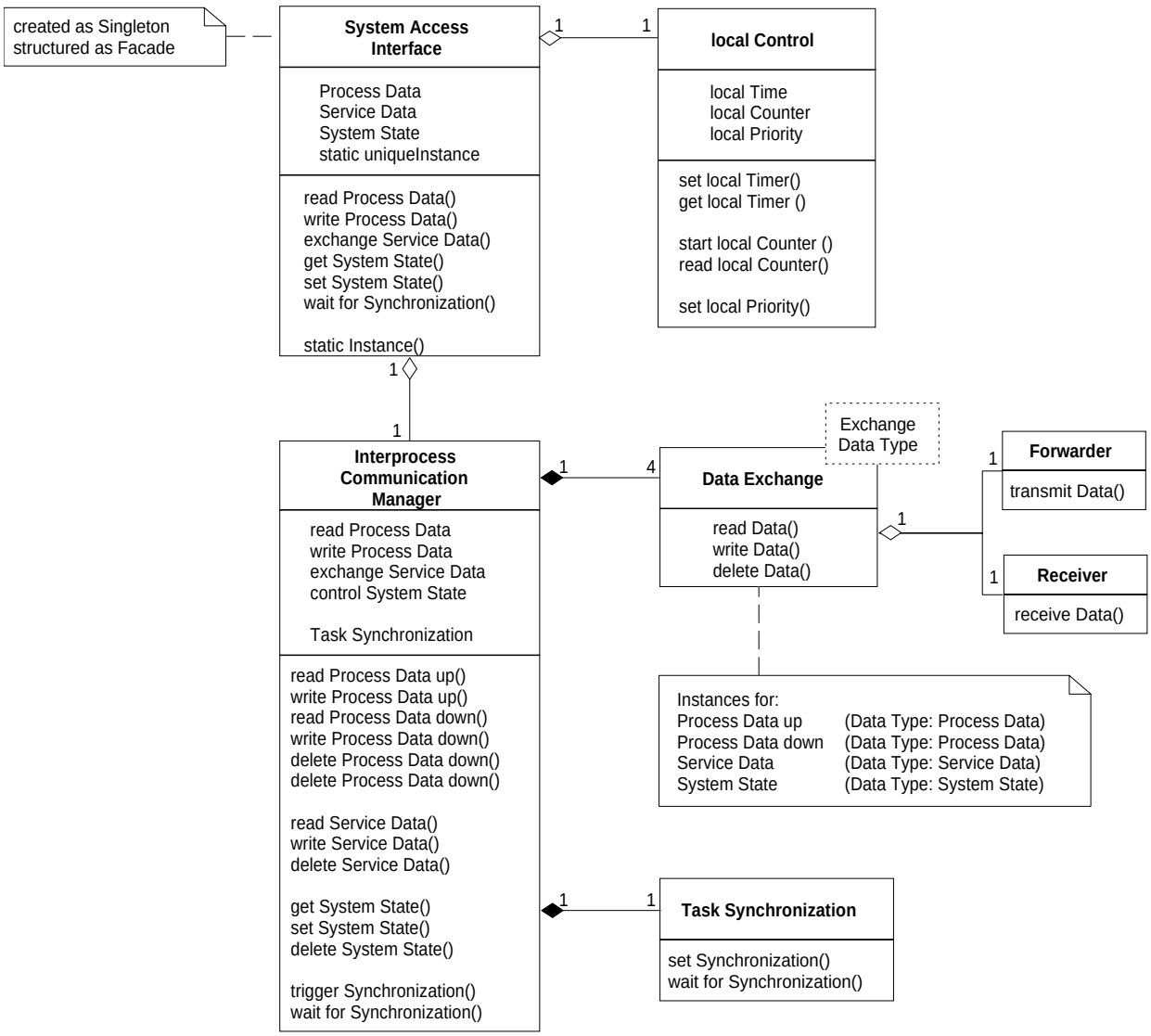


Abbildung 3.14: Detailiertes Klassendiagramm des *Interprocess Communication* Package

3.3.2.1 Statischer Aufbau

Das *Interprocess Communication* Package besteht aus folgenden Klassen (siehe Abbildung 3.13):

- *System Access Interface*
  Diese Klasse stellt die Zugriffsschnittstelle (Proxy und Observer nach [GHJV94]) für die *Client Packages* (d.h. die *simple Device Control* Packages und das *User Interface*) dar. Die *Control Application Programs* können über diese Schnittstelle ebenfalls mit den Geräten kommunizieren.

  Diese Schnittstelle wird als Klasse modelliert, die die Zugriffe auf die weiteren Klassen (*Interprocess Communication Manager* und *local Control*) zentral entgegennimmt und weiterleitet. Damit realisiert das *System Access Interface* das Entwurfsmuster *Facade* aus [GHJV94].

  Eine Entwurfsalternative wäre die Verwendung einer abstrakten Klasse als Schnittstelle, die die Methoden für den Zugriff auf die Klassen des Packages definiert. Da jedoch *Interprocess Communication* aus verschiedenen Klassen besteht, deren Methoden genutzt werden sollen, bietet das Entwurfsmuster *Facade* die Möglichkeit, den Aufruf

der einzelnen Methoden aus den Klassen zu koordinieren. Dadurch bleibt die genaue Bearbeitung einer Anfrage von außen transparent.

Zur Gewährleistung einer sicheren Kommunikation dürfen die Klassen des *Interprocess Communication* Package nur einmal in einer *Application Task* instanziiert werden. Dies wird dadurch erreicht, daß die Klasse *System Access Interface* zusätzlich als *Singleton* (d.h. entsprechend dem *Singleton* Entwurfsmuster nach [GHJV94]) modelliert wird. Da dieses Muster nicht auf abstrakte Klassen angewendet werden kann, ist dies ein weiterer Grund für die Verwendung des *Facade* Entwurfsmusters.

- *local Control*
  Diese Klasse enthält Methoden, mit denen die *Application Tasks* auf das Betriebssystem zugreift. Dies sind zum Beispiel: lokale Zähler und Zeitgeber sowie die Prioritätsvergabe. Betriebssystemzugriffe zur Kommunikation und Synchronisation mit anderen Klassen sind nicht Teil von *local Control*.

- *Interprocess Communication Manager*
  Die Klasse *Interprocess Communication Manager* stellt die eigentlichen Methoden zur Interprozeßkommunikation und -synchronisation zur Verfügung. Sie nutzt dazu die Klassen *Data Exchange* und *Task Synchronization* (Komposition nach UML [Dou98b, BRI99]). Grundsätzlich ist eine Auslagerung der Funktionalität in weitere Klassen nicht notwendig. Allerdings ergibt sich durch diese Komposition der Vorteil, daß die implementierungsabhängigen Betriebssystemaufrufe (in *Data Exchange* und *Task Synchronization*) von den logischen Methoden in *Interprocess Communication Manager* getrennt werden. Dadurch können die Betriebssystemaufrufe geändert werden, ohne daß die Klasse *Interprocess Communication Manager* beeinflußt wird.

- *Data Exchange*
  Data Exchange übernimmt den Datenaustausch zwischen den Tasks (der Datenaustausch wird durch Shared Memory realisiert, siehe Abschnitt 6.1). Die Klasse ist parametrisierbar. Der Datentyp (*Exchange Data Type*), der als innerer Datentyp bei Instanziierung der Template Klasse angegeben wird, bestimmt die Struktur der Daten, die ausgetauscht werden. Dieser Austausch geschieht entsprechend des *Network* Entwurfsmusters aus [BMR+96] [3]. Die beiden Klassen des Musters, *Forwarder* und *Receiver*, kapseln den Mechanismus zum Datenaustausch. Die Echtzeitaspekte von *Network* werden in [Sel96] näher untersucht.

  Der *Interprocess Communication Manager* hält sich vier Instanzen von *Data Exchange*, die den benötigten Datenaustauschkanälen (jeweils ein ein- und ein ausgehender Prozeßdatenkanal sowie ein Kanal jeweils für die Servicedaten und die Systemzustände) entsprechen (siehe CANopen Kommunikationsmodell [CAN95a, CAN95b, CAN97] in Abschnitt 2.4.6.2.3).

- *Task Synchronization*
  Diese Klasse beinhaltet Methoden zur Synchronisation der Prozeßabläufe.

Das *Coordination* Package greift direkt auf die Klasse *Interprocess Communication Manager* zu (siehe Abhängigkeitsbeziehung in Abbildung 3.13) und nutzt ausschließlich deren Methoden.
Da immer genau eine Instanz von *Coordination Manager* mit einer Instanz von *Interprocess Communication Manager* gekoppelt ist, wird auch keine *Singleton* benötigt.

[3] Eine ähnliche Beschreibung des funktionalen Ablaufs findet sich in [BS69].

Eine Darstellung der wichtigsten Methoden und Daten findet sich im Kassendiagramm in Abbildung 3.14.

3.3.2.2 Dynamisches Verhalten

Das *Interprocess Communication* Package bietet den anderen Packages seine Kommunikations- und Synchronisationsdienste an. Die Aktionen der Objekte des Package werden stets von dem *Coordination* Package und den *Control Packages* bzw. dem *User Interface* ausgelöst. Die Interprozeßkommunikation verhält sich passiv.

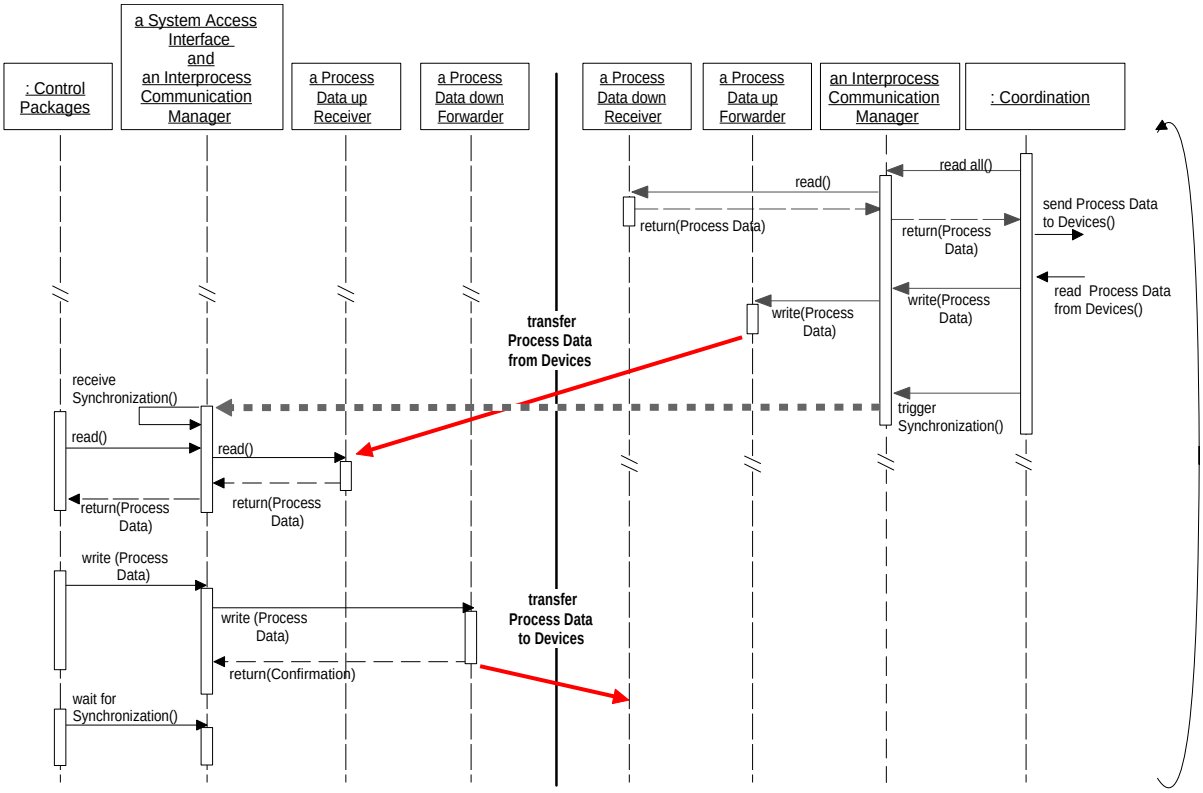


Abbildung 3.15: Szenario des Prozeßdatenaustauschs

Das Sequenzdiagramm in Abbildung 3.15 zeigt das Szenario für den Austausch von Prozeßdaten innerhalb eines Kommunikationszyklus.
Der Ablauf wird vom *Coordination* Package angestoßen. Dieses liest zunächst die Prozeßdaten über die Klasse *Interprocess Communication Manager* aus *Process Data down Receiver* [4] und gibt sie über die *Device Connection* an die Geräte weiter. Im Gegenzug gibt *Coordination* die aktuellen Daten von den Geräten an *Process Data up Forwarder* weiter. Darauf aktiviert das *Coordination* Package die *Application Tasks*. Diese bearbeiten die Gerätedaten, die sie aus der Klasse *Process Data up Receiver* erhält, und gibt die daraus resultierenden Anweisungen an *Process Data down Forwarder* zurück. An dem Datentransfer sind die Klassen *System Access Interface* und *Interprocess Communication Manager* ebenfalls beteiligt.
Die Klassen *Process Data down Receiver*, *Process Data up Forwarder*, *Process Data up Receiver* und *Process Data down Forwarder* sind jeweils als Datenspeicher angelegt (diese können

[4] Die Klassen *Process Data down Receiver*, *Process Data up Forwarder*, *Process Data up Receiver* und *Process Data down Forwarder* leiten sich aus dem Entwurfsmuster *Network* ab.

beispielsweise als Shared Memory oder Nachrichtenspeicher implementiert werden, siehe Implementierung in Abschnitt 6.1.1). Die *Control Packages* können über die Klassen *System Access Interface* und *Interprocess Communication Manager* nur lesend und hinzufügend auf diese Datenspeicher zugreifen. Gelöscht werden dürfen die Daten in den Speichern nur durch die Klassen des *Coordination* Package.
Der Austausch von Servicedaten und des Systemstatus, bzw. Kommandos an das System, geschieht auf die gleiche Art und Weise.

3.3.3 Coordination

Dieser Abschnitt beschreibt die statische Architektur und das Verhalten des Package.

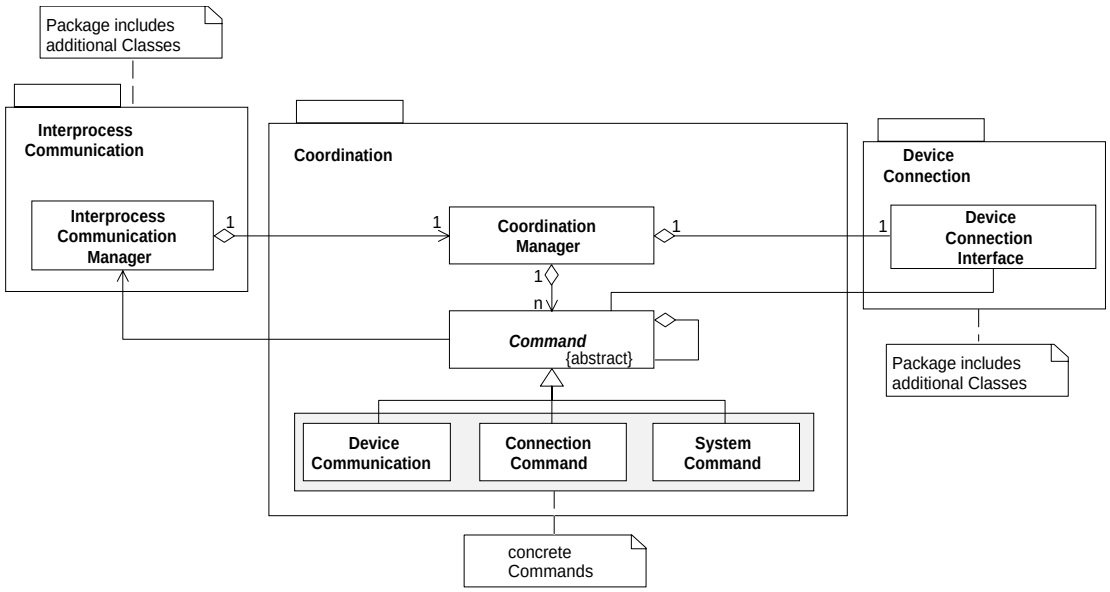


Abbildung 3.16: Klassendiagramm des *Coordination* Package

3.3.3.1 Statischer Aufbau

Zwei Arten von Klassen bestimmen das *Coordination* Package: die Klasse *Coordination Manager* und die Klassen des *Command* Entwurfsmusters [GHJV94]. Der Zugriff auf die Klassen des Musters geschieht über die abstrakte Oberklasse *Command*. Die konkreten Kommandos werden von *Device Communication*, *Connection Command* und *System Command* realisiert (konkrete Kommandoklassen). Die abstrakte Klasse *Command* ermöglicht, das System um zusätzliche konkrete Kommandoklassen zu erweitern.
In Anlehnung an das Entwurfsmuster *Command* fungiert das Package *Interprocess Communication* hierbei sowohl als Vertreter der Klienten (*Application Tasks* und *User Interface Task*), die Befehle und Aufträge an *Coordination* schicken, als auch als Empfänger der Ergebnisse (*Receiver*). Die Rolle des *Invoker* übernimmt die Klasse *Coordination Manager*. Sie nimmt die Aufträge der Klienten entgegen und reicht sie an die entsprechenden konkreten Kommandoklassen weiter.
Coordination übernimmt folgende Aufgaben:

- Koordination des Systemverhaltens
  Die Klasse *Coordination Manager* steuert das dynamische Verhalten der gesamten

Steuerung dadurch, daß sie den Zyklus des Datenaustauschs bestimmt und die *Application Tasks* synchronisiert.

Zur Ausführung des Datenaustauschs und zur Weiterleitung der Kommandos an die Steuerung benutzt sie die konkreten Kommandoklassen.

- Kommunikation mit den gesteuerten Geräten sowie Bearbeitung der systeminternen Kommandos
  Dazu werden die Klassen *Device Communication*, *Connection Command* und *System Command* mit dem gemeinsamen Interface *Command* verwendet.

3.3.3.2 Dynamisches Verhalten

Da die dynamischen Interaktionen des Subsystems von der Klasse *Coordination Manager* bestimmt werden, entspricht das Zustandsübergangsdiagramm in Abbildung 3.17 dem dynamischen Ablauf dieser Klasse. *Coordination Manager* setzt das in der Analyse in Abschnitt 3.2.4.2 beschriebene Systemverhalten um.

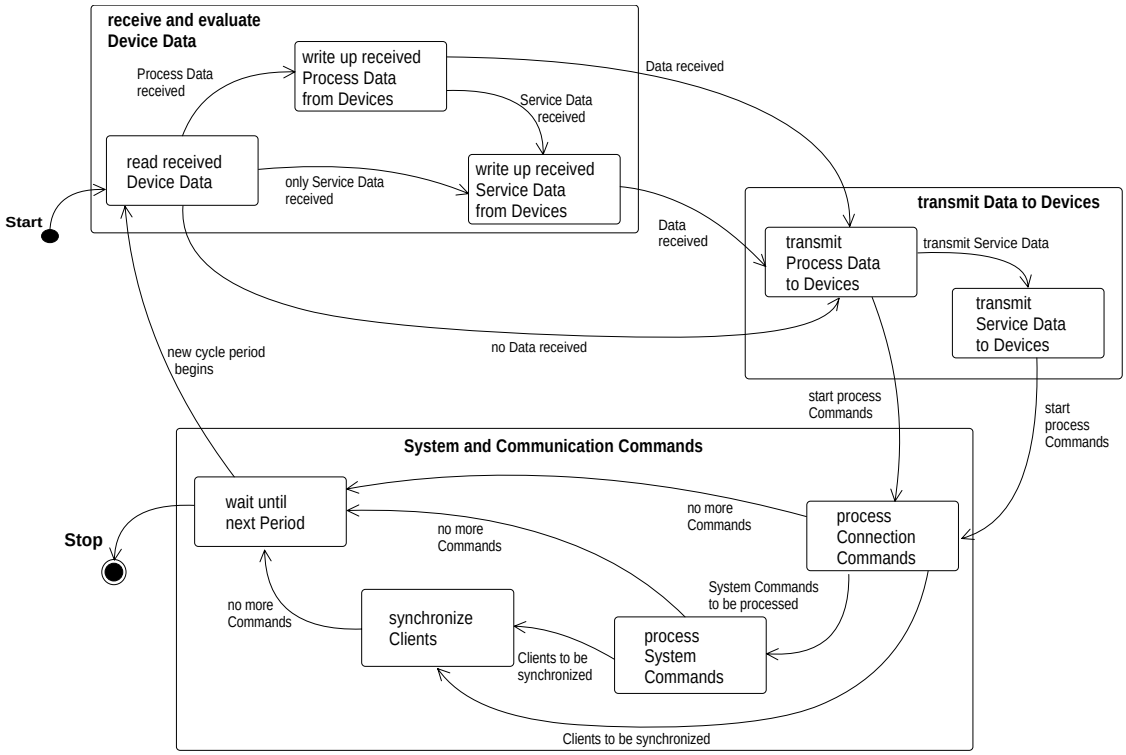


Abbildung 3.17: Zustandsübergangsdiagramm des *Coordination* Package

Nach dem Start des Zyklus werden zunächst im Superzustand *receive and evaluate Device Data* die Daten von Geräten am Feldbus eingelesen und entsprechend des Datentyps getrennt weitergereicht (die möglichen Datentypen sind in Abschnitt 2.4.6.2.3 beschrieben). Die Zustände dieses Superzustands (Superstate) sind:

1. *read received Device Data*
   Alle in der vorangegangenen Periode von den CANopen-Geräten versendeten Telegramme werden gelesen. Dazu wird das Package *Device Connection* aufgerufen.

2. *write up received Process Data from Devices*
   Die zuvor gelesenen Prozeßdaten werden über das *Interprocess Communication* Package an die *Application Tasks* und die *User Interface Task* weitergereicht.

3. *write up received Service Data from Devices*
   Falls Servicedaten empfangen worden sind, werden diese ebenfalls mittels des *Interprocess Communication* Package an die *Application Tasks* und die *User Interface Task* übergeben.

Im zweiten Superzustand *transmit Data to Devices* initiiert die Klasse *Coordination Manager* das Versenden der Daten an die gesteuerten Geräte. Zwei Zustände sind Teil von *transmit Data to Devices*:

1. *transmit Process Data to Devices*
   Alle zu versendenden Prozeßdaten werden vom *Interprocess Communication* Package gelesen und anschließend (mittels *Device Connection*) über den Feldbus versendet.

2. *transmit Service Data to Devices*
   Sollen Servicedaten versendet werden, so geschieht dies in diesem Zustand.

Im letzten Superzustand *System and Communication Commands* werden Befehle ausgeführt, die die Robotersteuerung oder die Kommunikation der Steuerung mit den Geräten betrifft. Die vier Unterzustände sind:

1. *process Communication Commands*
   In diesem Zustand werden die Kommandos, die die Gerätekommunikation betreffen, ausgeführt.

2. *process System Commands*
   Nach den Befehlen, die die Kommunikation betreffen, werden die Kommandos an das Robotersteuerungssystem bearbeitet.

3. *synchronize Clients*
   Hier werden die wartenden *Application Tasks* (d.h. die Klienten) synchronisiert, bzw. aktiviert. Dies geschieht mit Hilfe der Klasse *Task Synchronization* des *Interprocess Communication* Package.

4. *wait until next Period*
   Zuletzt wartet *Coordination* bis eine neue Periode beginnt.

Im folgenden sollen die Interaktionen während der Zustände [5] *receive and evaluate Device Data* und *transmit Data to Devices* sowie *System and Communication Commands* beschrieben werden.
Der Datenaustausch mit den Geräten wird entsprechend dem Kollaborationsdiagramm in Abbildung 3.18 jeweils von dem Objekt *aCoordination Manager* initiiert.
Aufgrund des Aufrufs *initiate Read received Data from Devices* durch *aCoordination Manager* liest die Instanz *aDevice Communication* alle empfangenen Daten vom *Interprocess Communication* Package (Zustand *read received Device Data*). Anschließend gibt *aDevice Communication* die Daten getrennt nach Prozeßdaten (Zustand *write up received Process Data from Devices*) und Servicedaten (Zustand *write up received Service Data from Devices*) mittels *Interprocess Communication* an die *Application Tasks* weiter.

[5] In Real-Time UML können Zustände auch Aktionen sein [Dou98b].

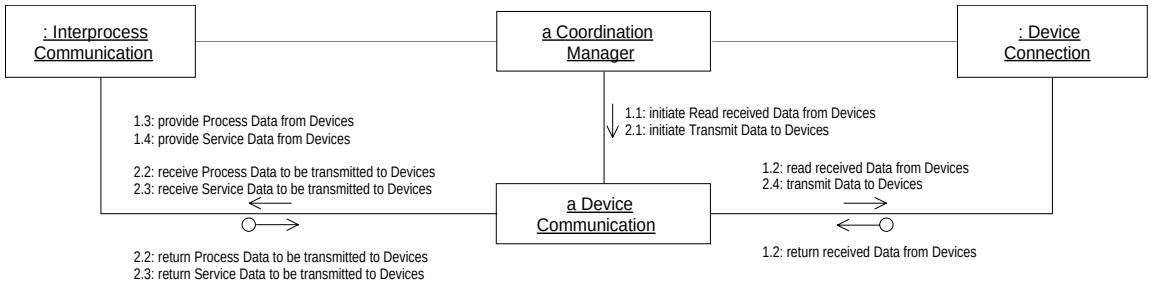


Abbildung 3.18: Kollaborationsdiagramm des Datenaustauschs mit den gesteuerten Geräten

Nach dem Aufruf *initiate Transmit Data to Devices* fordert *aDevice Communication* zuerst die Prozeßdaten und dann die Servicedaten von *Interprocess Communication* (*request Process Data to be transmitted* entsprechend dem Zustand *transmit Process Data to Devices* und *request Service Data to be transmitted* entsprechend dem Zustand *transmit Service Data to Devices*) und versendet diese mit Hilfe von *Device Connection*.

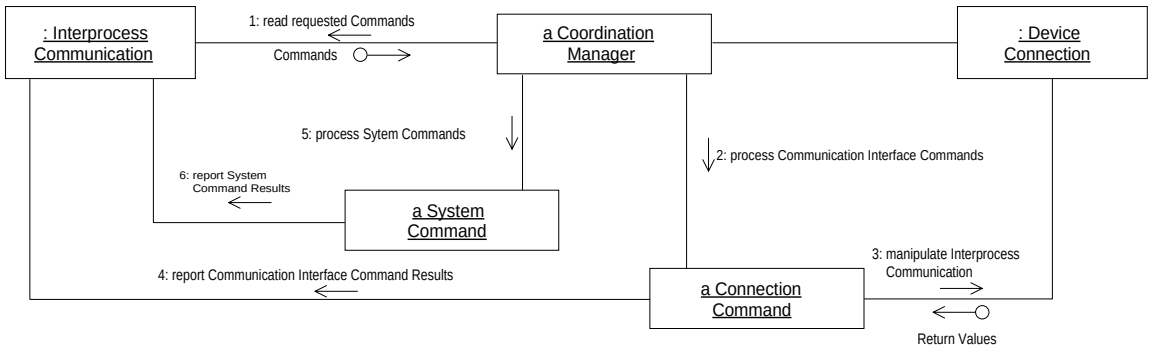


Abbildung 3.19: Kollaborationsdiagramm der Bearbeitung der Kommandos an die Gerätekommunikation und das Steuerungssystem

Zur Bearbeitung der Kommandos an die Kommunikationsschnittstelle mit den Geräten und das Steuerungssystem (siehe Abbildung 3.19) werden zunächst die anstehenden Befehle aus *Interprocess Communication* von dem Objekt *aCoordination Manager* gelesen.
Die Instanz *aCoordination Manager* gibt die Kommandos des Gerätekommunikationsprotokolls an *aConnection Command* weiter, worauf diese den Befehl ausführt (Methode *manipulate Device Connection*) und das Ergebnis der Aktion an das *Interprocess Communication* Package zurückgibt. Diese Aktionen laufen im Zustand *process Communication Commands* ab.
Danach werden die Kommandos an das Steuerungssystem durch das Objekt *aSystem Command* in ähnlicher Weise ausgeführt (Zustand *process System Commands*).

3.3.4 Device Connection

Die *Device Connection* ermöglicht den Datenaustausch mit den angesteuerten Geräten. Ein objektorientiertes Modell zur Kommunikation zwischen Geräten und Steuerungssystemen über einen Feldbus wird in [KGSL97b] vorgestellt.
Im Entwurf der universellen Steuerung wird die Aufgabe der *Device Connection* von einer einzigen Klasse (*Device Connection Interface*) übernommen. Das später entwickelte Framework

(siehe Abschnitt 4.2) und das komponentenbasierte Steuerungssystem (siehe Abschnitt 4.3) verwenden mehrere Klassen. Die Klasse *Device Connection Interface* interagiert direkt mit dem Treiber der Kommunikationsschnittstelle. In dem hier beschriebenen Steuerungssystem wird über den Feldbus CAN unter Verwendung des CANopen Protokolls kommuniziert (siehe Gliederung der Systemkomponenten in Abschnitt 3.1.3).
Die Funktionen von *Device Connection Interface* sind: Lesen der Daten von den Geräten und Schreiben von Daten an die Geräte. Zusätzlich werden spezielle Funktionen des Feldbusprotokolls durch diese Klasse unterstützt. Dabei interagiert das *Device Connection Interface* direkt mit dem Treiber der Kommunikations-Hardware.

3.3.5 Control Packages

Die *Control Packages* stellen dem Benutzer Methoden zur Steuerung von Automatisierungsgeräten zur Verfügung. Alle Packages und Klassen werden in dem Package *Control Packages* zusammengefaßt (siehe Abbildung 3.20). Die Packages des *Control Package* werden in zwei Schichten, eingeteilt: *complex Device Control* und *simple Device Control*. (Die Funktionalität dieser Packages ist in Abschnitt 3.3.1 beschrieben.)

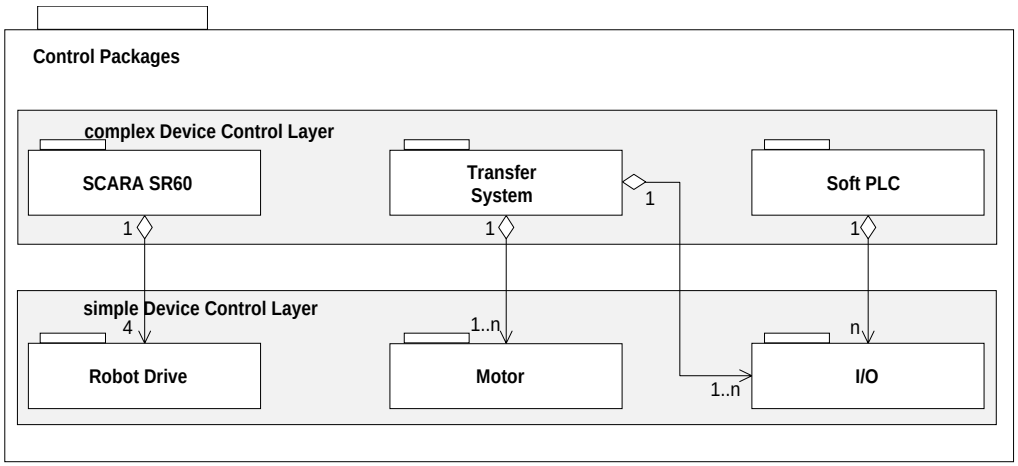


Abbildung 3.20: Übersicht über die *Control Packages*

Grundsätzlich gibt es zwei Varianten zur Einbindung der *Control Packages* in Applikationsprogramme (siehe Abbildung 3.21):

1. Die Applikationsprogramme greifen indirekt über die *Control Packages* auf die Interprozeßkommunikationsschnittstelle zu.
   Der Anwender bindet die Klassen der *Control Packages* in das von ihm geschriebene Applikationsprogramm ein. Die *Control Packages* erzeugen pro Applikationsprogramm jeweils genau eine Instanz der Klassen des *Interprocess Communication* Package. Dies wird durch die Anwendung des Singleton Entwurfsmusters bei der Interprozeßkommunikation erreicht (siehe Abschnitt 3.3.2.1).

2. Die Applikationsprogramme binden die Interprozeßkommunikationsschnittstelle direkt ein und geben die Referenz auf diese Schnittstelle an die *Control Packages* weiter.
   Durch die Einbindung einer Instanz der Klassen von *Interprocess Communication* in die Applikationsprogramme kann von diesen Programmen unmittelbar auf die Schnittstelle zugegriffen werden, d.h. von den Anwendungsprogrammen kann direkt auf die

universelle Steuerung und die gesteuerten Geräte zugegriffen werden. Dies ist insbesondere dann sinnvoll, wenn neue Gerätetypen angesprochen werden sollen, für die von den existierenden *Control Packages* keine Methoden zur Steuerung angeboten werden. Dadurch wird die Inbetriebnahme neuer Automatisierungsgeräte und die Entwicklung von *Control Packages* für diese Geräte besonders unterstützt.

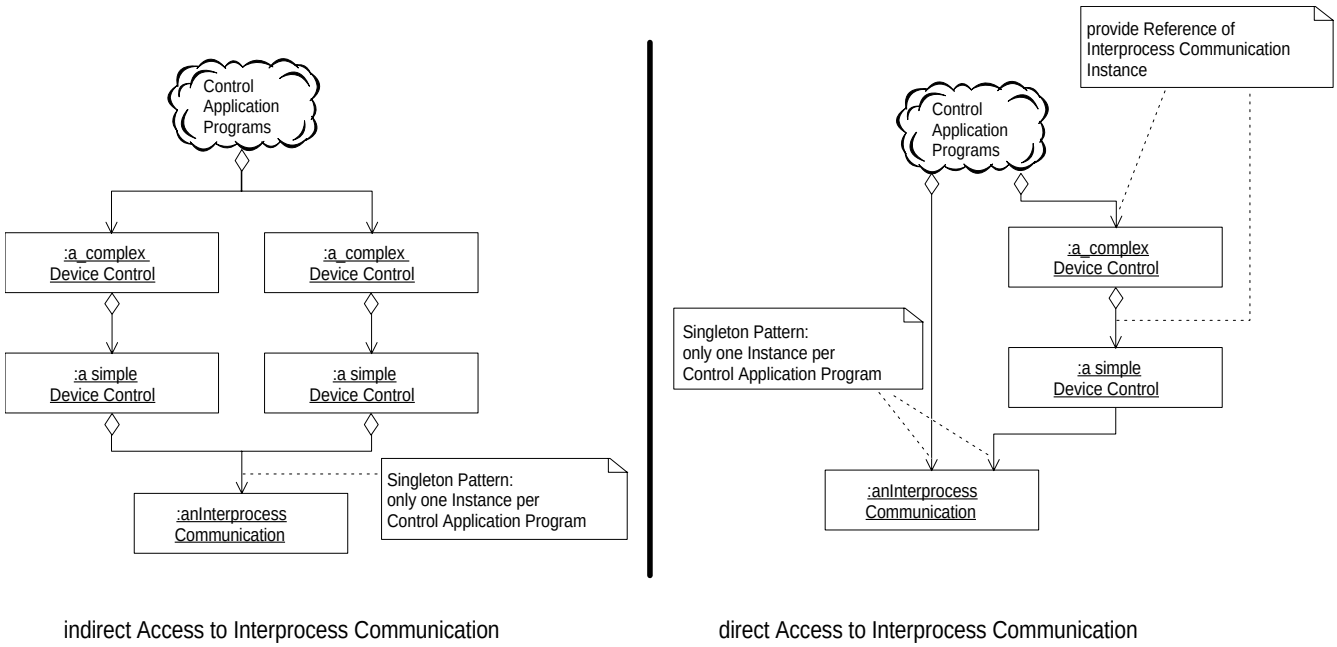


Abbildung 3.21: Einbindung der *Control Packages*

Im folgenden werden die Packages und Klassen der Steuerung eines SCARA Roboters sowie eines Transfersystems und einer speicherprogrammierbaren Steuerung beschrieben. Dabei werden sowohl Klassen der *complex Device Control* als auch *simple Device Control* Klassen in ihrer Zusammenwirkung dargestellt.

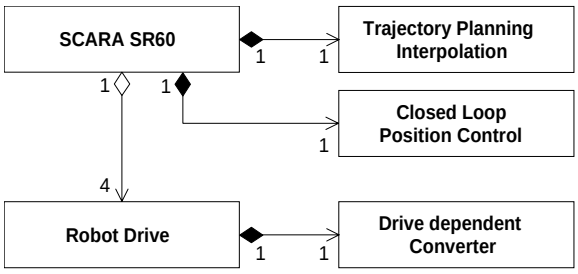


Abbildung 3.22: Klassen zur Steuerung eines SCARA Roboterarms

3.3.5.1 Steuerung eines SCARA Roboterarms

In den folgenden beiden Abschnitt werden die Klassen und deren Verhalten zur Kontrolle eines SCARA Roboters vorgestellt. Diese Architektur kann allerdings auch als Muster

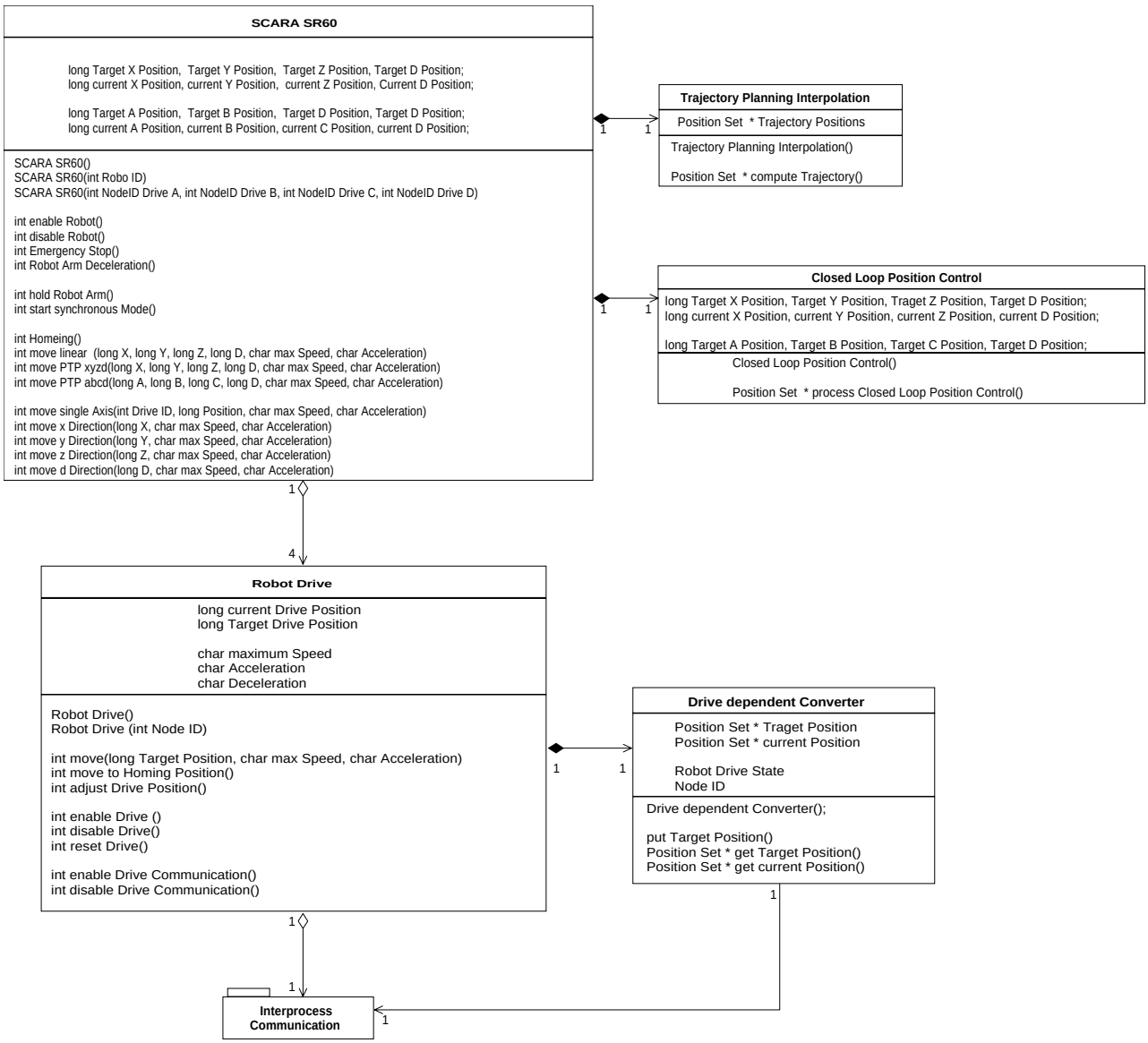


Abbildung 3.23: Klassen zur Steuerung eines SCARA Roboterarms

für die Implementierung von Steuerungen für andere Roboterarmtypen verwendet werden, da die grundsätzlichen Vorgehensweisen bei der Steuerung von Roboterkinematiken immer gleich sind. Vom Typ des Arms sind lediglich die Steuerungsalgorithmen abhängig. Diese Algorithmen sind in spezielle Klassen gekapselt.

3.3.5.1.1 Klassenaufbau

Die in Abbildung 3.20 beschriebenen Klassen *SCARA SR60* und *Robot Drive* werden um die Klassen *Trajectory Planning and Interpolation*, *Closed Loop Position Control* und *Drive dependent Converter* ergänzt. Die Klassen *Trajectory Planning and Interpolation* und *Closed Loop Position Control* beinhalten die Algorithmen zur Kontrolle des Roboterarms.
Die Anordnung dieser Klassen findet sich in Abbildung 3.22.
Die Klasse *SCARA SR60* bildet die Schnittstelle zum Zugriff auf Steuerungsfunktionalität für den Roboterarm. Zwei Arten von Servosteuerungen sind realisiert: Punkt-zu-Punkt Steuerung und kontinuierliche Bahnsteuerung (siehe Abschnitt 2.4.2.1).
Zur Ansteuerung des Roboters beinhaltet *SCARA SR60* folgende Methoden:

- Bewegung des Roboterarms:
  Der Roboter kann linear oder PTP verfahren werden. Dabei kann jeweils der komplette

Arm bewegt werden (d.h. gleichzeitige Bewegung mehrerer Achsen). Die Bewegungsmethoden sind: `int move_linear()`, `int move_PTP_xyzd()` sowie `int move_PTP_abcd()`.

Diese allgemeinen Funktionen sind die Basis der Methoden zum Verfahren einzelner Achsen, bzw. für das Fahren in eine Richtung: `int move_single_Axis()`, `int move_x_Direction()`, `int move_y_Direction()`, `int move_z_Direction()` und `int move_d_Direction()`.

Weiter bietet *SCARA SR60* die Möglichkeit, den Roboterarm in Referenzposition zu bringen. Dazu wird die Methode `int Homing()` verwendet (die Achsen werden einzeln in Referenzposition gefahren; dazu wird die Achse als Parameter übergeben).

- Aktivieren und Deaktivieren des Roboterarms:
  Die Methoden zum Ein- und Ausschalten des Roboters (d.h. der Roboterantriebe) sind: `int enable_Robot()` und `int disable_Robot()`.

  Ebenso kann der Roboter sanft angehalten (`int Robot_Arm_decelerate()`) sowie im Notfall sofort gestoppt werden (`int Emergency_Stop()`).

- Kontrolliertes Anhalten des Roboters:
  Durch die Methode `int hold_Robot_Arm()` wird der Roboter bis zum Eintritt eines bestimmten Ereignisses (Ablauf eines Zeitgebers oder Signal an einem Eingangsmodul, bzw. Sensor) angehalten.

  Während des Wartens unterliegt der Roboterarm weiter der Lageregelung. Eine Änderung der Position des Roboterarms (z.B. durch äußere mechanische Einflüsse) wird erkannt und entsprechend entgegengeregelt [6].

Ein Teil der Methoden wird hauptsächlich von den Applikationsprogrammen aufgerufen. Die anderen Methoden werden sinnvollerweise vom Benutzer interaktiv ausgelöst (der Anwender kann über das *User Interface* auf die Funktionen zugreifen).

- Die interaktiv vom Benutzer verwendeten Methoden sind: das Anschalten und Abschalten des Roboters, das Bewegen der einzelnen Achsen in Referenzposition, das Verfahren der einzelnen Achsen und das Fahren in eine Richtung.

  Das Bewegen der einzelnen Achsen und das Verfahren in eine Richtung wird hauptsächlich für die Roboterprogrammentwicklung nach dem Teach-In-Vorgehen (siehe Abschnitt 2.4.2.4) benötigt.

- Die Methoden zum Verfahren des kompletten Roboterarms oder zum kontrollierten Anhalten werden meist von den Anwendungsprogrammen verwendet.

Zur Bearbeitung der Aufrufe an die Klasse *SCARA SR60* werden die Klassen *Trajectory Planning and Interpolation* (zur Planung der Bewegungsbahn des Roboterarms in einzelnen Inkrementen) und *Closed Loop Position Control* (zur Lageregelung) benötigt. In diesen

[6] Die Lageregelung des Roboterarms wird nicht automatisch mit dem Aktivieren der Roboterantriebe eingeschaltet, sondern erfolgt erst mit dem Start eines Anwendungsprogramms oder beim manuellen Verfahren des Roboters. Bei den heute üblichen Robotersteuerungen werden die Applikationsprogramme (bzw. der Modus zum manuellen Bewegen des Roboterarms) automatisch mit dem Start der Steuerung aktiviert. Dieses Verhalten kann bei der hier beschriebenen Robotersteuerung ebenfalls implementiert werden, hat sich aber bei der konkreten Arbeit mit dem Roboterarm und dessen sich noch im Prototypstadium befindlichen Antrieben nicht als sinnvoll erwiesen.

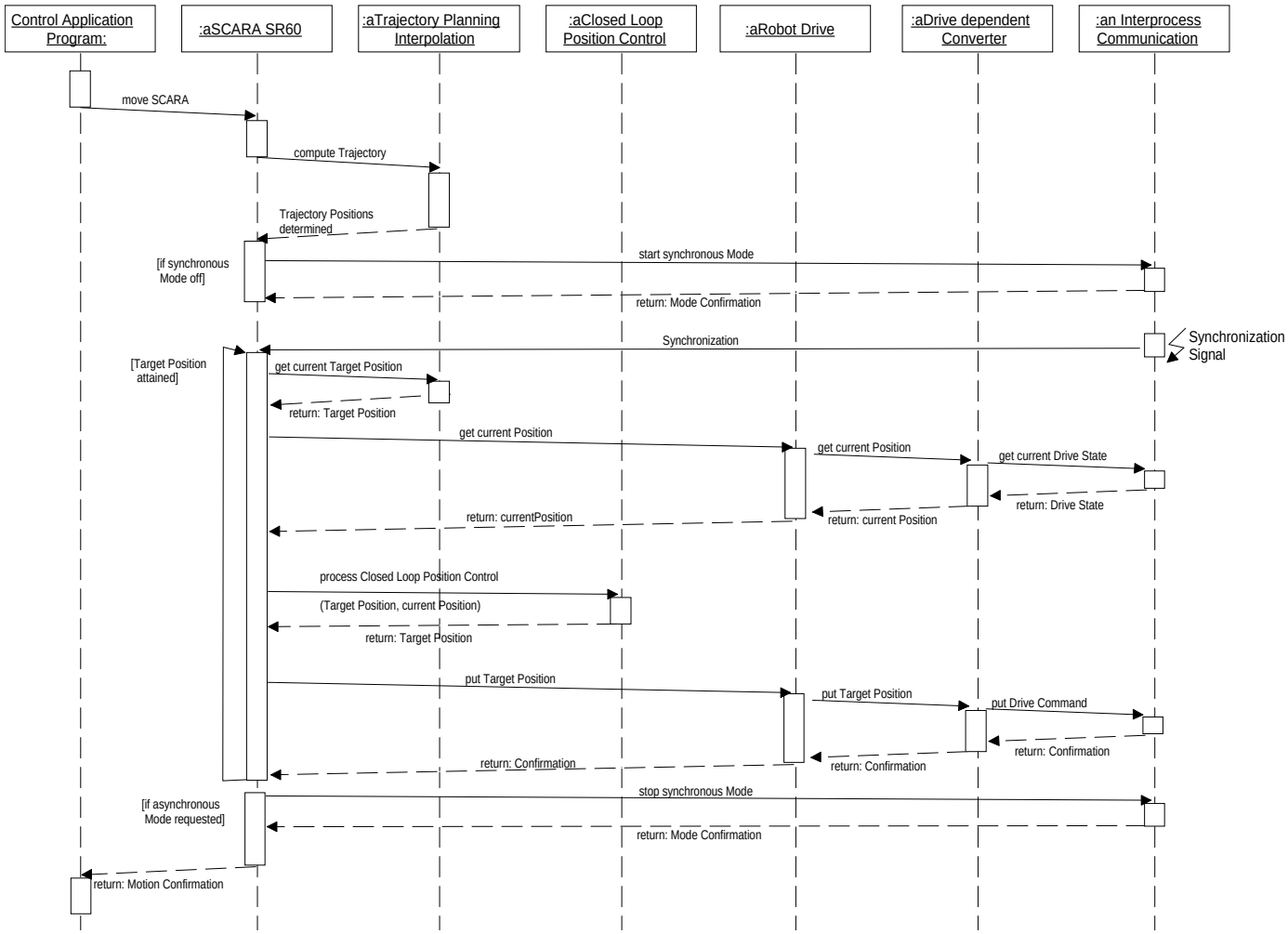


Abbildung 3.24: Bearbeitung eines Aufrufs zum Bewegen des Roboterarms

Klassen werden die Funktionen (Bahnplanung und Lageregelung) gekapselt. Durch diese Separation können die Funktionen leichter ausgetauscht (durch Auswechseln der Klasse) sowie wiederverwendet werden.
SCARA SR60 liest von der Klasse *Robot Drive* die aktuellen Positionen und den Zustand der Roboterantriebe und gibt über diese Klasse neue Zielpositionen und Steuerungsbefehle an die Antriebe weiter. Da der Bosch SCARA Roboterarm SR60 vier Roboterantriebe besitzt (siehe Abschnitt 3.1.3 der Analyse). Zur Umwandlung der gerätespezifischen Daten von den Roboterantrieben benutzt die Klasse *Robot Drive* eine Instanz der Klasse *Drive dependent Converter*. Die Umwandlungsfunktion ist in dieser Klasse gekapselt. Soll ein Roboter mit anderen Antrieben angesteuert werden, so kann diese Klasse ausgetauscht werden.

3.3.5.1.2 Dynamisches Verhalten

Im Sequenzdiagramm in Abbildung 3.24 wird der Ablauf einer Bewegung beschrieben.
Zu Beginn ruft das *Control Application Program* die Instanz von *SCARA SR60* auf. Diese läßt durch *Trajectory Planning and Interpolation* die Roboterbewegung und deren einzelnen Abschnitte berechnen. Darauf wird mit dem Aufruf `int start_synchronous_Mode()` die zeitliche Synchronisation der *Application Task* mit der *Communication and Coordination Task* aktiviert. (Dies muß nur geschehen, sofern die *Application Task* nicht schon synchron

abläuft.)
Anschließend wird die Sequenz zur Kontrolle der Bewegung (entsprechend dem im Abschnitt 2.4.2.1 beschriebenen Vorgehen) bis zum Erreichen der Zielposition zyklisch abgearbeitet:

1. Abfrage des aktuellen Bewegungsinkrements
2. Lesen der aktuellen Position der Roboterantriebe
3. Übergabe des aktuellen Bewegungsinkrements und der aktuellen Position an die Lageregelung und Berechnung der neuen Zielposition für die Roboterantriebe durch die Lageregelung
4. Versenden der neuen Zielpositionen an die Roboterantriebe

3.3.5.2 Speicherprogrammierbare Steuerung

Da Aufgrund der Anforderungen, daß (siehe Abschnitt 1.1) eine speicherprogrammierbare Steuerung Teil der universellen Steuerung sein soll, wird die Funktionalität einer derartigen SPS in Software emuliert. Im Anschluß wird der Aufbau der Klassen sowie deren dynamisches Verhalten beschrieben.

3.3.5.2.1 Klassenaufbau

Die Abarbeitung der SPS-Programme wird von den Klassen *SoftPLC* und *I/O* übernommen (siehe Abschnitt 3.3.5). Zur Kommunikation mit der Prozeßumgebung werden wie bei einer SPS (siehe Abschnitt 2.4.3) Ein- und Ausgabebaugruppen verwendet. Diese Baugruppen (die E/A Module) kommunizieren über den Feldbus CAN mit der Steuerung.
Die Software-Emulation der speicherprogrammierbaren Steuerung umfaßt nicht die in Abschnitt 2.4.3.2 beschriebenen Programmierverfahren (Kontaktplan, Funktionsplan und Anweisungsliste) [7]. Da die zur Beschreibung von SPS-Programmen notwendigen Operationen (binäre Verknüpfungen: z.B. Und, Oder, Invertierung) von C++ unterstützt werden, wird auf diese Operationen zurückgegriffen. Die Funktionen wie Zähler und Zeitgeber (Timer) sind durch Methoden der Klasse *SoftPLC* realisiert.
Im detaillierten Klassendiagramm in Abbildung 3.25 ist zusätzlich zu *SoftPLC* und *I/O* die Klasse *CANopen I/O Module* als Komposition von *I/O* eingeführt. Im einzelnen übernehmen die Klassen folgende Aufgaben:

- *SoftPLC*
  Diese Klasse stellt die Applikationsprogrammierschnittstelle (API) dar.

  Sie bietet Methoden zum Zugriff auf die Ein- und Ausgänge der E/A Module an (Lesen der Ein- und Ausgänge, Schreiben der Ausgänge).

  Neben der für speicherprogrammierbare Steuerungen üblichen zyklischen Abarbeitung von Applikationsprogrammen (Ausführung aller Anweisungen in jedem Zyklus) ermöglicht die Klasse die sequentielle Bearbeitung von Anwendungen (Schrittweise Abarbeitung von einzelnen Anweisungen). Dadurch können SPS-Anweisungen auch in sequentielle Anwendungen (z.B. Roboterprogramme) eingebunden werden.

[7] Durch die Verwendung von Präprozessoren oder graphischen Entwicklungswerkzeugen können auch die Programmierverfahren Kontaktplan, Funktionsplan und Anweisungsliste unterstützt werden. Die Entwicklung derartiger Werkzeuge ist jedoch nicht das Ziel dieser Arbeit.

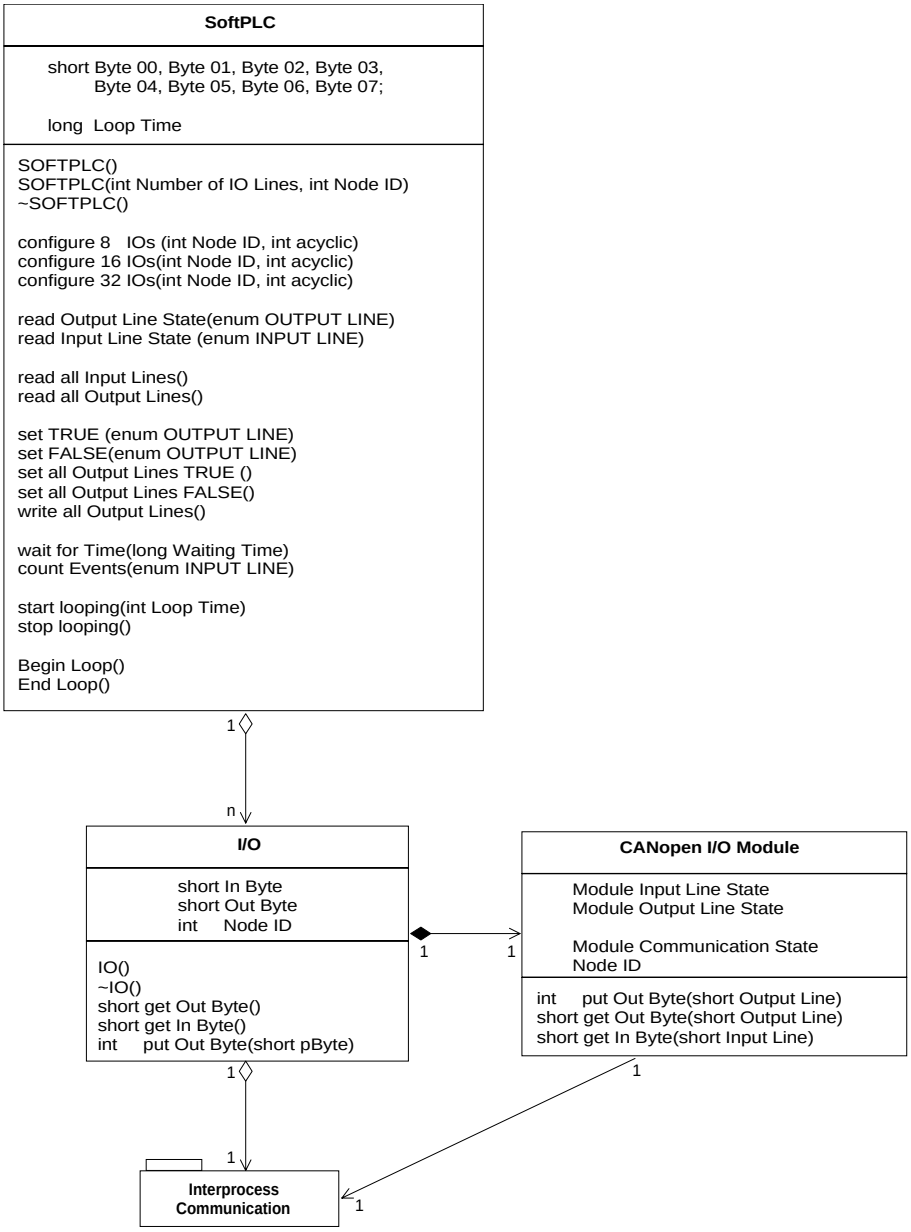


Abbildung 3.25: Klassen der speicherprogrammierbaren Steuerung

Die Klasse *SoftPLC* kann mit E/A Baugruppen mit 8 bis 32 Ein- und Ausgängen kommunizieren. Dies entspricht den zur Zeit angebotenen CANopen E/A Modulen. Bei Bedarf kann die Klasse auch E/A Baugruppen mit mehr Ein- und Ausgängen ansteuern.

- *I/O*
  I/O enthält den Zustand von jeweils acht Eingängen und acht Ausgängen sowie Methoden zum Lesen und Schreiben der Ein- und Ausgänge. Diese entspricht den kleinsten CANopen E/A Modulen. Sollen in *SoftPLC* mehr Ein- und Ausgänge angesteuert werden, so muß eine weitere Instanz der Klasse *I/O* von *SoftPLC* verwendet werden.

- *CANopen I/O Module*
  Diese Klasse realisiert die eigentlichen Datenaustauschfunktionen, die vom Feldbusprotokoll abhängen. *CANopen I/O Module* erhält daher eine Referenz auf *Interprocess*

Communication von *I/O*.

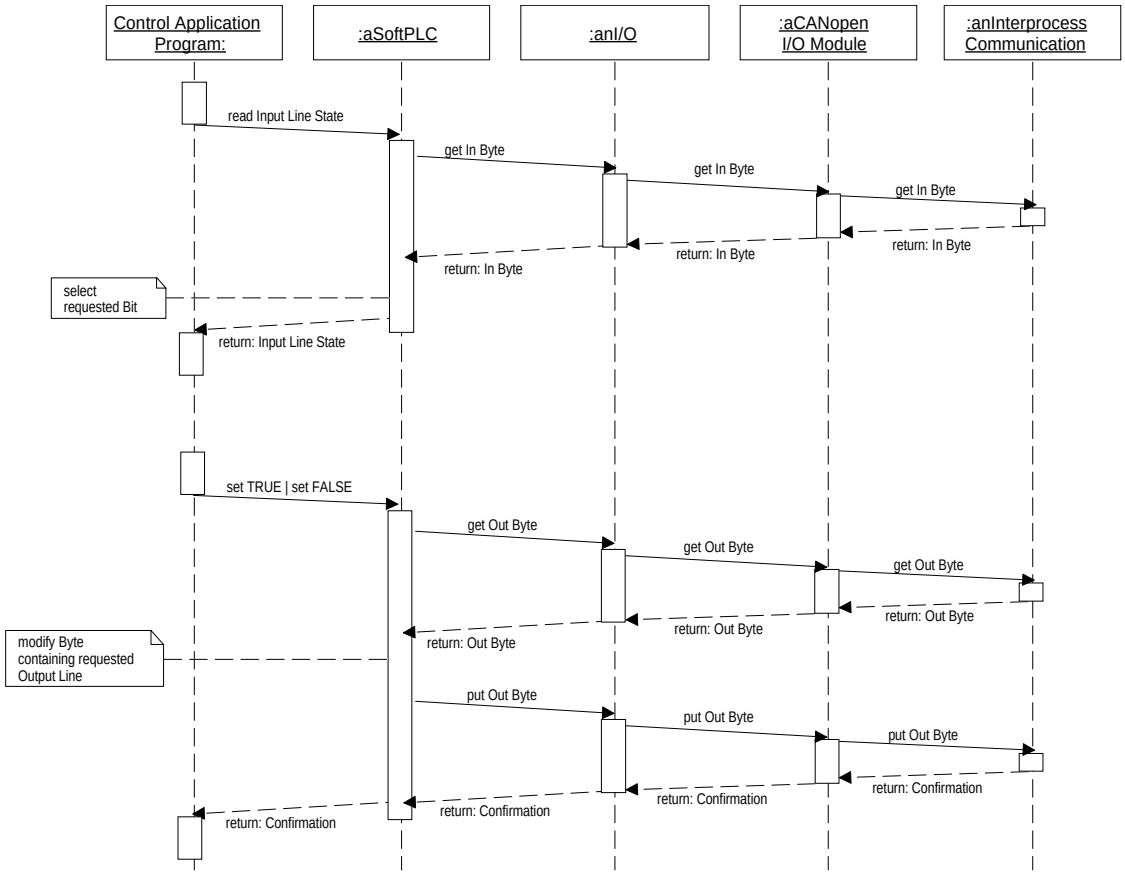


Abbildung 3.26: Datenaustausch bei der sequentiellen Abarbeitung

3.3.5.2.2 Dynamisches Verhalten

Entsprechend der herkömmlichen speicherprogrammierbaren Steuerungen bietet die universelle Steuerung die Möglichkeit, SPS-Programme zyklisch (wie in Abschnitt 2.4.3.2 beschrieben) zu bearbeiten. Zusätzlich können SPS-Anweisungen oder Programmabschnitte jedoch auch ein Roboterprogramme eingebettet werden. Dieses Vorgehen soll als sequentielle Abarbeitung bezeichnet werden und bezieht sich auf die jeweilige Task.

1. Sequentielle Abarbeitung von SPS-Anweisungen
   Bei der sequentiellen Bearbeitung muß vor der Verarbeitung eines bestimmten Eingangswerts einer E/A Baugruppe dieser Wert explizit eingelesen werden. Der Zustand eines einzelnen Eingangs (ein Bit) wird durch *SoftPLC* aus acht Eingangswerten (durch ein Byte repräsentiert) extrahiert.

   Anschließend kann der Wert mit den Operanden von C++ verarbeitet werden.

   Beim Zurückschreiben eines Ausgangswerts an ein E/A Modul werden von der Klasse *SoftPLC* zuerst die vorhergegangenen Ausgangswerte (d.h. acht Ausgangswerte in einem Byte) eingelesen. *SoftPLC* modifiziert den entsprechenden Ausgangswert in dem Ausgangs-Byte und gibt diesen über die Klassen *I/O* und *CANopen I/O Module* an das *Interprocess Communication* Package weiter. Die neuen Ausgangswerte werden dann von der universellen Steuerung unmittelbar an das E/A Gerät weitergegeben.

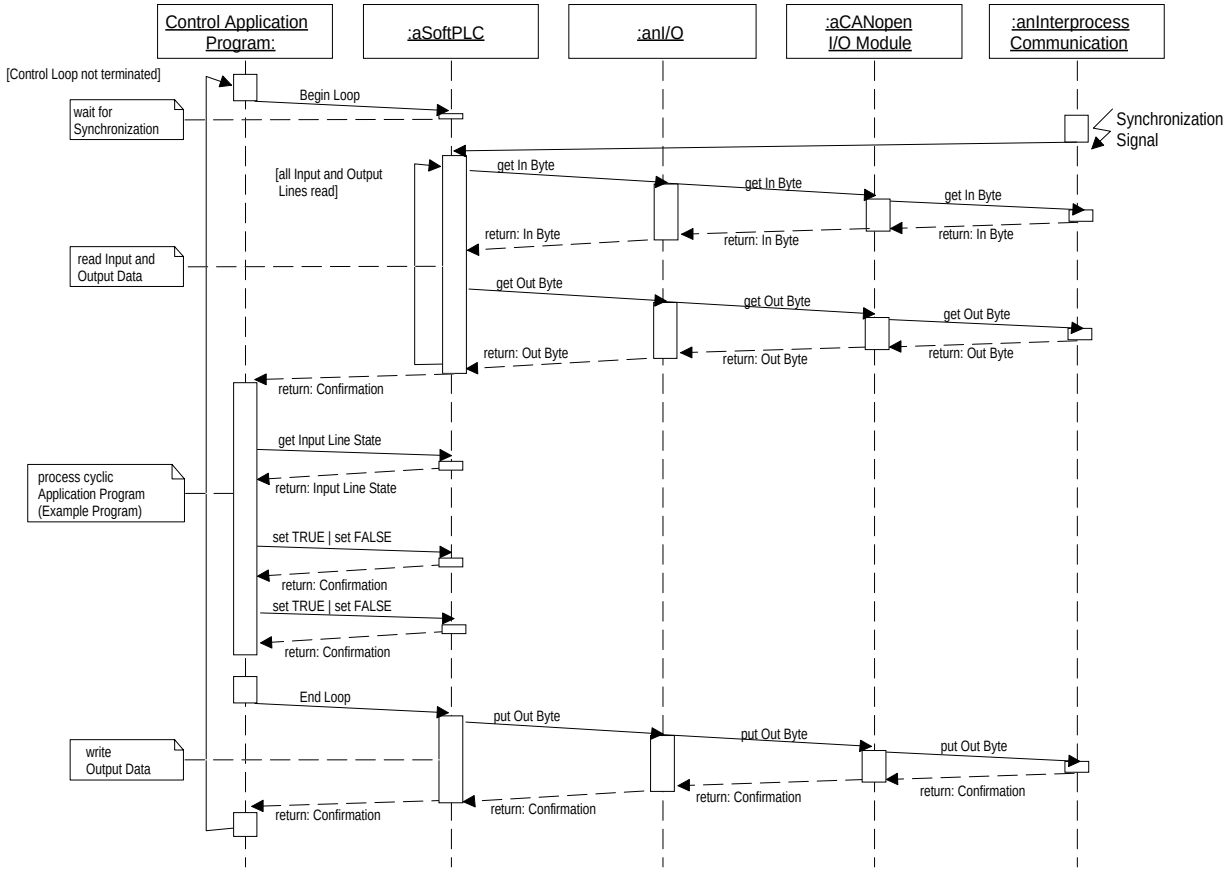


Abbildung 3.27: Zyklische Abarbeitung von SPS-Applikationsprogrammen

2. Zyklische Abarbeitung von SPS-Applikationen
   Dieses Vorgehen entspricht dem üblichen Vorgehen bei der Verarbeitung von SPS-Programmen (siehe Abschnitt 2.4.3.2).

   Zu Beginn eines Programmzyklus wartet das Applikationsprogramm auf das Synchronisationssignal. Anschließend werden alle Eingangs- und Ausgangswerte der gesteuerten E/A Baugruppen durch die Klasse *SoftPLC* eingelesen.

   Sobald die aktuellen Zustände der E/A Module in *SoftPLC* gespeichert sind, wird das eigentliche Applikationsprogramm ausgeführt. Als Ergebnis der Operationen werden die Ausgangswerte wieder in *SoftPLC* zurückgeschrieben.

   Nach dem Durchlauf durch das Anwendungsprogramm werden alle Ausgangswerte an die E/A Geräte versendet. Danach wartet das System wieder auf das nächste Synchronisationssignal.

3.3.5.3 Steuerung eines Transfersystems

Transfersysteme dienen meist zum Transport von Werkstücken. Sie bestehen aus unterschiedlichen Typen von Geräten: Antriebe und Geräte zur Beeinflussung des Materialfluß, wie z.B. Weichen, Vereinzelungsanlagen oder Hubstationen.
Der Materialfluß innerhalb von Fertigungszellen (siehe Abschnitt 2.4.6.1) ist von einem übergeordneten Transportsystem entkoppelt [8]. Da die Geräte zum Materialtransfer innerhalb von

[8] Die Fertigungszellen besitzen jeweils eigene Puffer für das in und aus der Zelle transportierte Material.

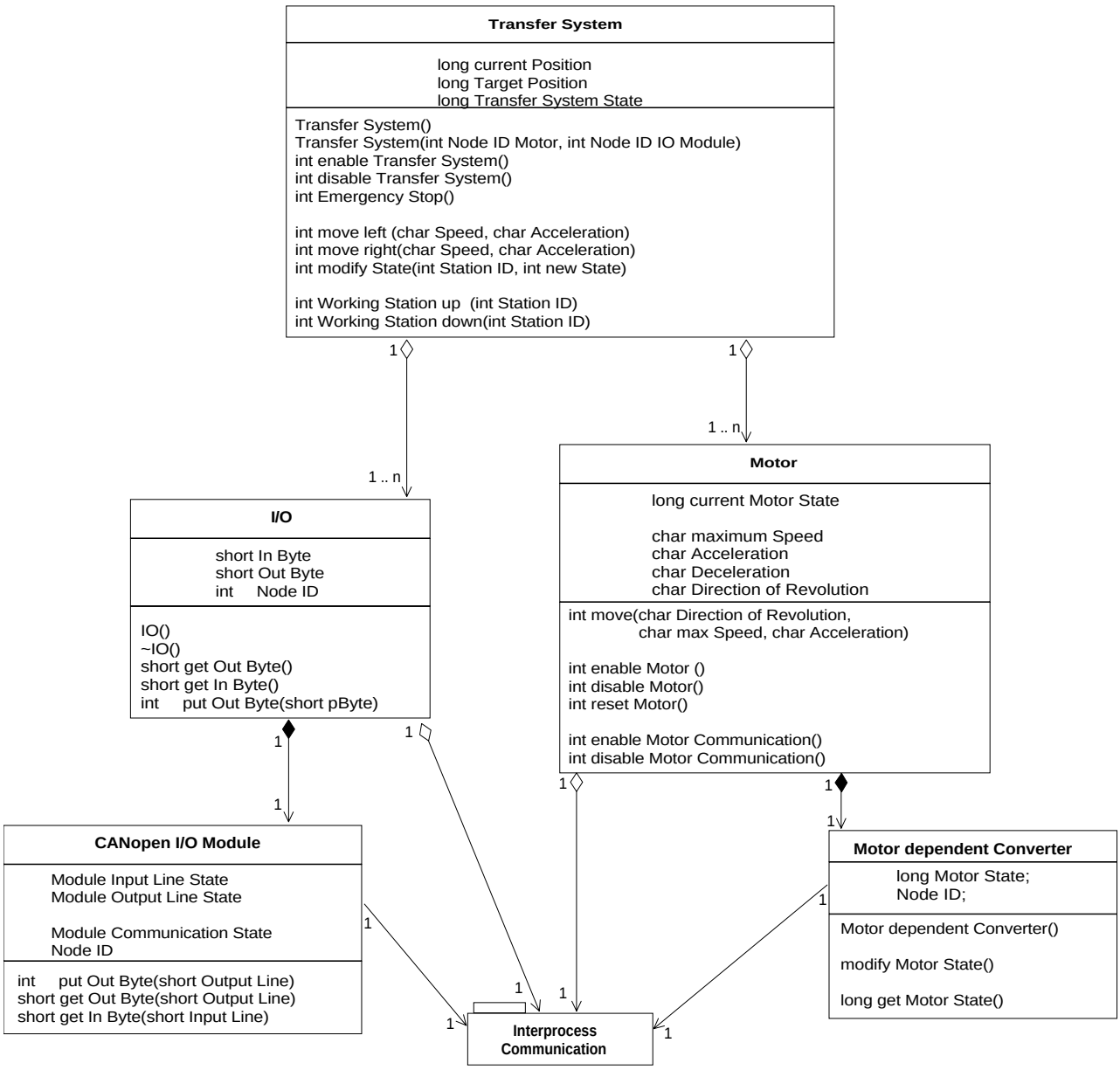


Abbildung 3.28: Klassen des Transfersystems

Fertigungszellen meist über digitale Schnittstellen angesteuert werden (Ausnahme: Drehzahlsteuerung von Antrieben), werden Transfersysteme durch speicherprogrammierbare Steuerungen oder von diesen Steuerungen abgeleiteten Spezialsteuerungen kontrolliert. Entsprechend der Aufzählung der zu steuernden Geräte in Abschnitt 3.1.1.2 wird eine Transfersystemsteuerung für den Materialtransport innerhalb von Fertigungszellen in die universelle Steuerung integriert. Ein derartiges Transfersystem kann aus ein bis mehreren Antrieben und digital angesteuerten Sensoren (z.B. mechanische und berührungslose Schalter sowie Lichtschranken) und Aktuatoren (z.B. Hubstationen oder Weichen) bestehen.
Im folgenden werden die Klassen und deren Verhalten näher beschrieben.

3.3.5.3.1 Klassenaufbau

Die Steuerung für das Transfersystem besteht aus der Klasse *Transfer System*, *Motor* und

Die Maschinen innerhalb der Fertigungszellen arbeiten autonom. Dies unterscheidet Fertigungszellen von Transferstraßen, bei denen die Geschwindigkeit des Materialfluß den Arbeitstakt der Maschinen bestimmt.

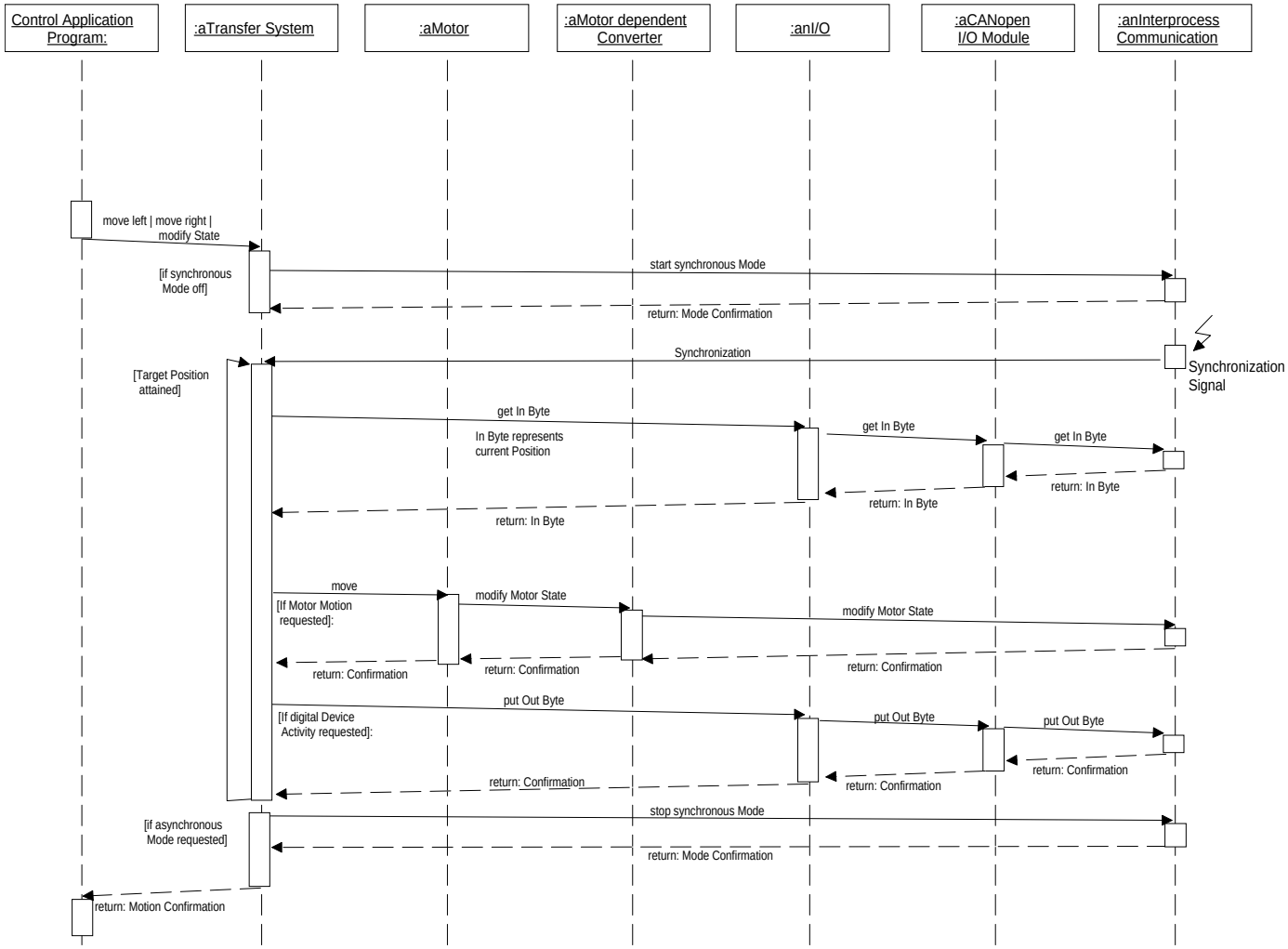


Abbildung 3.29: Dynamisches Verhalten der Transfersystemsteuerung

Motor dependent Converter. Zusätzlich werden die Klassen *I/O* und *CANopen I/O Module* aus der speicherprogrammierbaren Steuerung benutzt (siehe Abbildung 3.28).
Die Klasse *Transfer System* bildet (als Klasse aus der *complex Device Control Layer*) wie bei dem Subsystem zur Steuerung eines SCARA Roboterarms oder der integrierten speicherprogrammierbaren Steuerung die Anwendungsprogrammierungsschnittstelle (API).
Da die Transfersystemsteuerung zwei unterschiedliche Typen von Geräten kontrolliert, werden zwei Klassen (*Motor* und *I/O*) von *Transfer System* benutzt. Die Transfersystemsteuerung kann mit mehreren Geräten kommunizieren. Daher können mehrere Instanzen der Klassen *Motor* und *I/O* in *Transfer System* eingebunden werden.
Die Klasse *I/O* (mit der Unterstützungsklasse *CANopen I/O Module*) stammt aus der integrierten speicherprogrammierbaren Steuerung (siehe Abschnitt 3.3.5.2). Mit diesen Klassen werden die Geräte mit digitaler Schnittstelle angesprochen.
Zur Steuerung der Antriebe wird die Klasse *Motor* verwendet. Diese Klasse beinhaltet die Klasse *Motor dependent Converter*, um das CANopen Protokoll zu unterstützen. Im Gegensatz zu den Roboterantrieben besitzen die Antriebe des Transfersystems meist keine Positionsregelung, d.h. die Transfersystemantriebe können eine bestimmte Position nicht genau anfahren. Die Antriebe melden ihre Position nicht an die Steuerung zurück. Die einzelnen Werkstücke (bzw. Werkstückträger, d.h. Schlitten mit aufgespanntem Werkstück) werden

im Transfersystem durch Rückmeldungen der Sensoren lokalisiert (Kommunikation über die Klasse *I/O*). Anhand dieser Meldungen kontrolliert die Transfersystemsteuerung die Antriebe.

3.3.5.3.2 Dynamisches Verhalten

Das dynamische Verhalten der Transfersystemsteuerung entspricht grundsätzlich dem Verhalten der Steuerung des Roboterarms (siehe Abschnitt 3.3.5.1, Abbildung 3.24).
Transfer System startet nach Erhalt eines Befehls von einem *Control Application Program* zunächst den synchronen Kommunikationsmodus. Anschließend wird eine Schleife solange durchlaufen, bis die Anweisung ausgeführt ist. Zu Beginn der Schleife wird von den Instanzen der Klasse *I/O* der aktuelle Zustand der Geräte des Transfersystems eingelesen. Darauf werden je nach Steuerungsalgorithmus Befehle an die Antriebe (über die Klasse *Motor*) oder an die digitalen Aktuatoren des Transfersystems geschickt. Das System wartet danach auf den Beginn eines neuen Durchlaufs (Synchronisationssignal).

3.3.6 User Interface

Das *User Interface* ist die interaktive Schnittstelle, über die der Benutzer direkt in das universelle Steuerungssystem eingreifen kann. Dies ist die einzige Anforderung an das Design des *User Interface*, die sich aus den Use Case Modellen der Analyse ergibt (siehe Abschnitt 3.2.2.1).
Im folgenden werden die Funktionalität und die Klassen des *User Interface* sowie deren Verhalten vorgestellt.

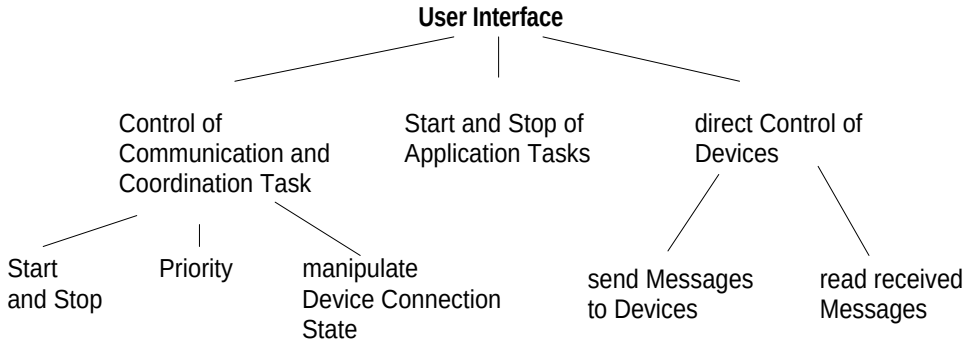


Abbildung 3.30: Hierarchie des Benutzermenüs

3.3.6.1 Funktionalität des User Interface

Das *User Interface* bietet dem Anwender die folgenden Menüpunkte (siehe Abbildung 3.30):

1. Kontrolle der zyklischen *Communication and Coordination Task*
   Dieses Untermenü ermöglicht die Kontrolle der *Communication and Coordination Task*. Dazu gehört das Starten und Stoppen der Task sowie die Einstellung der Priorität. Ebenso kann das CANopen Netzwerk über diesen Menüpunkt manipuliert werden.

2. Start und Stopp der *Application Tasks*
   Dieser Menüpunkt dient zum Starten und Stoppen der *Application Tasks*. Zum Start

(oder Stoppen) muß der Name der *Application Task* angegeben werden. Daraufhin wird automatisch die entsprechende Task erzeugt bzw terminiert.

3. Direkte Kontrolle der Geräte
   Dieser Menüpunkt bietet die Möglichkeit, Feldbustelegramme unmittelbar an Geräte am CAN-Bus zu versenden. Ebenso können die von den Geräten zurückgesendeten Telegramme vom Benutzer gelesen werden. Dadurch kann der Anwender Geräte am CAN-Bus direkt testen.

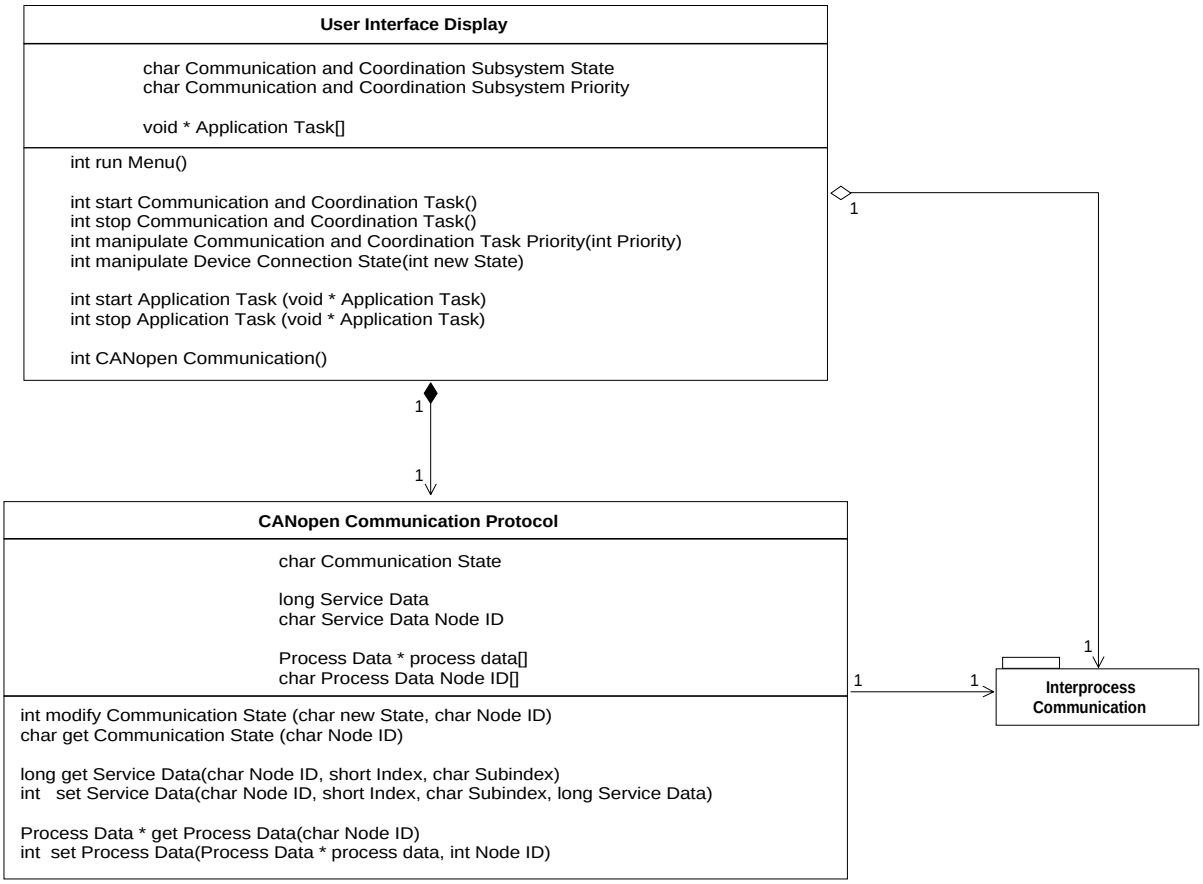


Abbildung 3.31: Klassendiagramm des *User Interface*

3.3.6.2 Klassenaufbau

Die Klasse *User Interface Display* bildet die Schnittstelle zum Benutzer. Zur Kommunikation mit dem *Coordination* Package greift sie auf *Interprocess Communication* zu. Der Zugriff auf die Geräte am CAN-Bus und die Manipulation des Kommunikationszustands entsprechend des Busprotokolls geschieht mittels der Klasse *CANopen Communication Protocol*. Diese Klasse ist durch eine Kompositionsbeziehung in *User Interface Display* eingebunden.

3.3.6.3 Dynamisches Verhalten

Beim Start der *User Interface Task* wird die Methode `int run_Menu()` der Klasse *User Interface Display* angestoßen. In dieser Methode wird die Maske des Benutzermenüs auf-

gebaut, die Eingaben des Benutzers abgefragt und die Methode, die dem vom Anwender ausgewählten Menüpunkt zugeordnet ist, aufgerufen.
Jede Auswahl wird (zunächst) von der Methode `int run_Menu()` bearbeitet. Diese Methode ruft anschließend die entsprechende Methoden der Klasse zur Bearbeitung der Anweisung auf. Eine Ausnahme bilden die Menüpunkte zur Kommunikation über den CAN-Bus. Diese werden direkt an die entsprechenden Methoden der Klasse *CANopen Communication Protocol* weitergegeben.

3.3.7 Tasks des Steuerungssystems

Bei der Entwicklung von Echtzeitsystemen wird die Einteilung der Klassen in Tasks möglichst spät vorgenommen [AKZ96]. Im folgenden werden die einzelnen Typen von Tasks der Steuerung und deren Zeitverhalten vorgestellt.

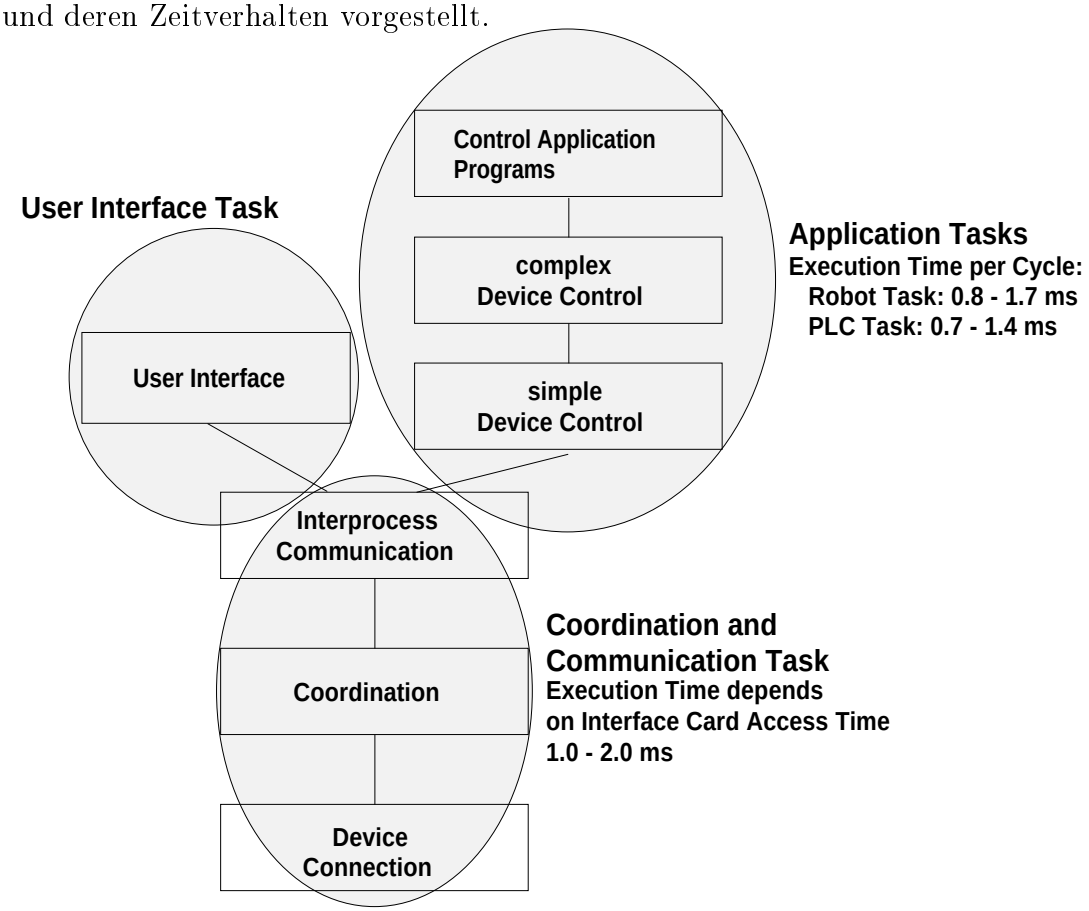


Abbildung 3.32: Kollaborationsdiagramm mit Einteilung der Tasks und Zeitabschätzung

3.3.7.1 Einteilung in Tasks

Die in den vorangegangenen Abschnitten (3.3.2, 3.3.3, 3.3.5 und 3.3.6) beschriebenen Klassen und Objekte werden hier in einzelne lauffähige Einheiten, die in dieser Arbeit als Tasks bezeichnet werden [9], aufgeteilt. Bei diesem System orientiert sich die Einteilung an der schon in der Analyse eingeführten Gliederung (siehe Abschnitt 3.2.2, Funktionale Anforderungen).

[9] Diese Tasks sind in der universellen Steuerung sowohl als Prozesse als auch als Threads realisiert (siehe Abschnitt 6.1.1).

Eine derartige Übereinstimmung zwischen der Gliederung der Anforderungen in der Analyse und der Einteilung der Klassen in Tasks ist nicht bei allen Echtzeitsystemen gegeben.
Im Steuerungssystem kommen drei Typen von Tasks vor (siehe Abbildung 3.32, die angegebenen Leistungsdaten werden in Abschnitt 6.3 näher erläutert):

1. *Communication and Coordination Task*
   Die Task besteht aus dem Package *Coordination* (enthält die Klassen *Coordination Manager*, *Command* als abstrakte Schnittstelle der Klassen *Device Communication*, *Connection Command* und *System Command*) und übernimmt die Aufgaben dieses Package (siehe Abschnitt 3.3.1). Außerdem enthält die Task das Package *Interprocess Communication* (mit den Klassen *Interprocess Communication Manager*, *System Access Interface*, *local Control*, *Data Exchange* mit *Forwarder* und *Receiver* und *Task Synchronization*) sowie die Klasse *Device Connection*.

   Die *Communication and Coordination Task* darf nur einmal in einem Steuerungssystem existieren.

2. *User Interface Task*
   Die Task mit der Benutzerschnittstelle umfaßt die Klassen *User Interface Display* und *CANopen Communication Protocol*. Zur Kommunikation mit der *Communication and Coordination Task* greifen die Klassen auf das Package *Interprocess Communication* zurück. Daher ist dieses Package ebenfalls Teil der *User Interface Task*. Der Aufbau und das Verhalten des *User Interface* wird in Abschnitt 3.3.6 beschrieben.

3. *Application Tasks*
   Der Aufbau dieser Tasks wird vom Anwender, dem Entwickler dieser *Application Tasks*, bestimmt [10]. Im Gegensatz zu den anderen Tasks kann die Zusammensetzung dieses Task-Typs variieren. Die *Application Tasks* können entweder nur einen Gerätetyp (*simple Application Tasks*) oder mehrere Gerätetypen (*complex Application Tasks*) unterstützen.

   Ein typisches Beispiel für eine *simple Application Task* ist eine reine SPS-Task. Speicherprogrammierbare Steuerungen greifen meist nur auf E/A Module zu. Daher ist es ausreichend, wenn eine derartige SPS-Task neben einem Anwendungsprogramm und einem Package *Interprocess Communication* nur aus den Klassen der speicherprogrammierbaren Steuerung (siehe Abschnitt 3.3.5.2) besteht.

   Eine Roboter-Task, die zusätzlich zur Kontrolle des Roboterarms auch ein peripheres Gerät (z.B. einen Greifer als Endeffektor) steuert, ist eine *complex Application Task*. Diese Task enthält neben dem Roboteranwendungsprogramm und der *Interprocess Communication* sowohl die Robotersteuerungsklassen aus Abschnitt 3.3.5.1 (zur Steuerung des Roboterarms) als auch die Klassen der speicherprogrammierbaren Steuerung zur Ansteuerung des Greifers.

Diese Tasks kommunizieren und synchronisieren sich mit Interprozeßkommunikationsmechanismen wie Shared Memory, Mutex und Semaphore sowie Signalen. Die Mechanismen werden in den Klassen *Data Exchange* und *Task Synchonization* des Package *Interprocess Communication* realisiert (siehe Abschnitt 3.3.2).

[10] Die *Control Application Programs* sind nicht Teil des Steuerungssystems. Trotzdem werden sie zusammen mit den *Control Packages* zu einer Task zusammengefaßt, da eine Trennung einen zusätzlichen Kommunikationsaufwand mit sich bringen würde.

Eine alternative Einteilung der Tasks wird im Konzept einer offenen Multitasking-Robotersteuerung (ISWRC) des ISW verwendet [PUD94]. Diese Steuerung ist im Rahmen des Projekts OSACA (siehe Abschnitt 2.4.5.1) entwickelt worden [OSA96]. Bei diesem Konzept sind die Tasks streng horizontal entsprechend der Schichten des Steuerungssystems angeordnet. Beispielsweise arbeitet eine Task als Programminterpreter und Preinterpolator, die das Anwendungsprogramm interpretativ abarbeitet. Während eine zweite Task dann die Werte der Interpreter-Task übernimmt und diese interpoliert sowie die Koordinatentransformation durchführt. Die Ergebnisse dieser Interpolations-Task werden anschließend von einer dritten Task, dem sogenannten Feininterpolator (Fine Interpolator) und der Lageregelung (Position Controller) abschließend bearbeitet und an die Roboterantriebe geschickt. Alle drei Tasks werden von unterschiedlichen Kontrollinstanzen aktiviert (die Interpreter-Task durch das Anwendungsprogramm, die Interpolations-Task durch den Interpolator Cycle und die Lageregelungs-Task durch die Lageregelungsschleife).
Dieses Konzept hat den Vorteil, daß es auch bei nicht-objektorientiertem Vorgehen eine Trennung der Belange und Aufgaben zwischen einzelnen Systemkomponenten bietet. Dadurch kann dieses Konzept auch mit low-level Programmiersprachen wie z.B. Assembler oder C umgesetzt werden.
Wird jedoch eine Steuerung objektorientiert entwickelt, so können die einzelnen Funktionalitäten in Objekten und Packages gekapselt werden. Dadurch können mehrere in einer Abfolge stehende Funktionalitäten in einer Task zusammengefaßt werden. Damit verringert sich der Kommunikations- und Synchronisations-Overhead gegenüber einem System nach dem ISWRC Konzept.

3.3.7.2 Zeitliches Verhalten der Tasks

In diesem Abschnitt werden die Prioritäten der Tasks bestimmt und deren zeitliches Verhalten durch Untersuchungen verifiziert.

3.3.7.2.1 Bestimmung der Prioritäten der Tasks

Die Ermittlung der Taskprioritäten erfolgt nach dem in [AKZ96] beschriebenen Vorgehen. Auf der Basis der auf diese Weise bestimmten Prioritäten wird das *Rate Monotonic Scheduling (RMS)* [Tan92, Gal95, Dou98b] angewendet. Dieses setzt preemptives und prioritätsgesteuertes Scheduling, das von allen in dieser Arbeit verwendeten Betriebssystemen (siehe Abschnitt 6.2) unterstützt wird, voraus.

Task(lauffähig) / **Task(laufend)**	*Communication and Coordination Task*	*Application Task*	*User Interface Task*
Communication and Coordination Task	X	Task nicht unterbrechen	Task nicht unterbrechen
Application Task	Task unterbrechen	X	Task nicht unterbrechen
User Interface Task	Task unterbrechen	Task unterbrechen	X

Tabelle 3.3: Unterbrechungskombinationen zwischen den Tasks

In Tabelle 3.3 wird beschrieben, welche Task durch welche andere Task unterbrochen bzw. nicht unterbrochen werden darf. Da die *Communication and Coordination Task* zur Steuerung und Regelung der Geräte immer vollständig in einer festen, periodischen Zeitspanne

ausgeführt werden muß, darf diese Task von keiner anderen unterbrochen werden. Dagegen kann die *User Interface Task* von allen anderen Tasks verdrängt werden. Vom Benutzer können mehrere *Application Tasks* gestartet werden. Diese können sich daher gegenseitig unterbrechen. Im weiteren werden jedoch Kollisionen zwischen *Application Tasks* nicht betrachtet, da die Priorität(en) dieser Tasks nur innerhalb einer bestimmten Prioritätsklasse variieren können (die Prioritäten der *Application Tasks* werden vom Anwender vergeben). Dadurch können die *Application Tasks* als ein Tasktyp mit einer eigenen Priorität angesehen werden.
Es ergibt sich damit folgende Prioritätsreihenfolge:

1. *Communication and Coordination Task* (höchste Priorität)

2. *Application Tasks*

3. *User Interface Task* (niederste Priorität)

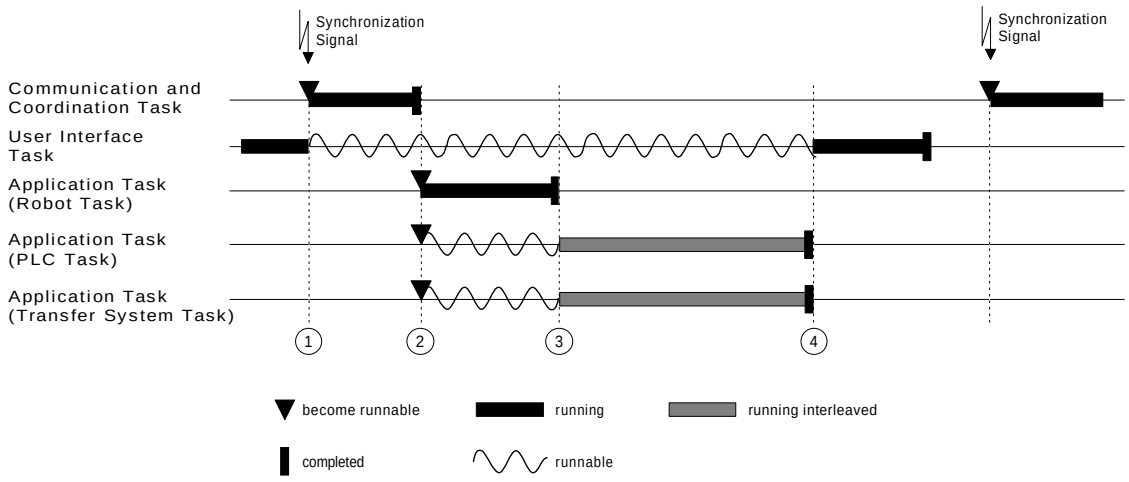


Abbildung 3.33: Zyklischer Ablauf der Tasks (Rechner mit einem Prozessor)

3.3.7.2.2 Zeitlicher Ablauf der Tasks

Entsprechend der im vorherigen Abschnitt vergebenen Prioritäten verhalten sich die Tasks wie in Abbildung 3.33 beschrieben (Beispiel bei einem Einprozessorsystem):

1. Beginn des Zyklus: Das Synchronisationssignal aktiviert die *Communication and Coordination Task*. Da diese Task die höchste Priorität besitzt, wird sie sofort ausgeführt. Die *Communication and Coordination Task* tauscht Daten mit den gesteuerten Geräten aus und aktiviert anschließend alle *Application Tasks*.

2. Wenn der Anwender der Roboter-Task die für *Application Tasks* höchstmögliche Priorität vergibt, so wird diese Task als erste *Application Task* ausgeführt.

3. Nach Abarbeitung der Roboter-Task laufen die anderen *Application Tasks* ab. Wird davon ausgegangen, daß diese Tasks die gleiche Priorität besitzen, dann wechseln sich beide Tasks möglicherweise mehrfach ab.

4. Nachdem alle *Application Tasks* abgearbeitet sind, wird die *User Interface Task* wieder eingeplant. In dem Szenario war diese Task schon vor dem Zeitpunkt 1 aktiv. Aufgrund ihrer niedrigen Priorität ist sie aber von allen anderen Tasks verdrängt worden.

Bei einem Mehrprozessorrechner können mehrere *Application Tasks* (bzw. die *User Interface Task*) zeitgleich abgearbeitet werden. Damit inkonsistente Zustände beim Datenaustausch vermieden werden, dürfen *Application Tasks* (d.h. Tasks, die zyklisch über die *Communication and Coordination Task* Daten mit den Geräten austauschen) nicht gleichzeitig mit der *Communication and Coordination Task* abgearbeitet werden. Die *Communication and Coordination Task* aktiviert nach Abschluß der Kommunikation mit den Geräten die *Application Tasks*.
Im Gegensatz zu den heute üblichen Robotersteuerungen, wie z.B. die rho3 von Bosch (Echtzeitschleife) bietet die hier beschriebene Steuerung einen Multitasking-Betrieb. Die Tasks werden nicht in einer festen Reihenfolge nacheinander bearbeitet, sondern sie werden dynamisch entsprechend ihres Zustands (lauffähig oder blockiert) und ihrer Priorität eingeplant (siehe Abschnitt 2.4.2.2.3).

Kapitel 4

Wiederverwendung

Nach der Entwicklung und Implementierung einer universellen Robotersteuerung auf einer SPARC Workstation sind verschiedene Portierungen des Systems erfolgt. Die neuen Plattformen für das Steuerungssystem sind Windows NT und Linux (PC-Architektur mit Intel386-kompatiblem Prozessor) sowie ein Siemens Industrie-PC (IMC 05) mit dem Echtzeitbetriebssystem RMOS. Zusätzlich ist das System um die Möglichkeit der Steuerung eines Dreiachsportalsystems (numerische Steuerung) erweitert worden. Da mit dieser NC nun die wichtigsten industriellen Steuerungsvarianten (siehe Abschnitt 1.1) vereint sind, kann von einem universellen Steuerungssystem gesprochen werden.
Die Grundlage jeder Portierung ist die Wiederverwendung der Architektur und des Codes. Die genaue Vorgehensweise wird in den folgenden Abschnitten beschrieben.
Zunächst wird ein Architekturmuster als Vorlage für den Entwurf eines objektorientierten Steuerungssystems eingeführt.
Darauf wird ein Framework vorgestellt, bei dem im Gegensatz zum Architekturmuster die Wiederverwendung von Code (Code Reuse) im Vordergrund steht. Dieses Steuerungs-Framework ist konventionell entsprechend des in Abschnitt 2.2 beschriebenen Vorgehens entwickelt worden.
Im dritten Teil wird ein allgemeines Architekturmodell zur optimierten Entwicklung komponentenbasierter Frameworks als neuer Ansatz [PRST99] vorgestellt. Ein solches komponentenbasiertes Framework, das nach diesem Vorbild erstellt wird, erlaubt eine bessere Wiederverwendung, da es wesentlich flexibler entwickelt werden kann wie objektorientierte Frameworks.
Den Abschluß des Kapitels bildet eine Bewertung der vorgestellten Konzepte.

4.1 Architekturmuster für industrielle Steuerungen

Die im Abschnitt 3.3 vorgestellte Architektur kann als Muster für die Entwicklung von weiteren industriellen Steuerungssystemen verwendet werden [1]. (Eine Einführung in Architekturmuster findet sich in Abschnitt 2.1.2.1, Absatz Architekturmuster.)
Der Name des Architekturmusters ist "**universelles industrielles Steuerungssystem**" (universal industrial Control System). Die Problemstellung, bei der das Muster eingesetzt werden kann, ist die objektorientierte Entwicklung eines industriellen Steuerungssystems.

[1] Konkret ist dieses Architekturmuster bei der Entwicklung des Steuerungssystems auf einem Siemens Industrie-PC mit dem Betriebssystem RMOS und auf einen PC mit dem Betriebssystem Windows NT erfolgreich verwendet worden. Aufgrund der großen Differenzen zwischen den Betriebssystemen (Unix und RMOS bzw. Windows NT) ist eine Wiederverwendung des Codes in größerem Umfang nicht möglich.

Besonders geeignete Plattformen für dieses Steuerungssystem sind Standardrechner mit Betriebssystemen nach den POSIX-Standards wie z.B. Workstations oder PCs, bzw. die im industriellen Umfeld verwendeten Varianten Industrie-Workstations oder Industrie-PCs (siehe Abschnitt 2.4.5.2). Das Muster kann dazu verwendet werden, Systeme zu entwickeln, die nur einen bestimmten Gerätetyp oder auch eine Vielzahl von Gerätetypen steuern. Im Gegensatz zu monolithischen Steuerungen hat die vorgestellte Architektur den Vorteil, daß auf Basis des zentralen *Communication and Coordination Subsystem* (wie in der Analyse beschrieben, siehe Abschnitt 3.2.2.1) unterschiedliche Steuerungsfunktionen ausgeführt werden. Bei der Entwicklung und Kombination von verschiedenen Steuerungstypen kann immer das gleiche *Communication and Coordination Subsystem* wiederverwendet werden. Damit eignet sich dieses Architekturmuster besonders für den Bau offener und universeller Steuerungssysteme. Im folgenden wird das Muster, bzw. dessen Merkmale, sowie die im Architekturmuster verwendeten Entwurfsmuster und deren Anordnung vorgestellt. Zum Abschluß werden die Konsequenzen des Musters diskutiert.

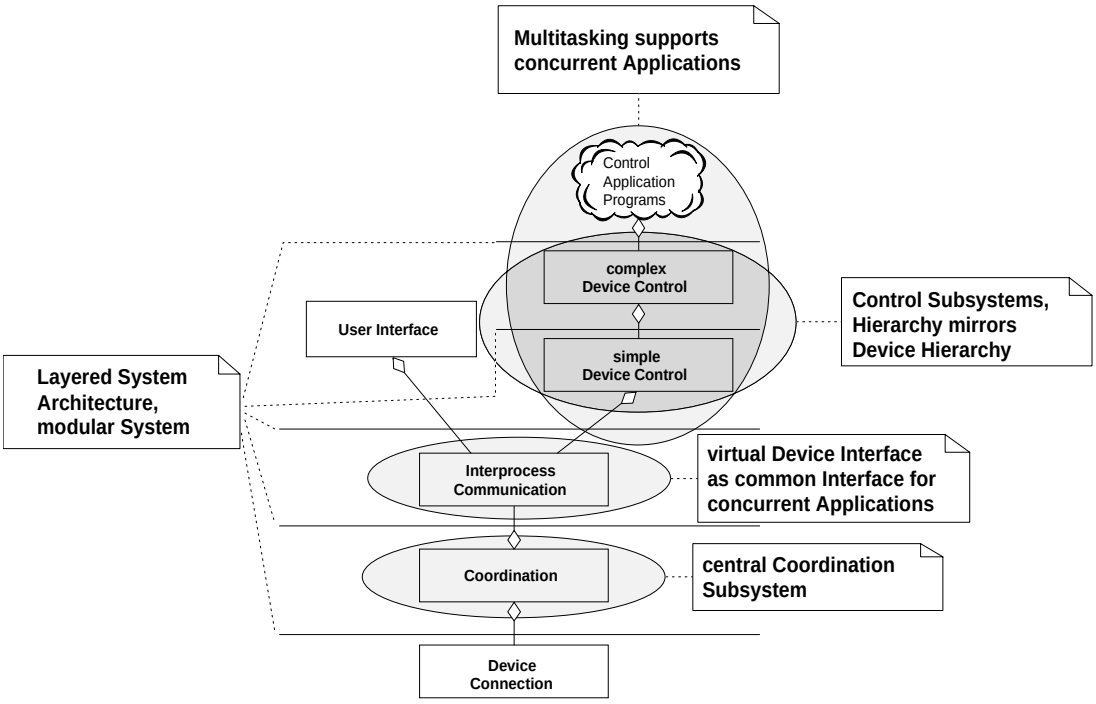


Abbildung 4.1: Merkmale des Architekturmusters

4.1.1 Merkmale des Architekturmusters

Die folgenden Merkmale definieren das Architekturmuster (siehe Abbildung 4.1):

- Einteilung des Systems in horizontale Schichten
  Die Einteilung in Schichten ist ein übliches Vorgehen zur Gliederung von objektorientierten Echtzeitsystemen [SGW94, Gal95]. In [BMR+96] und [Dou98b] wird die Verwendung von Schichten sogar als Architekturmuster *Layers* vorgestellt.

  Durch die eindeutige Definition der Schnittstellen zwischen den Schichten (siehe Abschnitt 3.3.1) können die Schichten modular getauscht werden. Soll beispielsweise über

eine andere CAN-Schnittstellenkarte mit anderen Kartentreiberaufrufen kommuniziert werden, so muß nur das Subsystem *Device Connection* ausgetauscht werden.

Die fünf Schnittstellen sind so gelegt, daß entsprechend der Funktionalität (bzw. Verantwortlichkeit) zusammengehörige Klassen in einer Ebene zusammengefaßt sind. Die Schicht *Coordination* übernimmt z.B. die zentrale Koordination des Verhaltens und der Komunikation innerhalb der Steuerung und zu den Geräten. Dabei initiiert sie die Kommunikation indem sie die Dienste der anderen Ebenen benutzt. Die Schichten *Interprocess Communication* und *Device Connection* realisieren den Datenaustausch, ohne sich um die zeitliche Koordination zu kümmern.

- Virtuelle Geräteschicht (*Virtual Device Interface*)
  Die virtuelle Geräteschicht (entspricht dem Package *Interprocess Communication* in Abschnitt 3.3.2) ermöglicht mehreren konkurrenten *Application Tasks* den Zugriff auf Geräte. Dabei wird festgelegt, welche *Application Task* auf welches Gerät wie zugreifen darf (schreibend und lesend, nur lesend, kein Zugriff).

  Unter der virtuellen Geräteschicht verbergen sich die eigentlichen Kommunikationsmechanismen und die Koordination des Systems, die von einer zentralen Task (*Communication and Coordination Task*) übernommen werden.

  Instanzen der Klassen der virtuellen Geräteschicht kommen sowohl in den *Application Tasks* (bzw. *User Interface Task*) als auch in der *Communication and Coordination Task* vor.

- Zentrale Koordinationsschicht *Coordination Subsystem*
  Diese Schicht steuert das dynamische Verhalten des Systems (siehe Abschnitt 3.3.3). Dabei werden die *Application Tasks* (wie in Abbildung 4.2 dargestellt) zyklisch abgearbeitet. Dieses Verhalten ist charakteristisch für Steuerungssysteme.

  Sie synchronisiert die *Application Tasks* und initiiert die Kommunikation zwischen diesen Tasks und den gesteuerten Geräten.

  Aufgrund ihrer zentralen Aufgaben dürfen die Instanzen der Klassen dieser Schicht jeweils genau einmal in einem Steuerungssystem vorkommen.

- Steuerungssubsysteme
  Die Subsysteme mit den Algorithmen zur Steuerung von Geräten (siehe Abschnitt 3.3.5) werden entsprechend der Hierarchie der Geräte angeordnet. Die übergeordneten komplexen Automatisierungsgeräte, die aus einfachen Geräten zusammengesetzt sind, werden von komplexen Subsystemen kontrolliert, die ihrerseits einfache Steuerungssubsysteme benutzen.

  Die Steuerungssubsysteme können beliebig untereinander ausgetauscht werden. Werden beispielsweise bei einem komplexen Gerät wie einem Roboterarm die Antriebe getauscht, so kann dies in der Steuerungs-Software nachvollzogen werden, indem die Klassen zur Kontrolle der Antriebe ausgetauscht werden.

- Konkurrent ablaufende *Application Tasks*
  Die *Application Tasks*, d.h. die vom Anwender geschriebenen Steuerungsprogramme mit den entsprechenden Steuerungssubsystemen, werden konkurrent abgearbeitet. Bei Mehrprozessorrechnern ist eine echte Parallelität möglich. Die Einplanung der Tasks erfolgt durch einen Scheduler entsprechend der Priorität und des Zustands der Task (siehe Abschnitt 3.3.7.2.2).

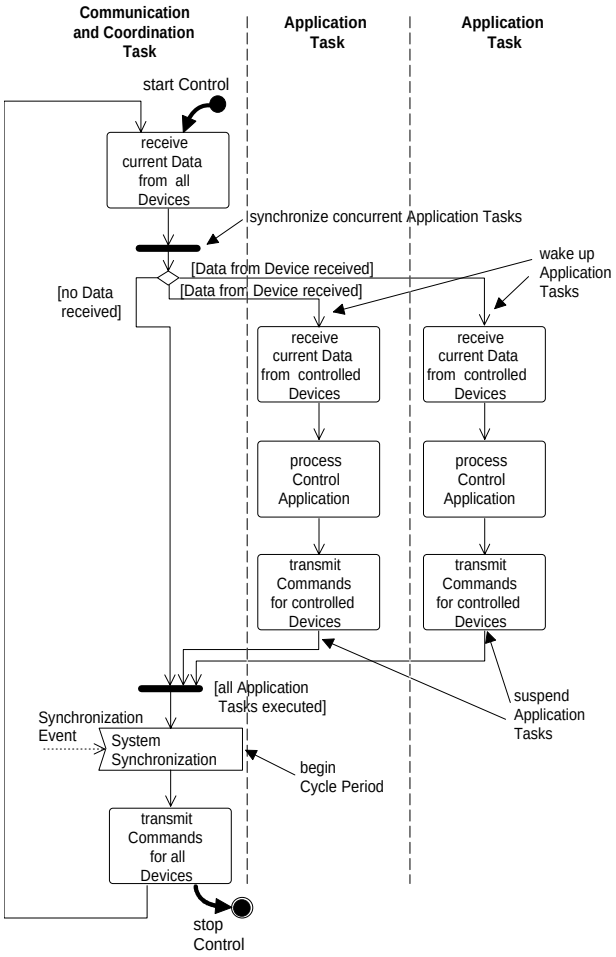


Abbildung 4.2: Dynamisches Verhalten

4.1.2 Entwurfsmuster im Architekturmuster

Neben der reinen Beschreibung der Struktur und des Verhaltens sind auch die zur Verwendung vorgeschlagenen Entwurfsmuster von Interesse. Diese beschreiben das Design der Architekturdetails. Durch diese Muster werden die Kooperationen zwischen den Subsystemen des Gesamtsystems oder der interne Aufbau von Subsystemen modelliert [BMR+96]. (Die hier erwähnten Entwurfsmuster sind Empfehlungen zur Vereinfachung der Entwicklung. Selbstverständlich können die Detailprobleme auch auf andere Art gelöst werden.)
Folgende Entwurfsmuster sind für die Realisierung der Architekturdetails geeignet (siehe Abbildung 4.3):

- *Command* nach [GHJV94]
  Dieses Muster beschreibt die Bearbeitung der systeminternen Kommandos und die Kommunikation mit den Geräten. Die systeminternen Kommandos werden entweder durch Klassen von *Coordination* oder *Device Connection* bearbeitet. Die Kommunikation mit den Geräten wird von *Device Connection* übernommen.

  Werden viele Klassen zur Bearbeitung der Kommandos benötigt, so können diese entsprechend dem Entwurfsmuster *Chain of Responsibility* aus [GHJV94] angeordnet werden.

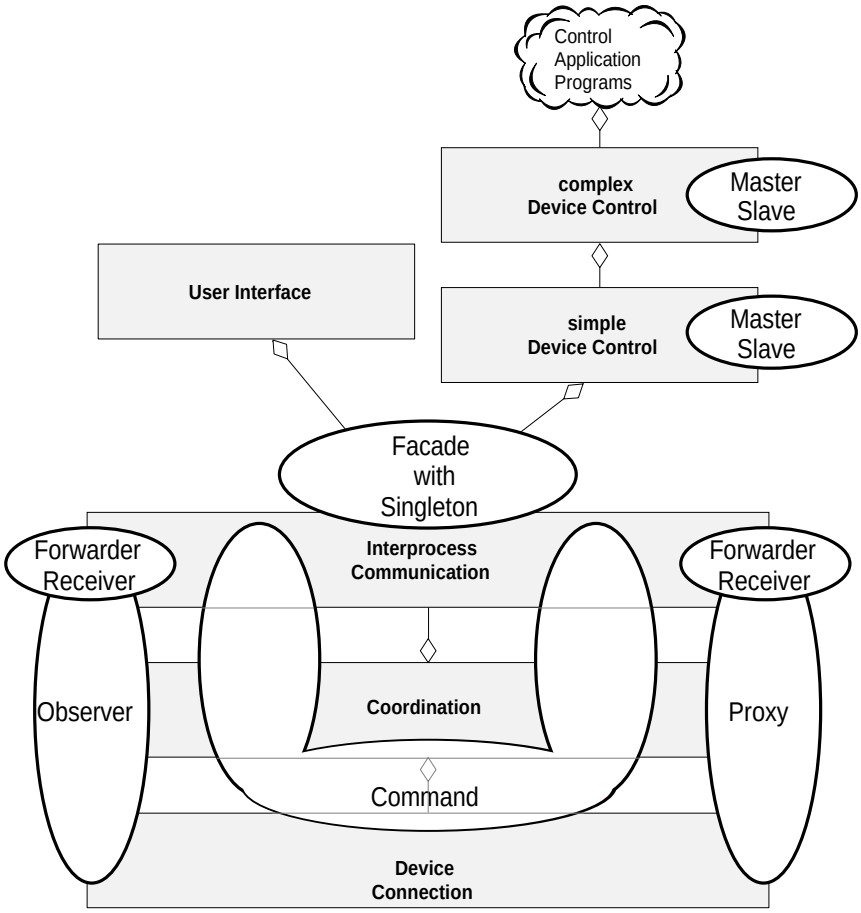


Abbildung 4.3: Verwendete Entwurfsmuster

- *Proxy* nach [GHJV94] und [Dou98b]
  Entsprechend dem Muster *Proxy* greifen die *Control Packages* und das *User Interface* auf die gesteuerten Geräte schreibend zu. Über diese Schnittstelle werden die Nachrichten an die Geräte verschickt.

- *Observer* nach [GHJV94] und [Dou98b]
  Der Empfang von Werten von den Geräten geschieht nach dem *Observer* Muster. Die *Application Tasks* werden von *Coordination* mit Hilfe von *Interprocess Communication* aktiviert (bzw. synchronisiert). Darauf entscheiden die *Application Tasks* wie sie reagieren sollen (z.B. Lesen der aktuellen Gerätedaten mittels *Interprocess Communication*).

- *Forwarder Receiver* nach [BMR+96]
  Zum Austausch der Daten (nach dem Muster *Proxy* und *Observer*) wird der in *Forwarder Receiver* beschriebene Mechanismus verwendet. Nach diesem Muster haben die beiden Partner, die Daten miteinander austauschen, zwei Klassen *Forwarder* und *Receiver*, über die der Austausch geschieht. Diese beiden Klassen sind spiegelbildlich aufgebaut, d.h. Daten, die vom *Forwarder* verschickt werden, empfängt der *Receiver*.

  Für eine Duplex-Kommunikation, wie sie im Steuerungssystem vorkommt, besitzt jeder Kommunikationspartner jeweils eine *Forwarder* und eine *Receiver* Klasse.

- *Facade* und *Singleton* nach [GHJV94]
  Diese beiden Muster beschreiben, wie auf die *Interprocess Communication* zugegriffen

werden kann. Die Muster *Facade* und *Singleton* ergänzen sich dabei.

Da in einer *Application Task* bzw. *User Interface Task* nur jeweils eine Instanz der Klassen von *Interprocess Communication* vorkommen darf, wird auf *Interprocess Communication* über das *Singleton* Muster zugegriffen.

Das Entwurfsmuster *Facade* verbirgt die Klassen der *Interprocess Communication*.

- *Master Slave* nach [BMR+96] und [Dou98b]
  Das *Master Slave* Muster beschreibt die Einbindung von Hilfsklassen in die *Control Packages*.

  Beispiele für ein konkretes Design sind (siehe Abschnitt 3.3.5): Die Beziehungen der Klassen *Trajectory Planning Interpolation* und *Closed Loop Position Control* mit der *complex Device Control* Package Klasse *SCARA SR60* oder die Einbindung der jeweiligen Feldbusprotokollklassen (wie *CANopen I/O Module*) in die Klassen der *simple Device Control* Packages.

Die vorgestellten Entwurfsmuster interagieren untereinander innerhalb des Architekturmusters. In diesem Sinne kann das Architekturmuster, das die Anordnung und die Interaktionen der Entwurfsmuster definiert, auch als Pattern Language [AIS+77, Ale77, GHJV94, BMR+96, Ale99] bezeichnet werden, d.h. als abgeschlossene Gruppe von interagierenden Mustern, die für eine bestimmte Applikationsdomäne verwendet werden können.

4.1.3 Konsequenzen

In diesem Abschnitt werden die Auswirkungen des Architekturmusters vorgestellt, da eine derartige Beschreibung ein fester Bestandteil der Dokumentation von Mustern darstellt [GHJV94].
Die Anwendung des Musters hat verschiedene positive Auswirkungen auf die objektorientierte Entwicklung von universellen Steuerungssystemen. Die Konsequenzen der Verwendung des Architekturmusters sind:

- Universelle Anwendbarkeit
  Das Architekturmuster kann unabhängig von einem konkreten Steuerungstyp (z.B. Robotersteuerung, speicherprogrammierbare Steuerung oder numerische Steuerung) angewendet werden.

  Dies ist dadurch möglich, daß zum einen das Muster ein generisches Verhalten der Steuerungssysteme realisiert (zyklische Bearbeitung der Steuerungsfunktionen). Zum anderen können durch den modularen Aufbau der Architektur beliebige Steuerungstypen verwendet werden. Durch die Teilung der Steuerungssubsysteme in komplexe und einfache Subsysteme kann bei der Einführung zusätzlicher komplexer Steuerungssubsysteme auf bestehende einfache Subsysteme zurückgegriffen werden.

  Bei den bisherigen Implementierungen sind alle Applikationsprogramme immer compiliert und mit den Steuerungssubsystemen zusammengebunden worden. Allerdings können Anwendungsprogramme auch interpretativ abgearbeitet werden. Dazu muß der entsprechende Interpreter als Steuerungssubsystem implementiert werden.

- Multitasking
  Das Muster bietet die Möglichkeit, mehrere *Application Tasks* zur Ansteuerung unterschiedlicher Geräte gleichzeitig ablaufen zu lassen. Die Einplanung der Tasks geschieht

z.B. nach dem Rate Monotonic Scheduling Prinzip, das durch den POSIX.4 Standard unterstützt wird [Gal95].

Die *Application Tasks* greifen über das *virtual Device Interface* auf die Geräte zu.

Die konkurrente Abarbeitung von Tasks mit verschiedenen Steuerungsanwendungen erlaubt die Ansteuerung einer Fertigungszelle durch ein einziges Steuerungssystem (das die hier beschriebene Architektur umsetzt); d.h. das universelle Steuerungssystem ersetzt mehrere einzelne konventionelle Steuerungen, wie z.B. eine Robotersteuerung zusammen mit einer SPS (siehe Abschnitt 2.4.6.1).

- Offene Steuerungssysteme
  Die beschriebene Architektur eignet sich besonders für die Entwicklung offener Steuerungssysteme (siehe Abschnitt 2.4.5). Wird die virtuelle Geräteschnittstelle offengelegt, so können beliebige Steuerungssubsysteme, die auf diese Schnittstelle zugreifen, entwickelt werden.

- Unterstützung der busorientierten Gerätekommunikation
  Die Verwendung eines Feldbus zur Kommunikation mit den Automatisierungsgeräten ergänzt sich mit dem Konzept der konkurrenten *Application Tasks*. Durch einen Bus können eine sehr große Anzahl von Geräten mit der Steuerung verbunden werden (beim CANopen Protokoll werden bis zu 128 Busteilnehmer unterstützt [CAN95a]). Die Steuerungsarchitektur ermöglicht die Ausführung von fast beliebig vielen *Application Tasks*, die diese Geräte ansteuern.

4.2 Objektorientiertes Steuerungs-Framework

Das Steuerungs-Framework [SP00] ist im Rahmen der Portierung des Steuerungssystems von Solaris auf die Plattformen Linux und Windows NT mit der GNU Win 32 POSIX API entwickelt worden. Diese Plattformen sind wesentlich homogener (d.h. orientieren sich am POSIX-Standard) als spezielle Betriebssysteme wie z.B. das Echtzeitbetriebssystem RMOS von Siemens.
Das Framework umfaßt die Steuerungssubsysteme, das zentrale *Communication and Coordination Subsystem* sowie die Verbindung zu den Geräten. Die Flexibilität des Framework erstreckt sich auf drei Bereiche:

1. beliebige Steuerungssysteme (Robotersteuerung, SPS, NC und weitere)

2. Unterstützung von Betriebssystemen nach dem POSIX-Standard

3. Unterstützung unterschiedlicher Kommunikationsverbindungen zu den Automatisierungsgeräten

Das Framework ist entsprechend der in Abschnitt 2.2.4 aufgeführten Vorgehensweise entwickelt worden. Dabei ist es von dem in Abschnitt 3.3 beschriebenen objektorientierten Steuerungssystem abgeleitet worden.
Im folgenden sollen nun die flexiblen Elemente des Framework, deren White Box oder Black Box Charakter und ein Anwendungsbeispiel näher betrachtet werden.

4.2.1 Flexible Elemente des Framework

Folgende flexible Elemente ("Hot Spots", siehe Abschnitt 2.2.3.1) sind Teil des Steuerungs-Framework (siehe Abbildung 4.4). Die flexiblen Elemente des Framework werden sowohl durch reine Vererbung (Flexibility by Inheritance) als auch durch Komposition (Flexibility by Composition) realisiert (siehe Abschnitt 2.2.3.2).

1. Parametrisierbare vorgefertigte Steuerungssubsysteme (*Control Functionality*)
   Die existierenden Steuerungssubsysteme (die komplexen Subsysteme: *Robot Control SCARA*, *Transport System* und *Soft PLC*, sowie die einfachen Subsysteme: *Robot Drive*, *Motor* und *I/O*) können entsprechend den gesteuerten Geräten parametrisiert werden. Die Möglichkeiten zur Anpassung finden sich in Tabelle 4.1.

   Die Parametrisierung geschieht sowohl durch Vererbung als auch durch Komposition, wobei voreingestellte Werte für ein Standardgerät überschrieben werden können.

   Ein konkretes Beispiel für die Anpassung des Robotersteuerungs-Package durch Komposition [Pre97] findet sich in Abbildung 4.5. *SCARA Robot Control* ist die Schablonenklasse. Mittels Komposition werden hier die *Robot Arm Kinematics*, die *Trajectory Planning Interpolation* und die *Closed Loop Position Control* hinzugefügt. Alle drei Klassen wirken als Einschubklassen. (Die Funktionsweise von Schablonen- und Einschubklassen ist in Abschnitt 2.2.3.1 beschrieben.) Allerdings sind sie keine abstrakten Klassen, sondern besitzen voreingestellte Parameter, bzw. Algorithmen. Diese können von den erbenden Klassen (*concrete Robot Arm Kinematics*, *concrete Trajectory Planning Interpolation* und *concrete Closed Loop Position Control*) überschrieben bzw. modifiziert werden.

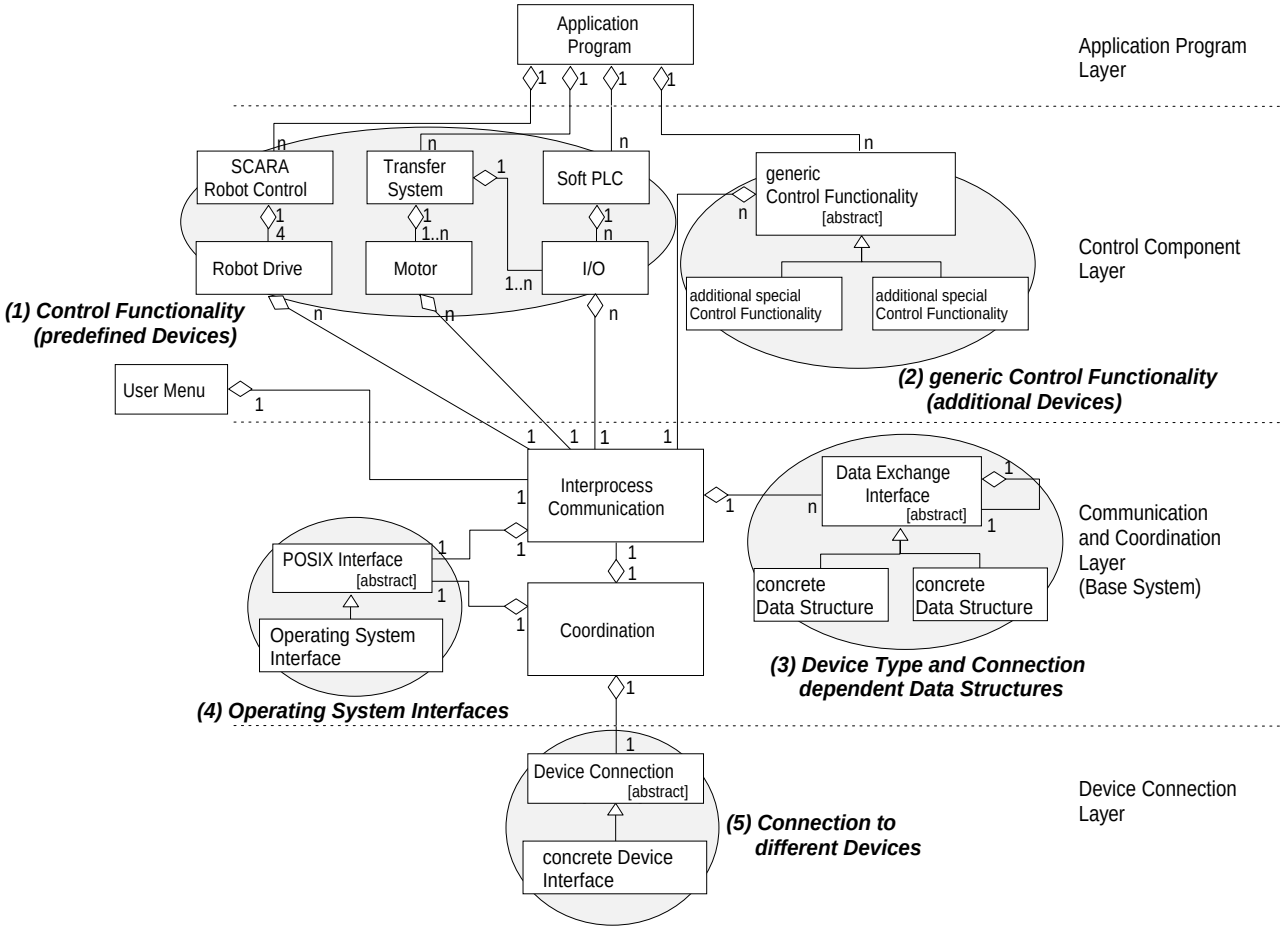


Abbildung 4.4: "Hot Spots" des Steuerungs-Framework

2. Generische Steuerungsfunktionen (*generic Control Functionality*)
   Zusätzlich zu den vorgegebenen Steuerungssubsystemen können noch weitere, neue Subsysteme in das Framework eingefügt werden. Diese Subsysteme dienen dann zur Ansteuerung von zusätzlichen Automatisierungsgeräten. Der Aufbau der Subsysteme ist nicht vordefiniert. Lediglich eine Schnittstelle zur Kommunikation über die *Interprocess Communication* wird in Form der *generic Device Control* vorgegeben. Die Flexibilität wird durch Vererbung erreicht.

3. Flexible Datenstrukturen zur systeminternen Kommunikation (*Device Type and Connection dependent Data Structures*)
   Die Datenstrukturen zur internen Kommunikation hängen von der gewählten Schnittstelle zur Kommunikation mit den Geräten ab. Die Datenstruktur enthält die Daten, die über diese Schnittstelle verschickt oder empfangen werden.

   Die Parametrisierung erfolgt durch Vererbung.

4. Flexible Betriebssystemschnittstelle (*Operating System Interface*)
   Der Datenaustausch und die Kommunikationsmechanismen hängen von dem gewählten Betriebssystem ab [SS95].

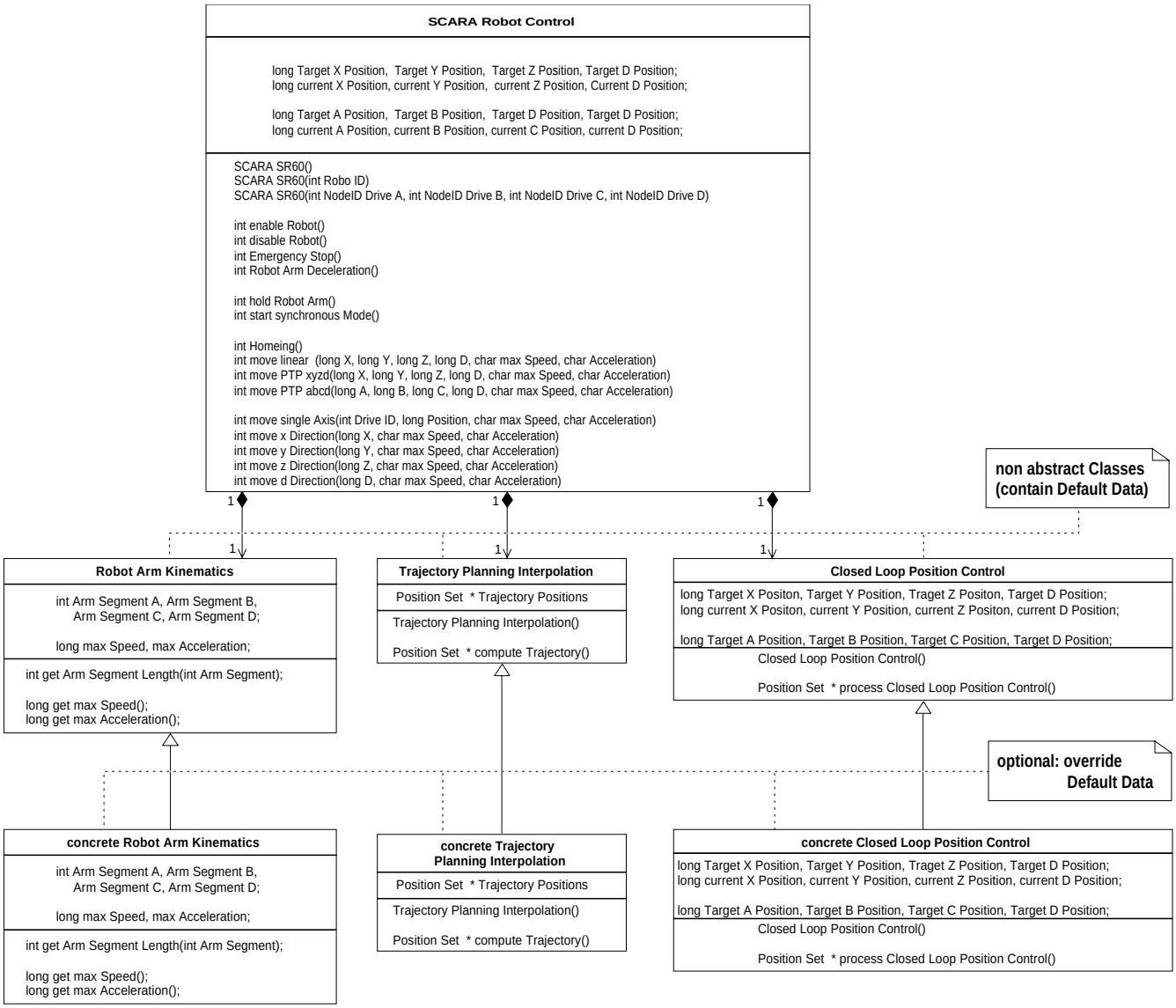


Abbildung 4.5: Parametrisierung des Robotersteuerungs-Package

Durch die vorgegebene abstrakte Schnittstelle für Betriebssystemaufrufe (*POSIX Interface*) kann auf das *Operating System Interface* zugegriffen werden. Diese Klasse realisiert die Aufrufe an das verwendete Betriebssystem.

Im Rahmen dieser Arbeit sind vorgefertigte *Operating System Interface* Klassen für die POSIX-orientierten Systeme Solaris, Linux und Windows NT (mit GNU Win 32 POSIX API von Cygnus) implementiert worden. Zusätzlich können weitere Klassen für andere POSIX konforme Betriebssysteme hinzugefügt werden. Die Flexibilität wird durch Vererbung erreicht.

5. Flexible Schnittstelle zur Kommunikation mit den Automatisierungsgeräten (*Connection to different Devices*)
   Je nach dem genutzten Betriebssystem und der jeweiligen Kommunikationsart (z.B. Schnittstellenkarte oder integrierte Schnittstelle) mit den Automatisierungsgeräten (bzw. deren Treiberschnittstellen) muß die Kommunikation angepaßt werden. Lediglich die Zugriffsarten (Lesen, Schreiben und Kontrolle der Hardware) ist vorgegeben.

Subsystem	Klasse	Parameter
Robot Control SCARA	*SCARA*	Längen der Armsegmente, Durchmesser des Arbeitsraums, maximale Geschwindigkeit und Beschleunigung
	Trajectory Planning Interpolation	Interpolationsarten (z.B. PTP, Linearfahrt), Anzahl der Bahnpunkte
	Closed Loop Position Control	Regelungsalgorithmen (P, PI, PD, PID)
Robot Drive	*Drive Dependent Converter*	Anpassung an das verwendete Kommunikationsprotokoll
Transfer System	*Transfer System*	Anzahl der digitalen Ein- und Ausgänge, Anzahl der Antriebe
Motor	*Motor Converter*	Anpassung an das verwendete Kommunikationsprotokoll
Soft PLC	*Soft PLC*	Anzahl der Eingänge und Ausgänge, maximale Zykluszeit
I/O	*I/O Module Converter*	Anpassung an das verwendete Kommunikationsprotokoll

Tabelle 4.1: Parametrisierbare vorgefertigte Steuerungssubsysteme

Durch Vererbung kann die Kommunikationsschnittstelle angepaßt werden.

4.2.2 White Box und Black Box Elemente

Wichtig für die Verwendung eines Framework ist das Wissen, welche Elemente White Box oder Black Box Charakter haben [2]. Das Steuerungs-Framework enthält wie die meisten Frameworks sowohl White Box als auch Black Box Elemente. Es wird daher als Grey Box Framework [FV98] bezeichnet.

- Black Box Elemente
  Als Black Box Elemente sind die parametrisierbaren vorgefertigten Steuerungssubsysteme, die flexiblen Datenstrukturen zur systeminternen Kommunikation und die flexible Betriebssystemschnittstelle zu betrachten.

  Die flexiblen Black Box Elemente der Steuerungssubsysteme sind die Steuerungsparameter und Steuerungsalgorithmen. Diese können entsprechend der Vorgaben in den Definitionsdateien (Header Files) der Einschubklassen modifiziert werden.

  Die Datenstrukturen können beliebig sein, da sie lediglich zum Weiterreichen von Daten verwendet werden. Die Strukturen sind nur von der gewählten Kommunikationsart mit den Geräten abhängig. Diese muß vom Benutzer ausgewählt werden. Die Funktionen zum Transport der Daten sind jedoch einheitlich und werden als Black Box gekapselt.

  Die Betriebssystemschnittstelle definiert in der Einschubklasse die Systemaufrufe, die die Steuerung benötigt (voreingestellt sind UNIX-Aufrufe). Der Anwender kann diese Aufrufe mit den Betriebssystemfunktionen des gewählten Systems überschreiben.

[2] Bei White Box Elementen muß ein Nutzer des Elements dessen Aufbau kennen. Bei Black Box Elementen ist dies nicht nötig (siehe Abschnitt 2.2.3.3).

- White Box Elemente
  Möchte der Anwender des Framework die flexible Schnittstelle zur Kommunikation mit den Geräten anpassen, so muß er das Kommunikationsprotokoll kennen. Aufgrund der großen Varianz der möglichen Kommunikationsvarianten (z.B. Kommunikation über einen Feldbus, Einzelverdrahtung, Verwendung einer Schnittstellenkarte oder integrierte Schnittstelle) kann keine geschlossene Architektur entwickelt werden, bei der nur wenige Parameter oder Methoden verändert werden müssen.

  Ebenso können die generischen Steuerungsfunktionen nicht als Black Box Elemente modelliert werden, da diese Steuerungssubsysteme viele unterschiedliche Geräte ansteuern sollen. Lediglich die Schnittstelle zur internen Kommunikation ist vorgegeben.

4.2.3 Anwendungbeispiel des Framework

Das Framework ist zur Entwicklung unterschiedlicher Systeme verwendet worden. Ein konkretes Beispiel für die Anwendung ist die Entwicklung eines universellen Robotersteuerungssystems unter Windows NT (mit GNU WIN 32 API). Da hier keine besonderen Geräte zusätzlich angesteuert werden sollen, wird auf die generischen Steuerungselemente verzichtet.
Für die Steuerung auf Basis eines Windows NT PCs kann bei folgenden flexiblen Elementen auf voreingestellte Methoden oder Daten zurückgegriffen werden:

- Die parametrisierbaren vorgefertigten Steuerungssubsysteme können verwendet werden, da nur die von diesen Subsystemen unterstützten Geräte (Roboterarm, digitale I/O Module und Transfersystem) gesteuert werden sollen.

- Als Datenstrukturen zur systeminternen Kommunikation können die Standardstrukturen verwendet werden. Diese sind in ihrer Voreinstellung für das Feldbusprotokoll CANopen geeignet.

Dagegen müssen die weiteren "Hot Spots" modifiziert werden:

- Die Betriebssystemaufrufe müssen auf die Windows NT Schnittstelle angepaßt werden. Dabei wird weitestgehend die POSIX konforme GNU WIN 32 Systemschnittstelle verwendet. Allerdings müssen auch spezifische Windows NT Funktionen aufgerufen werden (z.B. zur Datenübertragung mit Shared Memory).

- Die flexible Schnittstelle zur Kommunikation mit den Automatisierungsgeräten muß an die verwendete Schnittstellenkarte, bzw. deren Treiberschnittstelle, angepaßt werden. Da sich die Treiberschnittstelle der Karte unter Windows NT wesentlich von der einfachen UNIX-Schnittstelle unterscheidet, müssen größere Modifikationen durch das *concrete Device Interface* durchgeführt werden. Diese sind z.B. das Einfügen zusätzlicher Aufrufe zur speziellen Karteninitialisierung.

Zusätzlich muß bei diesem System die mangelnde Echtzeitfähigkeit von Windows NT beachtet werden (siehe Abschnitt 6.2.2).

4.3 Architektur komponentenbasierter Frameworks

Während die vorangegangenen Abschnitte ein Architekturmuster (Abschnitt 4.1) sowie ein Framework (Abschnitt 4.2) für eine universelle Steuerung beschreiben, baut dieser Abschnitt aus den hierauf gewonnenen Erfahrungen auf. In Verbindung mit den Erfahrungen zur Entwicklung von Frameworks aus weiteren Forschungsarbeiten anderer Universitäten ist eine Architektur für ein allgemeines komponentenbasiertes Framework abgeleitet worden. Diese Architektur spiegelt den Aufbau von Frameworks wider, der in allen betrachteten Frameworks zu erkennen ist.
Ähnlich wie Muster die Software-Entwicklung unterstützen, kann mit Hilfe dieser Architektur die Entwicklung von Frameworks in den verschiedensten Anwendungsbereichen unterstützt werden. Dabei ermöglicht der neue Ansatz die flexible Kombination von Komponenten zu einem Framework, bzw. lauffähigen System. Im Gegensatz zu den starren nur für eine vorgegebene Aufgabendomäne verwendbare Frameworks können komponentenbasierte Frameworks auch nach der Auslieferung an eine Vielzahl neuer Aufgabenstellungen angepaßt werden. Insbesondere wird eine Entwicklung von Frameworks unterstützt, die durch Standardkomponenten kundenorientiert zusammengesetzt werden können.
Im folgenden soll das allgemeine Architekturmodell vorgestellt werden. Anschließend wird dessen Anwendung zur Entwicklung eines konkreten komponentenbasierten Steuerungs-Frameworks dargestellt.

4.3.1 Aufbau des allgemeinen Architekturmodells

Dieses Architekturmodell ist auf Grundlage der Erfahrungen von unterschiedlichen, bestehenden Software-Systemen entwickelt und beim Bau von Frameworks in diesen Anwendungsgebieten getestet worden [PRST99]. Neben dem objektorientierten Steuerungssystem, bzw. Steuerungs-Framework, sind ein generisches Simulationssystem (von A. Telea, Technische Universität Eindhoven) und ein Modellierungs- und Simulationswerkzeug für elektronische Schaltungen (von D. Parsons, Universität Southampton) als Basis für die Entwicklung der Architektur des komponentenbasierten Framework zu Grunde gelegt worden.
Ziel der Architektur ist eine flexible Anordnung der Komponenten des Framework, d.h. die Komponenten werden in Komponentenkategorien klassifiziert, wobei deren Beziehungen vorgegeben werden. Dadurch können einzelne Komponenten innerhalb ihrer Kategorien beliebig ausgetauscht werden.
Diese einzelnen Komponententypen können von unterschiedlichen Gruppen von Personen, bzw. Firmen, wie halbfertige Produkte im Maschinenbau zugeliefert werden und nacheinander montiert werden [Cop99].
Die Komponenten werden in Schalen um eine zentrale Komponente (*Backbone Component*) angeordnet. Die einzelnen Komponentenkategorien sind:

- *Backbone Component*
  Diese Komponente kommt in einem Framework genau einmal vor. Die *Backbone Component* realisiert (bzw. kontrolliert) die zentralen Funktionen zur Kommunikation zwischen den anderen Komponenten. Damit steuert diese Komponente das Verhalten des Systems. Das Konzept von derartigen zentralen Subsystemen oder Komponenten wird vielfach in Frameworks angewendet [FS97]. Allerdings werden diese Subsysteme oder Komponenten nicht besonders identifiziert [PRST99].

- *Basic Components*
  Die *Basic Components* realisieren die grundsätzlichen Funktionen des Framework. Zu-

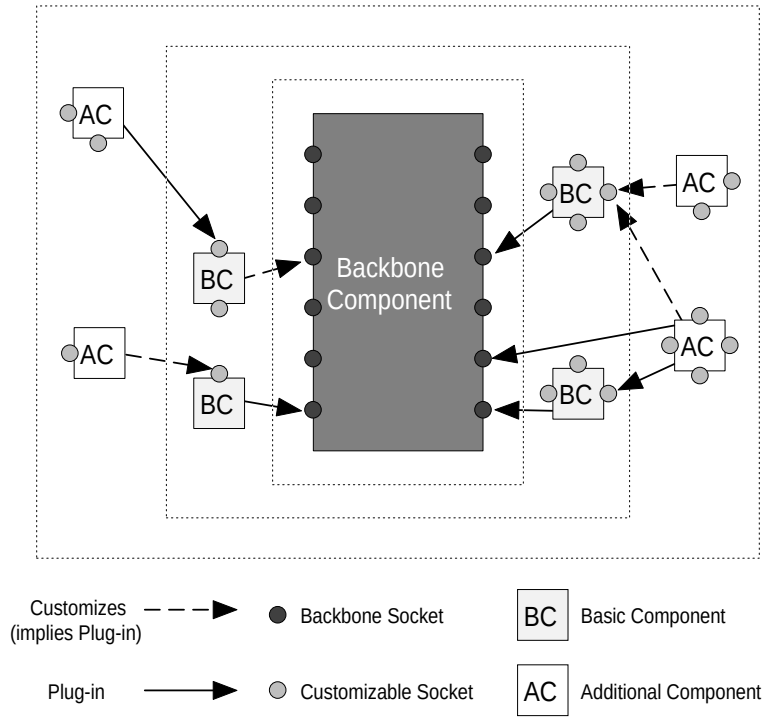


Abbildung 4.6: Allgemeine Architektur eines komponentenbasierten Framework

sammen mit der *Backbone Component* bilden sie ein rudimentäres Framework. Dieses definiert eine lauffähige und einsetzbare Basisversion.

- *Additional Components*
  Diese Komponenten bilden die Erweiterung eines Framework bestehend aus der *Backbone Component* und den *Basic Components*. Im Gegensatz zu den *Basic Components* sind die *Additional Components* nicht zwingender Bestandteil eines Frameworks. Die *Additional Components* können auch erst nach Auslieferung eines Framework nachgeliefert werden. Dadurch können Frameworks besser an neue Anforderungen angepaßt werden.

Die Schnittstellen zwischen den Komponenten des Framework müssen beim Entwurf in einem Konfigurations-Repository definiert werden [CE99]. Die genaue Beschreibung der Schnittstellen ermöglicht, daß nachträglich weitere Komponenten in das Framework eingebunden werden können.
Zur Bindung von Komponenten werden zwei Arten von Beziehungen verwendet: *Plug-in* und *Customizes*:

- Die *Plug-in*-Beziehung [ND95] beschreibt die Nutzung einer Komponente durch eine andere. Dabei kann die Initiative für eine Kommunikation zwischen den Komponenten von der einen oder von beiden Komponenten ausgehen.

  Diese Beziehungen zwischen den Komponenten können unterschiedlich realisiert werden: z.B. als Komponentenbindung mit CORBA bzw. COM oder als Aggregation. Dabei können die Komponenten dynamisch oder statisch gebunden werden [Szy97, PRST99].

- Der zweite Beziehungstyp *Customizes* entspricht der bei Frameworks verwendeten Anpassung [Pre97, Sch97a] (siehe Abschnitt 2.2.3.1). Eine Komponente modifiziert die von ihr verwendete Komponente.

Um aus einem komponentenbasierten Framework ein vollständiges System zu bauen, müssen zusätzliche Komponenten (die als *Applications* bezeichnet werden [PRST99]) hinzugefügt werden. Diese können zwar mit allen Komponenten über eine *Plug-in*-Beziehung verbunden werden, allerdings können sie nur die *Basic Components* und *Additional Components* anpassen (*Customizes*-Beziehung). Die *Backbone Component* darf nicht verändert werden, da sonst die grundsätzliche Funktion des Framework verändert werden könnte.

4.3.2 Komponentenbasiertes industrielles Steuerungs-Framework

Das Aufgabenfeld der industriellen Steuerungssysteme eignet sich besonders zum Bau von komponentenbasierten Systemen, da diese Steuerungen modular aufgebaute Gerätegruppen kontrollieren. Die modulare Anordnung der Automatisierungsgeräte muß nur auf den modularen Aufbau der Komponenten der Steuerung abgebildet werden. [Spe00]

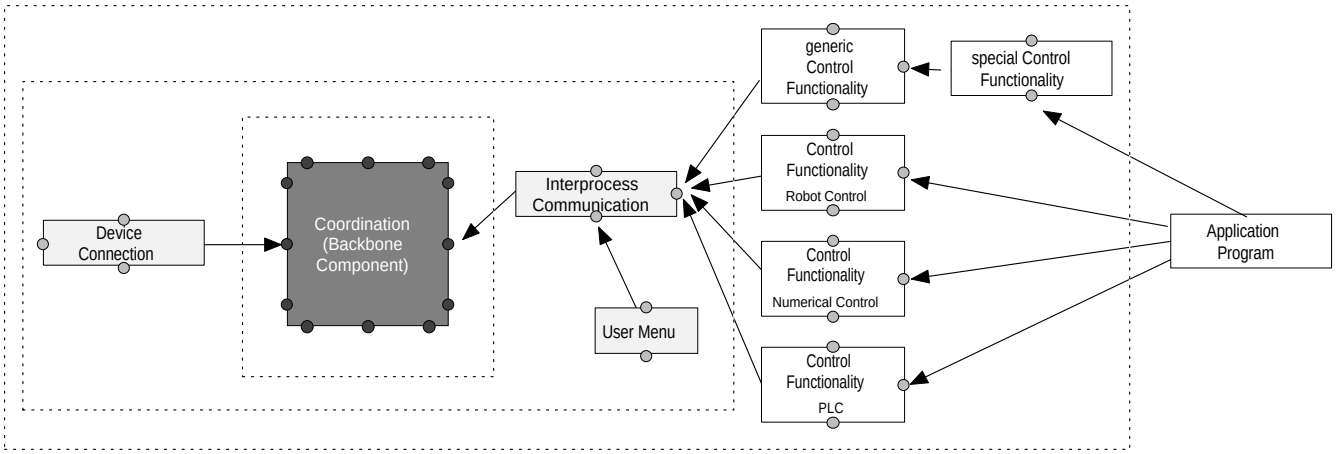


Abbildung 4.7: Erster Ansatz für ein komponentenbasiertes Steuerungs-Framework

4.3.2.1 Erster Ansatz

Eine Steuerung (bzw. ein Steuerungs-Framework für industrielle Steuerungen) ist aus Software-Komponenten, die den Hardware-Komponenten entsprechen, aufgebaut. Ein erster vereinfachter Ansatz (siehe Abbildung 4.7) besteht aus folgenden Komponenten:

- Die *Backbone Component* übernimmt die Kontrolle über das zeitliche Verhalten der Steuerung und die interne Datenkommunikation. Die Steuerungsfunktionen (und damit der Datenaustausch) werden üblicherweise zyklisch abgearbeitet [RNS93].

  Die eigentlichen Mechanismen zum Datenaustausch und die Steuerungsfunktionalität sind nicht Teil der *Backbone Component*, sondern werden von dieser zeitlich kontrolliert.

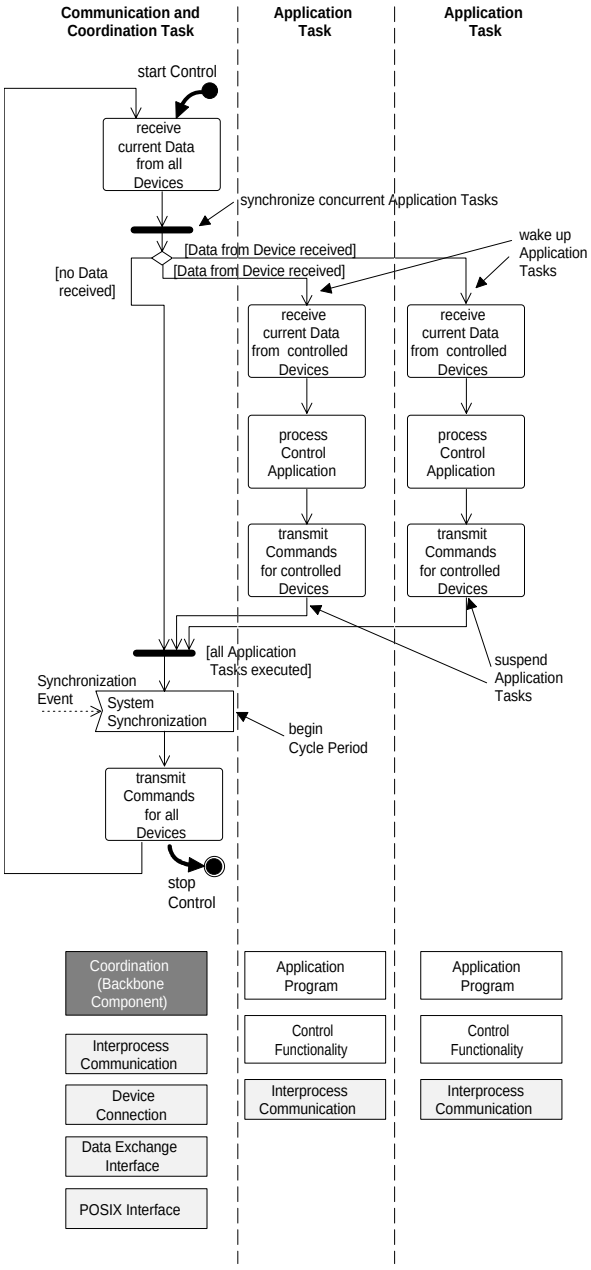


Abbildung 4.8: Komponenten der Tasks

- Die zentralen Steuerungsalgorithmen und die Methoden zum Datenaustausch werden als *Basic Components* realisiert, die dem System modular hinzugefügt werden. Mit der *Backbone Component* bilden sie ein Framework, das z.B. zur rudimentären manuellen Kommunikation mit Automatisierungsgeräten genutzt werden kann.

- Die Steuerungsfunktionalität wird durch *Additional Components* hinzugefügt. Damit kann das Framework beispielsweise zu einer industriellen Robotersteuerung ausgebaut werden. Dazu sind allerdings Anpassungen der *Basic Components* und *Additional Components* nötig. Diese umfassen unter anderem Angaben über den Robotertyp, die Längen der Armsegmente und die eingebauten Roboterantriebe. Ebenso müssen die Zykluszeit und die maximalen Antwortzeiten als Parameter vorgegeben werden.

  Soll das Steuerungs-Framework um zusätzliche Steuerungsfunktionen (z.B. speicherprogrammierbare Steuerung zur Kontrolle der peripheren Geräte des Roboters) erweitert werden, so können diese Funktionen ebenfalls in Form von *Additional Components*

realisiert werden.

Zur Kopplung von Komponenten werden im Steuerungs-Framework Aggregationen (*Plug-in*) und Hot-Spot-Anpassungen (*Customizes*) verwendet. Komponenten mit (zusätzlichen) Steuerungsfunktionen werden über Shared Memory dynamisch in die Steuerung eingebunden.

Die Aufteilung der Komponenten auf einzelne Tasks wird in Abbildung 4.8 dargestellt.
Das Verhalten der Tasks entspricht den durch das Architekturmuster (siehe Abschnitt 4.1.1, dynamisches Verhalten) beschriebenen dynamischen Abläufen.
Die *Communication and Coordination Task* besteht aus der *Backbone Component* und jeweils einer Instanz aller *Basic Components*.
Die *Application Tasks* laufen konkurrent ab. Diese Tasks beinhalten jeweils eine Instanz der *Interprocess Communication* (*Basic Component*) sowie *Additional Components* als Steuerungskomponenten.

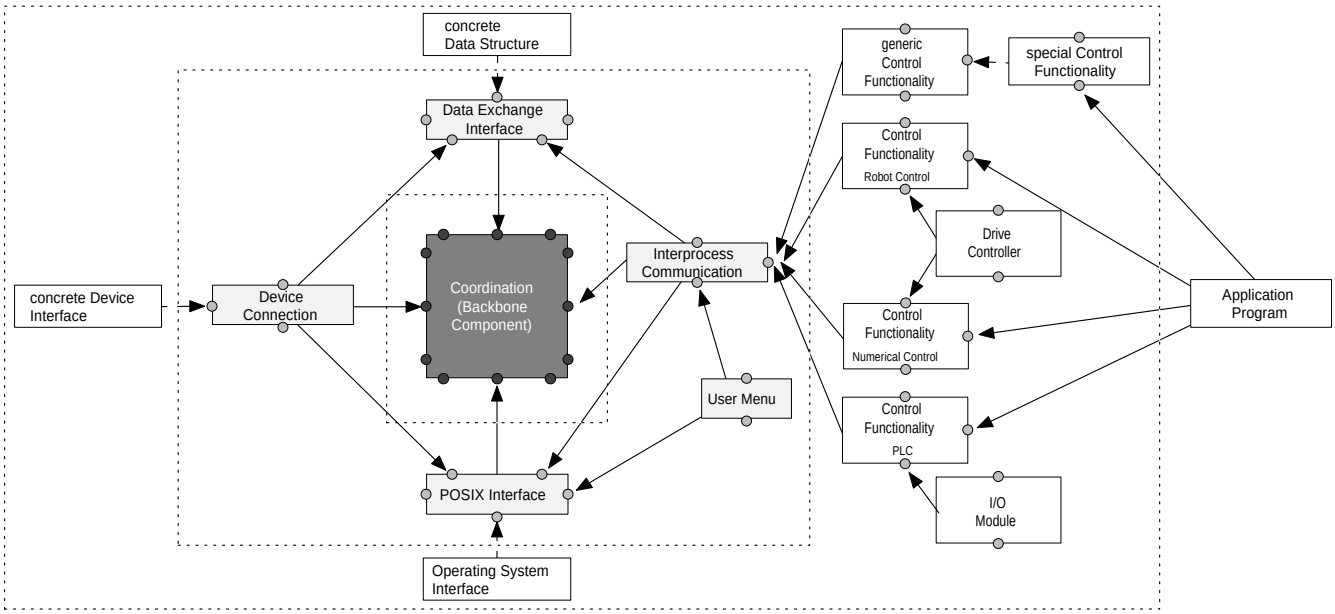


Abbildung 4.9: Vollständiges komponentenbasiertes Steuerungssystem

4.3.2.2 Vollständiges Komponentensystem

Während der vorhergehende Abschnitt eine erste, sehr allgemeine Beschreibung eines komponentenbasierten Steuerungs-Framework enthält, wird hier das vollständige komponentenbasierte Framework vorgestellt. Beim Bau des Systems entsprechend des neuen, in Abschnitt 4.3.1 eingeführten allgemeinen Architekturmodells zur Entwicklung komponentenbasierter Frameworks werden jedoch alle Erfahrungen und gewonnenen Anforderungen aus Analyse und Design des objektorientierten Steuerungssystems sowie des Architekturmusters und des Framework berücksichtigt. Das komponentenbasierte Steuerungs-Framework wird in Abbildung 4.9 dargestellt.

Insbesondere werden die im Abschnitt 4.2.1 eingeführten flexiblen Elemente des objektorientierten Framework übernommen. Diese sind parametrisierbare vorgefertigte Steuerungskomponenten, flexible Datenstrukturen zur systeminternen Kommunikation, eine flexible Betriebssystemschnittstelle und eine flexible Schnittstelle zur Kommunikation mit den Geräten. Daneben kann eine generische Steuerungskomponenten beliebig angepaßt werden.

4.4 Bewertung der Ansätze zur Wiederverwendung

Im folgenden sind die drei Ansätze zur Wiederverwendung gegenübergestellt. Das Architekturmuster dokumentiert die Erfahrungen aus der objektorientierten Steuerungsentwicklung auf höherer Abstraktionsstufe. Diese Erfahrungen stellen eine Anleitung zu einem guten Design für Steuerungssysteme in Form einer wiederverwendbaren Architektur zur Verfügung. Demgegenüber bildet das nach konventioneller Vorgehensweise gewonnene Framework diese Erfahrungen zusätzlich auf wiederverwendbaren Code ab.
Das Architekturmuster und das objektorientierte Framework sind eine Anwendung bekannter Vorgehensweisen in einem Anwendungsgebiet (industrielle Steuerungen), in dem diese Software-Engineering Mechanismen bisher nicht zum Einsatz gekommen sind. Demgegenüber beschreibt der dritte Ansatz mit dem Architekturmodell für komponentenbasierte Frameworks einen eigenen, neuen Ansatz. Nach diesem Vorbild können komponentenbasierte Frameworks nicht nur im Anwendungsgebiet der industriellen Steuerungen entwickelt werden.

Konzept	**wiederverwendbare Elemente**	**Entwicklungsaufwand**	**Erweiterbarkeit**
Architektur-muster	Architektur (enthält keinen Code)	geringer Aufwand, Verallgemeinerung einer bestehenden Architektur	Änderungen im Rahmen der Verallgemeinerung
objektorientiertes Framework	vorgefertigter Code mit vordefinierten flexiblen Elementen	hoher Aufwand durch Vorbestimmung aller flexiblen Elementen	nur im Rahmen der Vorausplanung erweiterbar
komponenten-basiertes Framework	Architekturmodell zur Anordnung der Komponenten, vorgefertigter Code in Komponenten gekapselt	Aufwand durch die Definition der Komponentengrenzen und der Kommunikation zwischen den Komponenten	flexible Erweiterbarkeit durch Hinzufügen oder Tausch von Komponenten

Tabelle 4.2: Bewertung der Ansätze zur Wiederverwendung

Die Wiederverwendung durch Anwendung des in dieser Arbeit entwickelten Architekturmusters für Steuerungssysteme, die Wiederverwendung durch Einsatz des objektorientierten Frameworks sowie drittens durch Verwendung des komponentenbasierten Frameworks (das im Rahmen dieser Arbeit für Steuerungssysteme durch Ableitung aus dem allgemeinen Architekturmodell entstanden ist) sind in Tabelle 4.2 mit ihren wichtigsten Merkmalen gegenübergestellt. Dies sind die Elemente, die wiederverwendet werden können, der Aufwand, der Entwicklung des Architekturmusters, des objektorientierten Frameworks und des komponentenbasierten Frameworks sowie das Merkmal der Erweiterbarkeit des jeweiligen Konzepts.

Weitere Eigenschaften der einzelnen Ansätze sind:

- Architekturmuster
  Das Architekturmuster kann praktisch für alle möglichen Systemplattformen verwendet werden. Es kann sogar zur Beschreibung des Aufbaus des objektorientierten und des komponentenbasierten Framework herangezogen werden.

Der Nachteil dieser universellen Anwendbarkeit ist allerdings, daß der Grad der Wiederverwendung auf das reine Architekturmuster beschränkt ist. Code wird nicht wiederverwendet.

- objektorientiertes Framework
  Gegenüber dem Architekturmuster stellt das objektorientierte Framework bereits vorgefertigten Code zur Verfügung.

  Eine negative Konsequenz hieraus ist jedoch, daß dieses Framework nicht alle Anwendungsfälle abdecken kann (z.B. Steuerungsanwendungen auf nicht-Standardsystemen wie Industrie-PCs mit besonderen Echtzeitbetriebssystemen).

- komponentenbasiertes Framework
  Das komponentenbasierte Framework bietet von den vorgestellten Konzepten den besten Kompromiß zwischen Plattformunabhängigkeit und Wiederverwendung von Architektur und Code. Im Vergleich zu einem konventionellen objektorientierten Framework erweist sich die Entwicklung entsprechend des beschriebenen Architekturmodells für Frameworks als einfacher. Im Gegensatz zum objektorientierten Framework müssen hier nicht alle vorkommenden flexiblen Elemente (Hot Spots) im voraus bestimmt und definiert werden.

  Eine komponentenbasierte Architektur erlaubt es, daß nachträglich weitere Komponenten hinzugefügt werden können, sofern sich diese Komponenten über die vordefinierten Standardschnittstellen des Framework einbinden lassen. Entsprechend dem Vorbild der Hardware-Komponenten in der Automatisierungstechnik können offene Standards zur Beschreibung der Funktionen und Schnittstellen definiert werden. Diese unterstützten die Entwicklung, Anpassung und Wiederverwendung komponentenbasierter Frameworks [Spe00].

  Nach der Offenlegung von Standards für Automatisierungkomponenten sind innerhalb relativ kurzer Zeit eine Vielzahl von Komponenten auf dem Markt angeboten worden. Eine vergleichbare Entwicklung ist auch im Bereich der Software-Komponenten möglich.

  Gegenüber dem in [Sch97a] beschriebenen Vorgehen zur Entwicklung von Frameworks durch Generalisierung eines bestehenden objektorientierten Systems (siehe Abschnitt 2.2.4) hat das komponentenbasierte Framework den Vorteil, daß kein System zur Ableitung vorhanden sein muß. Dadurch können die einzelnen Komponenten inkrementell entwickelt und zu einem wachsenden Framework hinzugefügt werden (analog dem “Piecemeal Growth” Ansatz in [Cop99]).

Kapitel 5

Simulations- und Monitoring-Werkzeug

Unter Verwendung des in Abschnitt 4.3 beschriebenen und gemäß des allgemeinen Architekturmodells realisierten komponentenbasierten Steuerungs-Framework ist das Simulations- und Monitoring-Werkzeug RoboSiM [Spe98, Spe99] entwickelt worden. Zum vorhandenen Steuerungssystem sind zusätzliche Simulations- und Beobachtungskomponenten hinzugefügt worden.
Im weiteren wird ein Überblick über die Architektur von RoboSiM gegeben. Darauf folgt eine kurze Beschreibung der graphischen Benutzeroberfläche sowie die Auflistung der besonderen Eigenschaften von RoboSiM gegenüber bestehenden Systemen.

5.1 Aufbau von RoboSiM

RoboSiM umfaßt das in Abschnitt 4.3 eingeführte komponentenbasierte Framework. Allerdings ist das ursprüngliche Steuerungs-Framework um folgende Komponenten erweitert (siehe Abbildung 5.1):

- *Simulation Subsystem*
  Dieses Subsystem beinhaltet die Methoden zur Simulation der Geräte. Da zwei Typen von Geräten (SCARA Roboterarm und digitale Geräte) unterstützt werden, umfaßt dieses Subsystem für jede der beiden Gerätearten eine besondere Komponente.

  Das *Simulation Subsystem* kommuniziert über die *Interprocess Communication and Synchronization* mit dem *Communication and Coordination Subsystem*. Im Gegensatz zu bestehenden Simulationssystemen kann RoboSiM gleichzeitig mit realen Geräten über den Feldbus CAN kommunizieren.

  Die simulierten Automatisierungsgeräte antworten mit den gleichen Rückmeldungen entsprechend des Feldbusprotokolls wie die realen Geräte. Reale und virtuelle Geräte werden eindeutig an ihrer Identifikationsnummer unterschieden.

- *Interconnection Task*
  Die zentrale Komponente (*Interconnection Component*) dieser Task ist als Spezialisierung der *generic Control Functionality* Komponente realisiert. Die *Interconnection Task* verwendet zusätzlich die Komponenten zur Steuerung eines Roboterarms (*Robot Control* und *Drive Controller*). Sie ist von der Einordnung in die Hierarchie (siehe

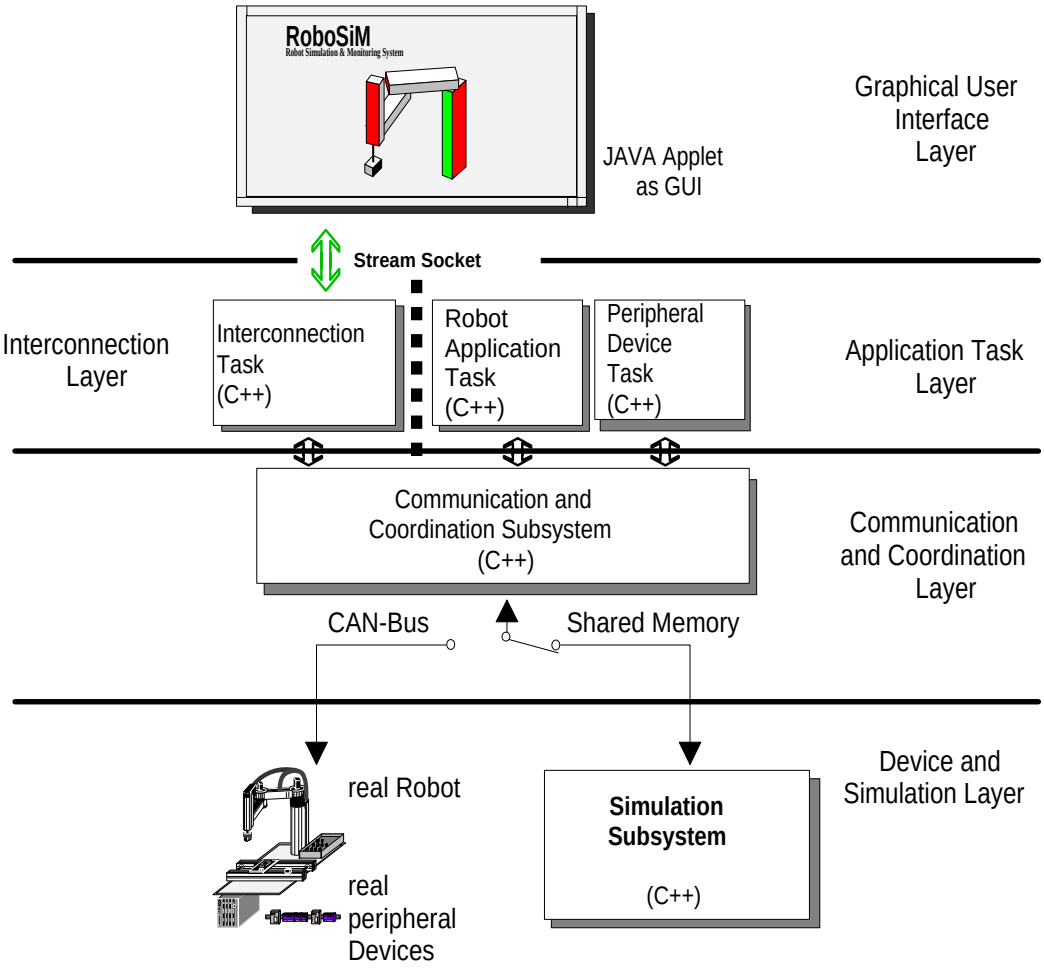


Abbildung 5.1: Überblick über die Architektur von RoboSiM

Abbildung 5.1) auf der gleichen Ebene wie die Tasks mit Steuerungsfunktionen (*Application Tasks*) angeordnet. Allerdings wird die Ebene aufgrund des Unterschieds in der Aufgabenstellung als *Interconnection Layer* bezeichnet.

Die Aufgabe der Verbindungskomponente besteht darin, das eigentliche Steuerungssystem mit der graphischen Java Benutzerschnittstelle zu verbinden. Diese Verbindung wird über eine Socket realisiert [San95]. Damit kann die graphische Benutzerschnittstelle auch auf einem entfernten Rechner ablaufen.

Die *Interconnection Component* sendet an die Benutzerschnittstelle zyklisch den aktuellen Zustand der gesteuerten Geräte (z.B. die aktuelle Position des Roboterarms) und empfängt die Befehle des Benutzers an die Steuerung, wie z.B. Kommandos zum Start eines Applikationsprogramms oder Befehle, mit denen die Roboterkinematik manuell verfahren wird.

- *Graphical User Interface*
  Die graphische Benutzerschnittstelle visualisiert die aktuelle Position des Roboterarms. Ebenso wird der Zustand des Roboters angezeigt, z.B. daß ein Applikationsprogramm abgearbeitet wird oder eine Störung aufgetreten ist. Im Gegenzug kann der Anwender über diese Schnittstelle auf die Robotersteuerung, bzw. die Roboterkinematik,

einwirken.

Das *Graphical User Interface* kann sowohl reale als auch virtuelle Roboter (gleichzeitig) darstellen.

5.2 Graphische Benutzeroberfläche

Die graphische Benutzerschnittstelle zeigt zum einen die Position und den Zustand des Roboters an, zum anderen kann über sie vom Benutzer auch auf die Robotersteuerung zugegriffen werden. Da mit dem graphischen Interface sowohl reale als auch simulierte Roboterarme visualisiert werden, können reale und simulierte Roboterkinematiken direkt miteinander verglichen werden.

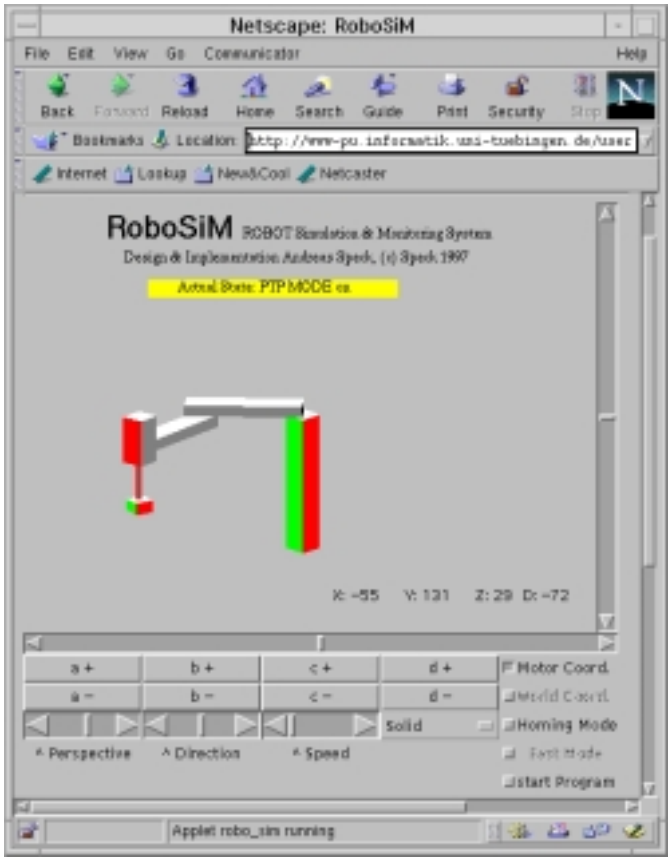


Abbildung 5.2: RoboSiM Java 1.1 Version

Von der graphischen Benutzerschnittstelle existieren zwei Varianten: Die erste Version basiert auf Java 1.1 und nutzt dessen graphische Möglichkeiten für eine eingeschränkte räumliche Darstellung des Roboters [Spe99]. Die zweite Schnittstelle ist mit Java 3D entwickelt worden und präsentiert einen sehr realistisch wirkenden dreidimensionalen Roboterarm (siehe Anhang B). Außer in der Qualität der 3D-Visualisierung bestehen jedoch keine Unterschiede in der Funktionalität der beiden Oberflächen [SK99].
Im Gegensatz zu Visualisierungen, die auf *X Window System* oder *OSF Motif* basieren, kann ein Java-System relativ plattformunabhängig eingesetzt werden. Die Java 1.1 Version ist sehr einfach gehalten. Damit kann sie auch auf weniger leistungsfähigen Rechnern ablaufen.

Die Java-Visualisierung kann auch auf einem anderen Rechner, der mit der Steuerung über ein TCP/IP-Netz verbunden ist, ausgeführt werden. (Ähnliche Ansätze zum Aufbau eines netzbasierten Steuerungssystems sind in [KGL+97] beschrieben.)
Das RoboSiM GUI (siehe Abbildung 5.2) besteht aus einer Darstellung des Roboterarms und einem Kommandofeld mit Buttons und Scrollbars. Wie bei anderen Simulationssystemen [LW95] kann die Roboterkinematik entweder als Vollbild oder als Drahtgittermodell dargestellt werden. Die Koordinaten des Werkzeugflansch am Roboterarm (Tool Center Point, TCP) werden zusätzlich angezeigt.
Im Kommandofeld kann der Benutzer interaktiv den Roboter bewegen oder Roboteranwendungsprogramme starten oder beenden. Darüberhinaus kann der Blickwinkel und die Blickrichtung auf den Roboterarm verändert werden.

5.3 Anwendung von RoboSiM im Vergleich zu bestehenden Systemen

RoboSiM kann für zwei unterschiedliche Einsatzzwecke (bzw. deren Kombination) verwendet werden: die Simulation virtueller Roboterarme und das Monitoring von realen Geräten.
Bei der reinen Simulation hat RoboSiM gegenüber anderen Simulationssystemen den Vorteil, daß sie auf der realen Steuerungsplattform ausgeführt wird. Damit kann das Zeitverhalten des Steuerungsrechners mit in die Simulation einbezogen werden. Die meisten Robotersimulationssysteme, wie z.B. das Robotic HyperBook (RHB) [MM95] (Lehrsystem) oder die Off-Line-Programmierungssysteme ROPES [AS94], MOSES [Bic94a] oder auch Systeme wie RobCAD (sowohl für die Lehre als auch zur Off-Line-Programmierung geeignet) und das 3D Bewegungssimulationssystem USIS (Universal SImulation System) [KR94] laufen auf speziellen Rechnern ab (z.B. Graphikrechner). RoboSiM erlaubt das Off-Line-Programmieren von Roboterarmen (siehe Abschnitt 2.4.2.4), wobei auf alle Informationen der Robotersteuerungen (z.B. Konfigurationsdaten) zugegriffen werden kann.
Das Monitoring von Roboterarmen kann direkt an der Robotersteuerung oder an einem entfernten Rechner (Verbindung z.B. über Ethernet) geschehen. Die Robotersteuerung dient zugleich als Server für die Visualisierung. Dies unterscheidet RoboSiM von anderen Monitoring-Systemen, bei denen der Zustand der Robotersteuerung nur mittelbar von vorgeschalteten Rechnern exportiert wird [Sto92, TS93, Tay99, TT95, GM96].
Die Gerätevisualisierung von RoboSiM mittels Java benötigt im Gegensatz zur Übertragung von Videobildern eine wesentlich geringere Bandbreite. Dabei kann die Aussagekraft der Java-Darstellung ein Videobild sogar übertreffen. Durch die Java-Visualisierung können sehr leicht zusätzliche Informationen (z.B. Werte taktiler Sensoren) angezeigt werden, die in einem Videobild nicht enthalten sind.
RoboSiM ermöglicht die Kombination von realen und virtuellen Geräten. Damit unterscheidet sich RoboSiM von praktisch allen Robotersimulationssystemen [TS93, AS94, KR94, MM95, Spe98]. Mehrere simulierte und reale Roboterarme sowie deren Interaktionen können dargestellt werden. Diese Fähigkeit ist besonders für den Unterricht an Robotern sehr interessant, da sich die Visualisierung virtueller und realer Geräte in ihrer Darstellung nicht unterscheidet und damit der Übergang von einer virtuellen Trainingsumgebung zu realen Roboterarmen und Geräten sehr leicht fällt.

Kapitel 6

Implementierung

In den vorherigen Abschnitten sind unterschiedliche Konzepte zur Entwicklung von industriellen Steuerungssystemen vorgestellt worden. Entsprechend dieser Konzepte sind mehrere Implementierungen (Steuerungssysteme auf Basis des Architekturmusters, des objektorientierten Framework und des Architekturmodells eines komponentenbasierten Framework) auf unterschiedlichen Plattformen entstanden. Bei der Codierung werden Idiome verwendet (siehe Abschnitt 2.1.2.1), wobei insbesondere auf [HN92] zurückgegriffen wird.
Im folgenden werden zunächst die in allen Implementierungen verwendeten Konzepte zur Realisierung des Multitasking und zur Synchronisation der Tasks vorgestellt. Anschließend folgt die Beschreibung der zentralen Implementierungsmerkmale einiger exemplarischer Realisierungen des Steuerungssystems auf verschiedenen Rechnerplattformen sowie deren Leistungsangaben.

6.1 Konzepte zum Multitasking und zur Synchronisation

In diesem Abschnitt werden die grundlegenden Konzepte zur Realisierung des Multitasking sowie der Synchronisation und des Scheduling der parallelen Tasks vorgestellt.

6.1.1 Multitasking

Multitasking kann auf unterschiedliche Art und Weise realisiert werden: durch parallele Prozesse oder Threads, bzw. durch eine Kombination aus parallelen Prozessen und Threads (siehe Abbildung 6.1). Alle drei Varianten sind Implementierungen desselben Design (mit den gleichen Klassen und Kollaborationen).
Allerdings muß für die Kombination aus parallelen Prozessen und Threads das *Interprocess Communication* Package um einige wenige Methoden erweitert werden, da in diesem Fall sowohl zwischen Prozessen als auch zwischen Threads kommuniziert wird.
Bei allen Varianten ist der Datenaustausch zwischen den Tasks (Prozesse oder Threads) durch Shared Memory realisiert.
Die Unterschiede zwischen den Ansätzen sind (siehe Abbildung 6.1):

1. Applikationen als parallele Prozesse:
   Bei dieser Implementierungsvariante laufen die *Application Tasks* in jeweils eigenständigen Prozessen und damit in eigenen Adreßräumen. Die *Communication and Coordination Task* synchronisiert die Applikationsprozesse.

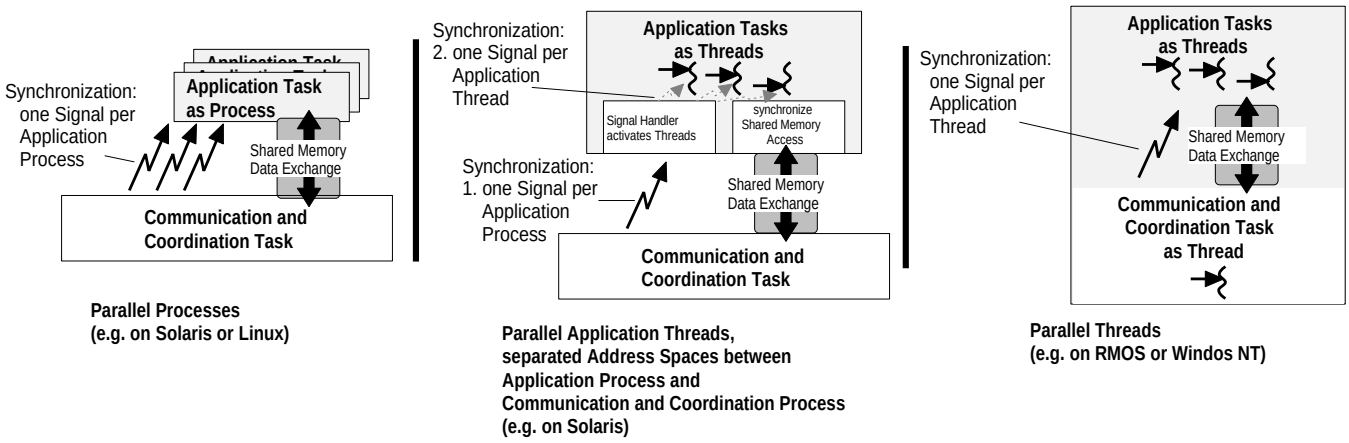


Abbildung 6.1: Parallele Applikationsprozesse, Kombination aus parallelen Prozessen und Threads und reines Multi-Threading

Die Vorteile paralleler Prozesse sind die einfachere Steuerungsentwicklung, da die Komplexität, die sich durch den Einsatz von Threads ergibt, vermieden wird. Darüberhinaus können prozeßbasierte Steuerungen auch auf Betriebssysteme portiert werden, die Threads nicht unterstützen.

Nachteilig wirkt sich demgegenüber der größere Aufwand für den Kontextwechsel von Prozessen aus.

Dieser erste Ansatz ist auf den UNIX-Plattformen Solaris und Linux realisiert.

2. Kombination aus parallelen Prozessen und Threads (Multi-Threaded Applikationen): Mehrere *Application Tasks* laufen als Threads innerhalb eines Prozesses ab. Dazu muß die Klasse *Task Synchronization* des *Interprocess Communication* Package entsprechend erweitert werden.

   Zum einen müssen die Applikations-Threads zyklisch ablaufen können (Blockieren und Aufwecken der einzelnen Threads). Zum anderen muß der per-Prozeß Mechanismus Shared Memory vor dem gleichzeitigen Zugriff durch mehrere Threads eines Prozesses geschützt werden.

   Gegenüber der prozeßbasierten Variante haben Threads den Vorteil, daß der Aufwand für einen Kontextwechsel wesentlich geringer ist. Dies ist vorallem vorteilhaft, wenn viele Steuerungs-Threads auf einem System ablaufen. Allerdings bleiben die Applikations-Threads in einem eigenen Adreßraum getrennt von der zentralen *Communication and Coordination Task*. Damit sind die Funktionen dieser Task vor Manipulationen durch den Anwender besser geschützt.

   Der Nachteil der Kombination von Prozessen und Threads ist der wesentlich größere Aufwand zur Realisierung. Die Kommunikation sowohl zwischen Prozessen als auch Threads muß entwickelt werden. Darüberhinaus müssen beide Kommunikationsarten koordiniert werden.

   Dieser Ansatz der Kombination ist unter dem Betriebssystem Solaris untersucht worden. Eine nennenswerte Verringerung der Systemlast hat sich allerdings nicht ergeben. Threads werden sinnvollerweise bei Rechnern verwendet, deren Leistungsgrenze erreicht ist. Bei der verwendeten, leistungsfähigen Workstation (SPARC Workstation,

siehe Abschnitt 6.3) ist dies mit einer Auslastung von 15 % nicht der Fall.

3. reines Multi-Threading:
   Diese Lösung entspricht weitestgehend dem Konzept der getrennten Prozesse. Allerdings mit dem Unterschied, daß Threads anstelle von Prozessen verwendet werden. Alle Kommunikationsmechanismen sind hier per-Thread Funktionen.

   Dieser Ansatz hat den Vorteil, daß er den geringsten Aufwand beim Kontextwechsel zwischen den einzelnen Threads verursacht. Allerdings laufen die Applikationen und die *Communication and Coordination Task* nicht in getrennten Adreßräumen ab. Daher kann die *Communication and Coordination Task* vom Anwender verändert werden.

   Diese Version des Steuerungssystems ist unter RMOS und Windows NT erprobt worden.

Neben einem Applikationsprozeß mit Applikations-Threads (entsprechend der Variante 2) oder mehreren Applikationsprozessen mit Applikations-Threads können gleichzeitig einzelne Applikationsprozesse (Variante 1) im Steuerungssystem ablaufen, d.h. ein Hybridansatz zwischen Variante 1 und 2 ist möglich.

6.1.2 Synchronisation und Scheduling der Tasks

In diesem Abschnitt wird das Vorgehen bei der Synchronisation und das Scheduling der Prozesse beschrieben.

- **Synchronisation:**
  Die Synchronisation der Tasks ist zweistufig. Die *Communication and Coordination Task* des industriellen Steuerungssystems läuft zyklisch ab, d.h. wird zyklisch synchronisiert. Diese wiederum synchronisiert dann die *Application Tasks* (siehe Abbildung 6.2).

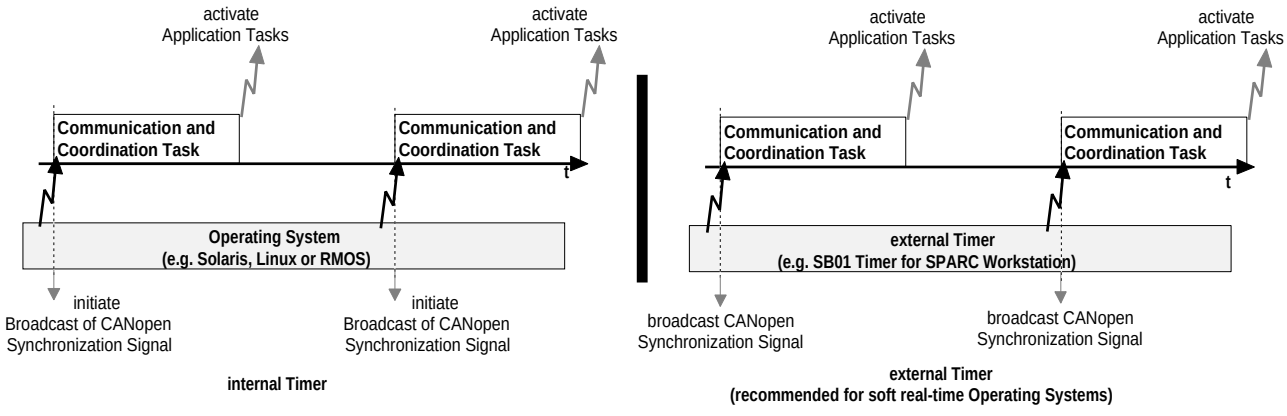


Abbildung 6.2: Synchronisation durch den Zeitgeber des Betriebssystems und Synchronisation durch einen externen Zeitgeber

Die Synchronisation der *Communication and Coordination Task* kann entweder durch einen Zeitgeber des Betriebssystems (wie z.B. bei dem Echtzeitbetriebssystem RMOS) oder durch einen externen Timer (z.B. mit dem externen Timer der SB01 Schnittstellenkarte) geschehen.

Bei der Synchronisation durch den externen Timer der SB01 Schnittstellenkarte für das SPARC System wird die *Communication and Coordination Task* von den Signalen der Treiber-Software der CAN-Schnittstellenkarte angestoßen. Die Grundeinstellung für die Dauer einer Kommunikationsperiode ist 20 ms. Durch eine Compiler Direktive (`#define`) im Quellcode des *Communication and Coordination* Package kann diese Zeit jedoch verändert werden.

- **Scheduling:**
  Die Reihenfolge der Abarbeitung der *Application Tasks* wird durch die Vergabe von Prioritäten entsprechend dem Rate Monotonic Scheduling (RMS) [Gal95] gesteuert. Dabei legt der Benutzer die Priorität selbst fest. Wichtige *Application Tasks*, wie z.B. die Applikationen, die einen Roboter ansteuern, sollten dabei hohe Prioritäten erhalten, damit diese Anwendungen in jeder Kommunikationsperiode ablaufen.

  Das Rate Monotonic Scheduling wird bei den parallelen Anwendungsprozessen, den Multi-Threaded Applikationen und dem reinen Multi-Threading angewendet.

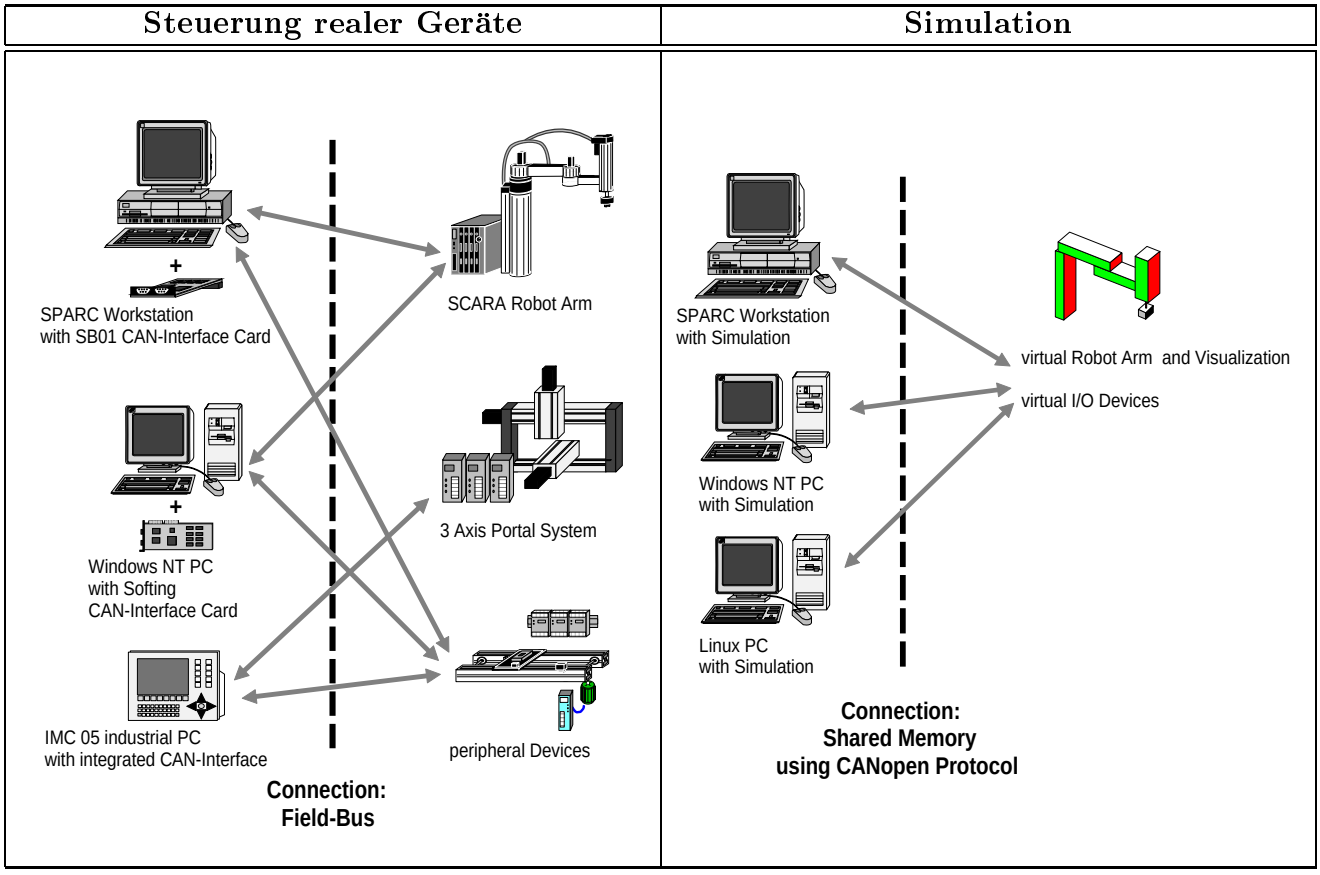


Tabelle 6.1: Übersicht über implementierte Systeme

6.2 Realisierungen des Steuerungssystems

Das Steuerungssystem ist auf verschiedenen Plattformen entwickelt worden:

- UNIX Betriebssysteme (Solaris 2.x und Linux)

- Windows NT (Verwendung der Windows NT API und der POSIX konformen GNU WIN 32 API von Cygnus)
- RMOS (Echtzeitbetriebssystem von Siemens)

In Tabelle 6.1 werden verschiedene Plattformen, auf denen das Steuerungssystem realisiert ist, den gesteuerten Geräten gegenübergestellt. Die folgenden Abschnitte beschreiben die wichtigsten Besonderheiten dieser Implementierungsvarianten.

6.2.1 UNIX Steuerungssystem

Die bei Implementierungen der Steuerung auf den UNIX-Betriebssystemen Solaris (Solaris Releases 2.4 und 2.5) und Linux (Kernel Releases 2.x) verwendeten Betriebssystemaufrufe sind in Tabelle 6.2 aufgeführt. Da für Linux jedoch keine Schnittstellenkarte für den CAN-Bus zur Verfügung steht, können die Mechanismen zur Kommunikation mit einer derartigen CAN-Karte unter Linux nicht eingesetzt werden.

Im Steuerungs-system eingesetzt für:	**Betriebssystemmechanismen**
Prozeßsynchronisation	POSIX.1 Signale und POSIX.4 Signale, System V Semaphore
Änderung der Prozeßpriorität	POSIX.4 Aufruf `priocntl()` (maximale Priorität: `maxrtpri - 1` der Prioritätsklasse Real-Time; aus Sicherheitsgründen wird die höchst mögliche Priorität nicht vergeben [Gal95])
Kommunikation mit der SB01 CAN-Schnittstellenkarte (nicht unter Linux)	das POSIX.1 Signal `SA_SIGINFO` zeigt an, daß ein SYNC-Telegramm verschickt worden ist (POSIX.4 Signale können vom Treiber der Schnittstellenkarte nicht versendet werden.)
	über folgende Betriebssystemfunktionen wird mit dem Kartentreiber kommuniziert: `open()`, `close()`, `pread()`, `pwrite()` oder `ioctl()`

Tabelle 6.2: Betriebssystemmechanismen des UNIX-basierten Steuerungssystems

- Unter Solaris sind sowohl eine prozeßbasierte Steuerungsvariante als auch die Kombination von Prozessen und Threads in einer Steuerung realisiert worden.
  Für das Multithreaded-System wird die proprietäre Thread-Schnittstelle von Solaris verwendet.

  Diese Steuerungssysteme sind auf einem SPARCstation 20 Clone mit zwei SuperSPARC Prozessoren (50 MHz) und 64 MByte Hauptspeicher getestet worden. Dabei ist eine Roboterzelle mit einem Bosch SR60 SCARA Roboterarm, einem Bosch Transfersystem und einer Vielzahl von weiteren digitalen peripheren Geräten kontrolliert worden. Zur Kommunikation mit diesen Geräten über den Feldbus CAN wird die CAN-Schnittstellenkarte SB01 der Firma Stock Microcomputersysteme verwendet.

  Ohne Veränderung des Codes sind die Binärdateien auch auf weiteren Workstations von Sun Microsystems ausgeführt worden (MicroSPARC 5, SPARCstation 10 mit 4 HyperSPARC Prozessoren und UltraSPARC 1). Allerdings ist hierbei nur auf simulierte Geräte zugegriffen worden.

- Die Linux Version ist durch Portierung des Solaris Systems entstanden. Dazu ist das in Abschnitt 4.2 beschriebene objektorientierte Framework verwendet worden.

  Das Linux-System ist mangels geeigneter Schnittstelle zum CAN-Bus nur mit simulierten Geräten eingesetzt worden. Dabei ist es auf einer Vielzahl unterschiedlicher PCs getestet worden.

6.2.2 Windows NT Steuerungssystem

Für Windows NT sind zwei Varianten der Steuerung entstanden. Die erste Variante benutzt ausschließlich die von Windows NT zur Verfügung gestellte Betriebssystemschnittstelle. Die andere Implementierung verwendet GNU WIN 32 als POSIX-Schnittstelle.
Zum Bau des reinen Windows NT Systems ist das Architekturmuster "**universelles industrielles Steuerungssystem**" aus Abschnitt 4.1 verwendet worden, da die Unterschiede der Betriebssystemschnittstelle gegenüber UNIX sehr groß sind. Das Windows NT Steuerungssystem ist ein Thread-basiertes System mit von UNIX abweichenden Interprozeßkommunikationsmechanismen. Im Gegensatz dazu betreffen die Unterschiede zwischen GNU WIN 32 und UNIX nur den Datenaustausch mittels Shared Memory. Das GNU WIN 32 System unterstützt einzelne Prozesse. Daher ist zur Realisierung das objektorientierte Framework aus Abschnitt 4.2 eingesetzt worden.
Zur Kommunikation mit den Geräten (Roboterzelle mit SCARA, Transfersystem und peripheren Modulen) wird eine CAN-Schnittstellenkarte der Firma Softing verwendet.
Beide Windows NT-basierten Varianten sind auf einer Vielzahl unterschiedlicher PCs ausgeführt worden. Allerdings hat sich Windows NT nur sehr bedingt als echtzeitfähig erwiesen. Teilweise sind Zeitabweichungen (Jitter) von 20 ms bei einer Periodendauer von bis zu 25 ms gemessen worden. Die Priorität des Prozesses oder Threads hat dabei keinen Einfluß auf die Zeitabweichung.

6.2.3 RMOS Steuerungssystem

Das Echtzeitbetriebssystem RMOS der Siemens AG ist speziell für die Verwendung auf Siemens Industrierechnern angepaßt. Diese Industrierechner werden in unterschiedlichen Leistungsklassen angeboten.
Das in dieser Arbeit entwickelte Steuerungssystem ist auf einem Industrie-PC (Sicomp IMC 05) mit geringer Leistung (Intel 386 EX Prozessor mit 40 MHz und zwei MByte Hauptspeicher) unter Verwendung des Architekturmusters aus Abschnitt 4.1 übertragen worden. Durch diese Portierung wird zum einen die universelle Verwendbarkeit des Steuerungskonzepts dieser Arbeit demonstriert, zum anderen zeigt sich, daß das Steuerungssystem nur geringe System-Ressourcen benötigt.
Der Industrie-PC IMC 05 besitzt eine integrierte CAN-Schnittstelle sowie proprietäre Schnittstellen zur Ansteuerung von Antrieben und integrierte digitale und analoge Ein- und Ausgänge.
Die Software-Entwicklung für diesen Rechner geschieht durch Cross-Compilation. Darin unterscheidet sich der Industrie-PC von den anderen in dieser Arbeit verwendeten Plattformen.
Für die Implementierung dieser Steuerungsvariante ist das in Abschnitt 4.1 aufgeführte Architekturmuster angewendet worden. Das objektorientierte Framework kann nicht auf das vom POSIX-Standard stark abweichende Betriebssystem RMOS angepaßt werden [1]. Aller-

[1] Grundsätzlich bietet RMOS ähnliche Betriebssystemmechanismen wie im POSIX-Standard definiert. Allerdings weichen die konkreten Aufrufe und deren Parameter von den Vorgaben in POSIX ab. Zusätzlich

dings sind einige der *Additional Components* (*Control Functionality PLC*, *I/O Module*, *Control Functionality Numerical Control* und *Drive Controller*) zur Steuerung der Geräte des komponentenbasierten Steuerungs-Framework aus Abschnitt 4.3.2 ohne Modifikation wiederverwendet worden. Dies ist möglich, da diese Komponenten keine Betriebssystemaufrufe enthalten.
Die Steuerung auf dem IMC 05 wird zur Kontrolle eines Dreiachsportalsystems verwendet. Dieses Portalsystem wird durch die Komponenten *Control Functionality Numerical Control* und *Drive Controller* gesteuert, wobei *Control Functionality Numerical Control* von der Komponente *Control Functionality Robot Control* abgeleitet ist.

6.3 Leistungsdaten

Aussagen über die Leistungsfähigkeit der implementierten Steuerungssysteme sind stark von dem verwendeten Betriebssystem und Rechnertyp abhängig. In Tabelle 6.3 werden die Daten von drei exemplarischen Steuerungssystemen verglichen:

- Hamilton 19 (SPARCstation 20 Clone), 2 SuperSPARC Prozessoren mit 64 MByte Hauptspeicher, Betriebssysteme Solaris 2.4 und 2.5
- PC, AMD K6 200 MHz Prozessor mit 64 MByte Hauptspeicher, Betriebssystem Linux Kernel 2.1
- PC, AMD K5 100 MHz Prozessor mit 40 MByte Hauptspeicher, Betriebssystem Windows NT 4.0 mit GNU WIN 32 API

Die Werte beziehen sich jeweils auf Systeme, die von dem in Abschnitt 4.3 beschriebenen komponentenbasierten Steuerungs-Framework abgeleitet sind. Zur besseren Vergleichbarkeit sind in keinem der aufgeführten Systeme Threads verwendet worden.
Bei allen Meßwerten müssen Messungenauigkeiten der verwendeten Werkzeuge und die Belastung der Systeme durch die Meßwerkzeuge und das Betriebssystem mit in Betracht gezogen werden.
Da der in dieser Arbeit verwendete Industrie-PC Sicomp IMC 05 ein Prototyp mit geringerer Leistungsfähigkeit ist, wird er in diesen Vergleich nicht aufgenommen. Eine Portierung auf ein Seriengerät ist in nächster Zukunft geplant. In diesem Zuge werden entsprechende Messungen durchgeführt.
Im folgenden sollen einige Erläuterungen zu den Messergebnissen aus Tabelle 6.3 gegeben werden:

1. Zur Messung der Auslastung sind jeweils die Meßwerkzeuge der Betriebssysteme verwendet worden: *Performance Meter* von Solaris, *xosview* von Linux und der Windows NT *Task Manager*.

2. Die zyklische Kommunikation mit den Geräten wird durch das CANopen Protokoll vorgegeben. Gleichzeitig stellt die Periodendauer den Takt dar, in dem alle Tasks des Systems einmal abgearbeitet werden.

   Die durch Voreinstellung gewählte Periodendauer ist 20 ms (diese Periodendauer wird auch bei der Messung der Auslastung unter 1. verwendet). Zur Messung der Abweichung von diesem Zyklus (Jitter) wird das zyklische CANopen Synchronisationssignal

muß ein bestimmtes (von POSIX abweichendes) Schema zum Erzeugen der Tasks angewendet werden.

Szenario	Leistungsdaten		
	SPARC Workstation mit Solaris	PC mit Linux	PC mit Windows NT
1. Systemauslastung (eine Robotersteuerungstask und neun weitere *Application Tasks*)			
nur reale Geräte:	15 %	10 %	30 %
mit Simulation:	18 %	11 %	35 %
mit Simulation und Java 1.1 Visualisierung:	25 %	16 %	45 %
2. zyklische Kommunikation			
durchschnittliche Dauer einer Kommunikationsperiode:	20 ms	20 ms	20 ms
Jitter bei durchschnittlicher Kommunikationsperiodendauer:	< 0.3 ms	–	≤ 25 ms
minimale Dauer einer Kommunikationsperiode:	1 ms	1 ms	–
Dispatch Latency der Robotertask:	$0.2 - 0.3$ ms	$0.2 - 0.3$ ms	–
3. Dauer der einmaligen Bearbeitung zyklischer *Application Tasks*			
PTP Roboterbewegung:	$0.8 - 1.4$ ms	$0.7 - 1.2$ ms	–
lineare Roboterbewegung:	$0.9 - 1.7$ ms	$0.9 - 1.3$ ms	–
SPS-Task mit 100 logischen Operationen:	$0.7 - 1.4$ ms	$0.5 - 1.0$ ms	–

Tabelle 6.3: Leistungsdaten der implementierten Systeme

auf dem CAN-Bus verwendet. Da dieser Meßvorgang eine reale Verbindung zum CAN-Bus voraussetzt, können nur die Systeme mit CAN-Kommunikation gemessen werden. Daher liegen keine Messergebnisse für das Linux Steuerungssystem vor.

Die minimale Periodendauer ist die Zykluszeit, in der ein System neben der *Communication and Coordination Task* mindestens eine weitere *Application Task* ausführen kann.

Die Dispatch Latency [2] kann nur durch Vergleich von Zeitmarken, die zu Beginn und Ende der Ausführung von Tasks auf dem Bildschirm ausgegeben werden, ermittelt werden. Die Genauigkeit dieser Messung wird stark beeinträchtigt durch den Zeitaufwand für den Erhalt des Zeitdatums vom Betriebssystem und die Anzeige des Datums auf dem Bildschirm.

3. Die Messungen der Zeitdauer für die einmalige Bearbeitung zyklischer *Application Tasks* verwenden wiederum Zeitmarken des Betriebssystems.

   Die *Communication and Coordination Task* als Prozeß realisiert benötigt auf der SPARC Workstation ca. 1,0 bis 2,0 ms. Dieser Wert hängt von der Dauer der Kommunikation mit der CAN-Schnittstellenkarte ab.

Aufgrund der ungenauen Zeitmarken von Windows NT können bei Messungen auf Betriebssystemebene keine sinnvollen Ergebnisse erzielt werden. Daher fehlen diese in Tabelle 6.3.

[2] Die Dispatch Latency bezeichnet die Zeitdauer für das Einplanen eines Prozesses, d.h. die Zeitspanne ab dem Zeitpunkt des Wechsels in den Zustand "wird ausgeführt" bis zur tatsächlichen Abarbeitung auf dem Prozessor.

Zur Verbesserung der Performance wird in den Steuerungssystemen, die in dieser Arbeit implementiert worden sind, ein zusätzlicher Mechanismus zur Verfügung gestellt. Dabei sorgt das *Communication and Coordination* Subsystem dafür, daß Tasks nur dann ausgeführt werden, wenn sie neue Werte von den kontrollierten Geräten erhalten. Damit wird die Systemlast verringert. Die *Application Tasks* können diesen Mechanismus durch ein bestimmtes Kommando an das *Communication and Coordination* Subsystem nutzen.
Dieser Mechanismus kann besonders bei SPS-Tasks angewendet werden. Fallen in einem Taktzyklus keine neuen Eingangswerte an, so kann auf den Durchlauf des SPS-Programms verzichtet werden (es würden keine geänderten Ausgangswerte berechnet). Durch diese Technik kann die durchschnittliche Ausführungszeit einer SPS-Task deutlich reduziert werden (auf ca. 40 % bis 70 %). Der Mechanismus eignet sich jedoch nicht für Roboter- und NC-Tasks, da diese Tasks eine permanente Lageregelung gewährleisten müssen.

Kapitel 7

Zusammenfassung und Ausblick

In der vorliegenden Arbeit werden Konzepte (sowie deren Implementierungen) zur Entwicklung einer neuen Generation portabler, universeller, industrieller Steuerungssysteme vorgestellt. Die Anforderungen, die dieses neue Steuerungskonzept erfüllt, sind:

- Die Steuerung kann universell eingesetzt werden. Durch eine derartige Steuerung wird eine Vielzahl unterschiedlicher Geräte (wie z.B. Roboterarme, digitale E/A Baugruppen oder kartesische Portalsysteme) kontrolliert. Daher beinhaltet dieses System verschiedene Steuerungsfunktionen. Die wichtigsten unterstützten Steuerungstypen sind Robotersteuerungen, speicherprogrammierbare Steuerungen und numerische Steuerungen.

- Durch die Integration unterschiedlicher Steuerungstypen ist die Grundlage für eine einheitliche Programmierumgebung gegeben. Zum einen können Applikationsprogramme die verschiedenen Steuerungsfunktionen kombinieren (z.B. Roboterprogramme nutzen SPS-Funktionen). Zum anderen können die Applikationsprogramme zur Steuerung unterschiedlicher Geräte ohne Hardware-Aufwand miteinander kommunizieren, da sie auf dem gleichen Steuerungssystem ablaufen.

- Das Steuerungskonzept ist auf mehrere Standardplattformen (wie z.B. Workstations oder PCs bzw. deren industrietaugliche Pendants) portiert worden. Dazu sind entsprechende Ansätze zur Wiederverwendung entwickelt und eingesetzt worden: objektorientiertes System, Architekturmuster, objektorientiertes Framework und Architekturmodell für komponentenbasierte Frameworks bzw. komponentenbasiertes Steuerungs-Framework.

- Das Steuerungssystem unterstützt Multitasking durch Verwendung der im POSIX-Standard beschriebenen Scheduling-Mechanismen. Im Gegensatz zu starren Systemen mit einer festen Echtzeitschleife können viele Tasks dynamisch konkurrent bearbeitet werden. Das System unterstützt Mehrprozessorrechner.

- Durch die Plattformunabhängigkeit kann das System auf beliebig leistungsstarke Rechner übertragen werden. Es nimmt daher automatisch am Leistungsfortschritt der Hardware-Entwicklung teil. Neben der Verwendung auf leistungsfähigen PCs oder Workstations ist die universelle Steuerung auch mit Erfolg auf den kleinsten Industrie-PCs der Sicomp Serie IMC 05 der Firma Siemens portiert worden. Dies war durch eine Optimierung des Systems möglich, bei der nur die für die Steuerungsaufgaben zwingend notwendigen Software-Komponenten verwendet worden sind.

Diese Arbeit beschreibt vier Konzepte zur Entwicklung universeller industrieller Steuerungssysteme. Diese Ansätze sind aufeinander aufbauend erarbeitet worden, können jedoch unabhängig voneinander zur Lösung von Problemen verwendet werden.
Der erste Ansatz realisiert ein universelles Robotersteuerungssystem, bei dem erstmals die Prinzipien der objektorientierten Software-Entwicklung konsequent umgesetzt worden sind. Da diese erste Steuerung als universelle Robotersteuerung (HighRobot) konzipiert worden ist, sind in ihr lediglich Steuerungsfunktionen zur Kontrolle einer kompletten Roboterzelle (Robotersteuerung, SPS und Transfersystemsteuerung) realisiert worden. Allein durch diese Menge an Steuerungsfunktionen ersetzt diese HighRobot Steuerung gleichzeitig mehrere konventionelle Steuerungssysteme, die üblicherweise zur Kontrolle der Geräte einer derartigen Fertigungszelle benötigt werden.
Im ersten Ansatz wird mit Hilfe der objektorientierten Software-Technik ein neues universelles Konzept integrierter Steuerungsfunktionen realisiert. Dieses stellt die Ausgangsbasis für die weiteren Entwicklungen dar, die über reine Verbesserungen im Bereich der industriellen Steuerungssysteme hinaus auch Neuerungen in der Software-Technik darstellen. Hierbei wird gezeigt, wie auf möglichst einfache Art eine flexible Wiederverwendung bestehender Konzepte und Systemfragmente erfolgen kann.
Der zweite Ansatz besteht aus einem Architekturmuster zur Entwicklung eines universellen objektorientierten Steuerungssystems. Dazu sind die Erfahrungen aus der Entwicklung des universellen Steuerungssystems abstrahiert und zu einem wiederverwendbaren Standardmuster überarbeitet worden. Dieses unterstützt die Entwicklung unterschiedlicher industrieller objektorientierter Steuerungssysteme und ist dabei nicht nur auf den Bau von Robotersteuerungen beschränkt. Die Hauptanwendung findet das Muster bei der Portierung des Steuerungskonzepts auf Plattformen, die Standards nur teilweise umsetzen (z.B. Industrie-PCs mit speziellen Betriebssystemen oder eingebettete Systeme). Das Architekturmuster selbst stellt eine erste umfassende Vorlage zur Entwicklung von Steuerungen dar. Bisher sind nur Muster für einzelne Fragmente und Details von Steuerungssystemen veröffentlicht worden.
Im Gegensatz zum Architekturmuster bietet das objektorientierte Steuerungs-Framework nicht nur eine abstrakte Vorgabe für die Architektur, sondern ermöglicht auch die Wiederverwendung von Code. Das Framework ist durch die Identifikation und Implementierung der variablen Elemente (Hot Spots), durch die der Steuerungs-Code auf spezielle Anforderungen angepaßt werden kann, entstanden. Im Vergleich zum Architekturmuster kann das Steuerungs-Framework nicht beliebig verwendet werden, da die Anpaßbarkeit sich auf die im Entwurf des Frameworks festgelegten flexiblen Elemente beschränkt und die Ausarbeitung derartiger Hot Spots sehr aufwendig ist.
Im vierten Ansatz wird ein Architekturmodell zur Entwicklung flexibler komponentenbasierter Frameworks vorgestellt. Diese weitere Abstraktionsstufe ist durch eine internationale Zusammenarbeit entstanden, in der gemeinsame Strukturen unterschiedlichster Frameworks untersucht worden sind. Ziel war die Entwicklung eines Vorgehens zur Entwicklung eines wesentlich flexibleren Systems als die bisherigen objektorientierten Frameworks. Das Ergebnis dieser Kooperation ist ein allgemeines Architekturmodell, das domänenübergreifend verwendet werden kann. Diese Modell bietet eine neuartige Klassifikation der einzelnen Komponenten eines Framework und zeigt wie die Klassen von Komponenten miteinander zusammenarbeiten. Entsprechend dieses Ansatzes ist ein komponentenbasiertes Framework entwickelt worden, von dem unterschiedliche Steuerungssysteme für mehrere Plattformen abgeleitet sind.

Das Simulations- und Monitoring-Werkzeug RoboSiM stellt eine erweiterte Ableitung des komponentenbasierten Steuerungs-Framework dar. Zu dem abgeleiteten Steuerungssystem sind weitere Simulations- und Visualisierungskomponenten hinzugefügt worden. Dadurch

ist ein sehr leistungsfähiges Werkzeug entstanden, durch das zum einen Roboterarme sehr realistisch auf dem Steuerungsrechner simuliert werden können. Zum anderen werden die gesteuerten Geräte durch eine Visualisierungskomponente dargestellt. Im Gegensatz zu den bisherigen Simulationssystemen können reale und virtuelle Automatisierungsgeräte gleichzeitig angezeigt und verglichen werden. Dadurch ergeben sich neue Anwendungsmöglichkeiten in Forschung und Lehre sowie im industriellen Einsatz.
RoboSiM demonstriert die Flexibilität und Erweiterbarkeit des komponentenbasierten Ansatzes unter Verwendung von Standardplattformen.

Ausblick

Diese Arbeit stellt sowohl Konzepte wie auch Implementierungen für ein universelles industrielles Steuerungssystem der nächsten Generation zur Verfügung. Neue Steuerungssysteme auf weiteren, bisher nicht verwendeten Plattformen und für bisher nicht erforschte Anwendungen (z.B. Portierung auf einen eingebetteten Einplatinenrechner) können auf dieser Basis (vor allem durch Einsatz der Mechanismen zur Wiederverwendung) entwickelt werden. Eine Weiterentwicklung der Steuerung in Zusammenarbeit mit Siemens hat bereits begonnen.
Insbesondere RoboSiM kann in seinen Steuerungs- und Simulationsfunktionen zu einem leistungsfähigen Werkzeug ausgebaut werden, das neue Möglichkeiten eröffnet. Durch erweiterte Darstellungsmöglichkeiten kann auf dieser Basis beispielsweise die Fabrikplanung oder Modellierung von Fertigungsprozessen unterstützt werden. Dazu wird die dreidimensionale Darstellung verbessert und erweitert. Zusätzliche Möglichkeiten zur interaktiven Kombination von Geräten auf graphische Art und Weise am Monitor oder durch Skriptdateien sind hierzu zu ergänzen.
Das eingeführte Architekturmodell zur Entwicklung komponentenbasierter Steuerungs-Frameworks eröffnet eine neue Richtung zur Unterstützung der Entwicklung von Frameworks im Allgemeinen. Neben dem vorgestellten Architekturmodell sind parallel zu bekannten Entwurfsmustern und Architekturmustern weitere Muster spezifisch zur Entwicklung von Frameworks denkbar. Zwar können für die Entwicklung von Frameworks bekannte Architektur- und Entwurfsmuster hilfreich eingesetzt werden, sie genügen für diesen speziellen Einsatzzweck jedoch nicht. Die spezifischen Eigenarten bzw. Charakteristika von Frameworks führen zur Notwendigkeit, spezifische Muster einzusetzen.
Darüberhinaus ist die Suche nach weiteren Mustern zur Beschreibung von Details innerhalb des vorgestellten Architekturmodells sinnvoll.
Neben einer Untersuchung von Mustern auf dem Gebiet der Frameworks, ist auch die Betrachtung der verschiedenen technischen Realisierungsmöglichkeiten für das Architekturmodell eine erfolgsversprechende weiterführende Forschungsarbeit. Insbesondere die Frage nach den Möglichkeiten der Kombination der Komponenten innerhalb des Architekturmodells sind bisher kaum erforscht. Traditionelle Vorgehensweisen zur Kombination (wie z.B. Aggregationen [Pre97]) sollten ergänzt und erweitert werden.
Auf der Basis des Architekturmodells zur Entwicklung komponentenbasierter Frameworks können darüberhinaus auch generative Techniken (wie in [SB98, CE99, PKS99, SRPC99, PSR00, SPM00] beschrieben) zur automatischen Ableitung konkreter Frameworks eingesetzt werden [SP00].

Anhang A

Abbildungen der Geräte

Abbildung A.1: Roboterzelle mit Steuerungsrechner

Die Roboterzelle in Abbildung A.1 besteht aus einem SCARA Roboterarm (Bosch SR 60) und verschiedenen weiteren peripheren Geräten (Transfersystem und digitale Komponenten). Im Vordergrund befinden sich eine SPARC Workstation und ein Windows NT PC. Beide Rechner können zur Steuerung der Geräte in der Roboterzelle eingesetzt werden.

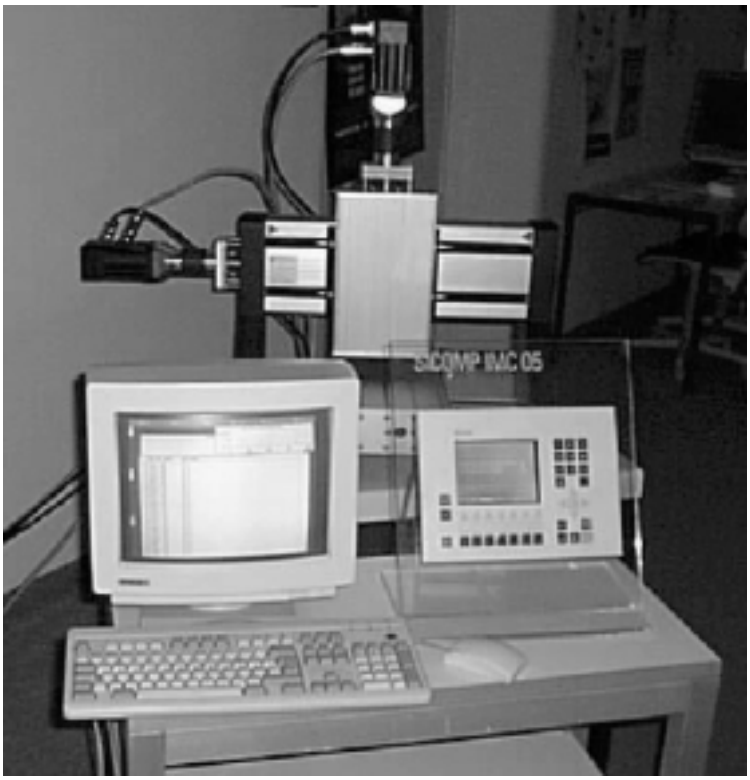


Abbildung A.2: Sicomp Industrie-PC IMC 05 und Portalsystem

Im Hintergrund der Abbildung A.2 ist eines der angesteuerten Dreiachsportalsysteme zu sehen. Vorne rechts befindet sich ein Sicomp Industrie-PC IMC 05 als Steuerungsrechner. Rechts daneben steht ein Analysewerkzeug zur Überwachung der Kommunikation zwischen dem IPC und dem Portalsystem.

Anhang B

RoboSiM 2.0 Java 3D

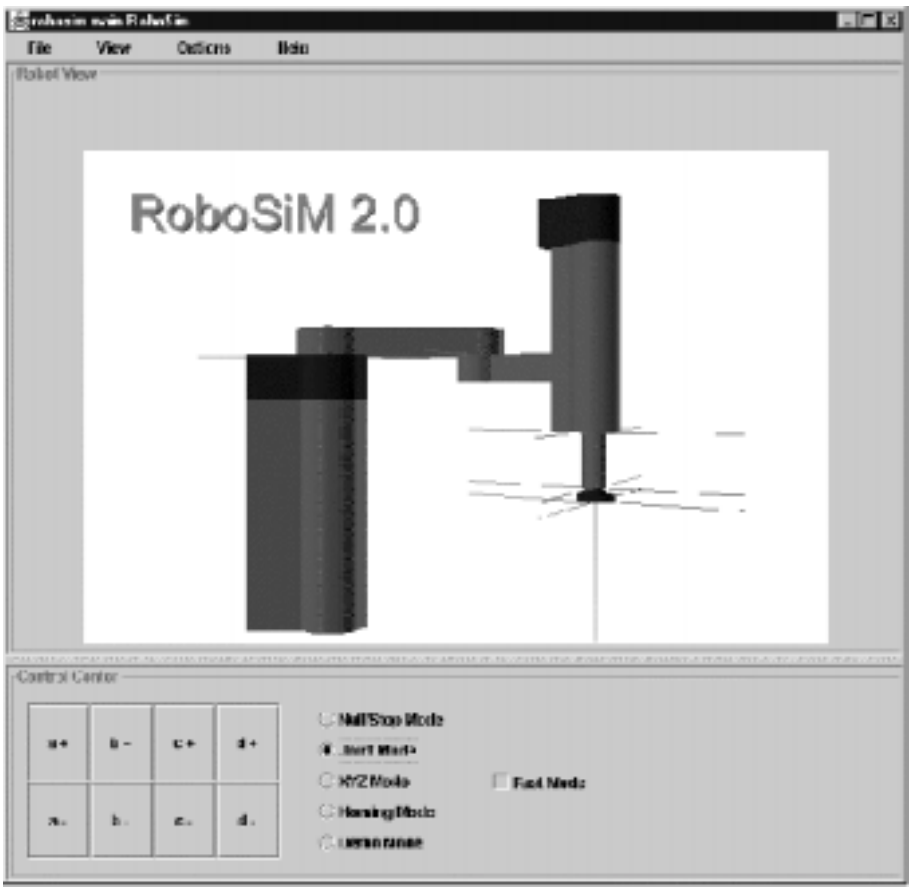


Abbildung B.1: RoboSiM 2.0 – implementiert mit Java 3D

Literaturverzeichnis

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, und S. Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[AKZ96] M. Awad, J. Kuusela, und J. Ziegler. *Object-Oriented Technology for Real-Time Systems.* Prentice Hall, Englewood Cliffs, 1996.

[Ale77] C. Alexander. *The Timeless Way of Building.* Oxford University Press, New York, 1977.

[Ale99] C. Alexander. The Origin of Pattern Theory. *IEEE Software*, 48(5):71 – 82, September/Oktober 1999.

[Alf77] M.W. Alford. A Requirements Engineering Methodology for Real-Time Processing Requirements. *IEEE Transactions on Software Engineering*, 3(1):60 – 69, Januar 1977.

[AS85] B. Adelson und E. Soloway. The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering*, 11(11):1351–1360, 1985.

[AS94] P. Arjanne und J.J. Suutarinen. Fully automatic offline programmable robotized press brake controlled with two transputer based multi axis controllers. In *Proceedings of 25th, Int'l Symposium on Industrial Robots, Robotics '94 - Flexible Production Flexible Automation*, Seiten 191 – 197, Hannover, April 1994.

[Bac86] M. J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, Englewood Cliffs, 1986.

[Bar94] R. Bartelt. A New Concept for Robot Controls. In *Proceedings of 25th, Int'l Symposium on Industrial Robots, Robotics '94 - Flexible Production Flexible Automation*, Seiten 381 – 387, Hannover, April 1994.

[Bau93] F.L. Bauer. Software Engineering – wie es begann. *Informatik-Spektrum*, 16:259 – 260, 1993.

[BC85] A.J. Buckley und G.F. Collins. A Structured Programming Robot Language. In S.Y. Nof, Editor, *Handbook of Industrial Robotics*, Seiten 381 – 403. Wiley, New York, 1985.

[BDMP87] D. Belsnes, H.P. Dahle, und B. Møller-Pederson. Rationale and tutorial on OSDL: An object-oriented extention of SDL. Mjølner Report Series N-EB-6, 1987.

[Bei90] L. Beiner. Time-optimization of point-to-point robotic motions. *Robotersysteme*, (6):171 – 176, Juni 1990.

[BHK+89] J. Bares, M. Hebert, T. Kanade, E. Kortkov, T. Mitchell, R. Simmons, und W. Whittaker. Ambler: An Autonomous Rover for Planetary Exploration. *IEEE Computer*, 1989.

[Bic94a] J. Bickendorf. Full-Automatic Off-Line programming of Complex Cutting Paths - A Contribution to the Economic Production of "Lotsize 1". In *Proceedings of 25th, Int'l Symposium on Industrial Robots, Robotics '94 - Flexible Production Flexible Automation*, Hannover, 1994.

[Bic94b] J. Bickendorf. Teaching Robot Programming Using CAL. In *Proceedings of 25th, Int'l Symposium on Industrial Robots, Robotics '94 - Flexible Production Flexible Automation*, Seiten 499 – 506, Hannover, 1994.

[Bir93] E.T. Birrer. Frameworks in the financial engineering domain: An experience report. In *Proceedings of ECOOP'93, European Conference on Object-Oriented Programming*, Kaiserslautern, 1993, Springer LNCS 707.

[BKKZ92] R. Budde, K. Kautz, K. Kuhlenkamp, und H. Züllighoven. *Prototyping*. Springer, New York, 1992.

[Bli88] G.J. Blickley. Variety of Languages Offered in PLCs. *Control Engineering*, (1):44 – 46, Januar 1988.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, und M. Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. Wiley, Chichester, 1996.

[Boe76] B.W. Boehm. Software Engineering. *IEEE Transactions On Computers*, C-25(12):1226–1241, 1976.

[Boo91] G. Booch. *Object-Oriented Analysis and Design*. Benjamin/Cummings, Redwood City, CA, 1991.

[Boo94] G. Booch. *Object-Oriented Analysis and Design*. Benjamin/Cummings, Redwood City, zweite Ausgabe, 1994.

[BR95] G. Booch und J. Rumbaugh. *Unified Method for Object-Oriented Development*. Rational Software Corporation, Santa Clara, 1995.

[Bra96] B. A. Brandlin. The Real-Time Supervisory Control of an Experimental Manufacturing Cell. *IEEE Transactions On Robotics and Automation*, 12(1):1 – 14, Februar 1996.

[BRI99] G. Booch, J. Rumbaugh, und Jacobson I. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, 1999.

[BS69] K. Bartlett und R. Scantelbury. A note on reliable full-duplex transmission over half-duplex lines. *Communications of the ACM*, 12(5):260 – 265, Mai 1969.

[BSW96] R. Bernhard, G. Schreck, und C. Willnow. A Cost Efficient Robot Control System – CEROS. In *Proceedings of 27th, Int'l Symposium on Industrial Robots, Robotics '96 - Flexible Production Flexible Automation*, Mailand, 1996.

[BTT87] B. Bhanu, M. Thune, und N. Thune. CAOS: A Hierarchical Robot Control Systems. In *Proc. of Int'l IEEE Conference on Robotics and Automation*, Seiten 1603 – 1608. IEEE Computer Society Press, 1987.

[CAB+93] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, und P. Jeremaes. *Object-Oriented Development – The Fusion Method.* Prentice Hall, Englewood Cliffs, 1993.

[CAL94] CAN in Automation e.V. (CiA). *CAL Application Layer for Industrial Applications, CiA Draft Standard DS-201, CAN in the OSI Reference Model*, 1994.

[CAN95a] CAN in Automation e.V. (CiA). *CAL based Communication Profile for Industrial Systems, CiA Draft Standard Proposal DS-301*, 1995.

[CAN95b] CAN in Automation e.V. (CiA). *CAL based Device Profile for I/O Modules, CiA Draft Standard Proposal DS-401*, 1995.

[CAN97] CAN in Automation e.V. (CiA). *CAL based Device Profile for Drives and Motion Control, CiA Draft Standard Proposal DS-402*, 1997.

[Cas83] R. Cassinis. A ROBOTIC SYSTEM FOR COMPLEX COATING PROCESS. In *Proceedings of 13th, Int'l Symposium on Industrial Robots and Robotics '83*, Seiten 12 – 19, Chicago, April 1983. Springer.

[CCITT88] CCITT, International Telecommunication Union, Geneva. *CCITT Blue Book, Functional specification and description language*, 1988.

[CDSM97] S. Calvalieri, A. Di Stefano, und O. Mirabella. Impact of Fieldbus on Communication in Robotic Systems. *IEEE Transactions On Robotics and Automation*, 13(1):30 – 48, Februar 1997.

[CE99] K. Czarnecki und U.W. Eisenecker. Synthesizing Objects. In *Proceedings of ECOOP'99, European Conference on Object-Oriented Programming*, Seiten 18 – 42, Lissabon, Juni 1999, Springer LNCS 1628.

[CFAJ86] J.B. Chen, R.S. Fearing, B.S. Amstrong, und Burdick J.W. NYMPH: A Multiprocessor for Manipulation Applications. In *Proc. of Int'l IEEE Conference on Robotics and Automation*, Seiten 1731 – 1736. IEEE Computer Society Press, April 1986.

[CG95] M.J. Chonoles und C.C. Gilliam. Real-time object-oriented system design using the object modelling technique OMT. *Journal of Object-Oriented Programming*, 8(3):16 – 24, Juni 1995.

[Chr94] Chrysler, Ford and General Motors. *Requirements of Open Modular Architecture Controllers for Applications in the Automotive Industry.* December 1994.

[CNM95] P. Coad, D. North, und M. Mayfield. *Object Models, Strategies, Patterns & Applications.* Yourdon Press, Prentice Hall, Upper Saddle River, 1995.

[Coa92] P. Coad. Object-Oriented Patterns. *Communications of the ACM*, 35(9):152 – 159, September 1992.

[Cop92] J.O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, Reading, 1992.

[Cop96] J.O. Coplien, J. Vlissides, und N. Kerth, Editoren. *Pattern Languages of Program Design, Vol 2.* Addison-Wesley, Reading, 1996.

[Cop99] J.O. Coplien. Reevaluating the Architectural Metaphor: Towards Piecemeal Growth. *IEEE Software*, 48(5):40 – 44, September/Oktober 1999.

[COR96] Object Management Group (OMG), `http://www.omg.org`. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, Juli 1996.

[Cra89] J.J. Craig. *Introduction to Robotics.* Addison-Wesley, Reading, zweite Ausgabe, 1989.

[CY91a] P. Coad und E. Yourdon. *Object-Oriented Analysis.* Yourdon Press, Prentice Hall, Englewood Cliffs, zweite Ausgabe, 1991.

[CY91b] P. Coad und E. Yourdon. *Object-Oriented Design.* Yourdon Press, Prentice Hall, Englewood Cliffs, 1991.

[Dam94] M. Damm. A Real-Time Control System for the Coordination of Two Industrial Robots. *IEEE Transactions on Control Systems*, 2(5):1182 – 1187, Mai 1994.

[DEF+96] W. Daxwanger, E. Ettelt, F. Fischer, U. Hanebeck, und G. Schmidt. ROMAN: Ein mobiler Serviceroboter als persönlicher Assistent in belebten Innenräumen. In G. Schmidt und F. Freyberger., Editoren, *12. Fachgespräch Autonome und Mobile Systeme (AMS '96)*, Seiten 314 – 333, Berlin, 1996. Springer.

[DeM78] T. DeMarco. *Structured Analysis and Systems Specification.* Yourdon Press, Prentice Hall, Englewood Cliffs, 1978.

[DFP95] P. Darscht, A. H. Frigeri, und C. E. Pereira. Building up Object-Oriented Industrial Systems: Experiences Interfacing Active Objects with Technical Plants. In *Factory Communication Systems*, Seiten 53 – 61, Leysin, Schweiz, 1995. IEEE Computer Society Press.

[DH91] R. Dillmann und M. Huck. *Informationsverarbeitung in der Robotik.* Springer, Berlin, 1991.

[DH97] D. Doscher und R. Hodges. SEMATECS'S Experiences with the CIM Framework. *Communications of the ACM*, 40(10):82 – 84, Oktober 1997.

[DIN83] *DIN 66257: Numerisch gesteuerte Arbeitsmaschinen, Begriffe.* Berlin, 1983.

[DIN85] Deutsche Elektrotechnische Kommission im DIN und VDE (DKE). *Elektrische Ausrüstung von Industriemaschinen, Teil 1: Allgemeine Festlegungen, Deutsche Fassung EN 60 204, DIN VDE 0113 Teil 1/02. 86*, 1985.

[DIN95] *DIN 66312: Industrial Robot Language IRL, Language Dscription (Draft), Version 1.6 Teil 1 und Teil 2.* Stuttgart, 1995.

[DMM98] C. M. Davidson, J. McWhinnie, und M. Mannion. Introducing Object Oriented Methods To PLC Software Design. In *Engineering of Computer based Systems*, ECBS, Seiten 150 – 157, Jerusalem, Israel, 1998. IEEE Computer Society Press.

[Dod96] M. Dodani. Object-Oriented methodologies in practice: The "Big Picture". *Journal of Object-Oriented Programming*, 9(2):26 – 29, April 1996.

[Dor85] R.C. Dorf. *Robots and Automated Manufacturers.* Reston Publishing, Reston, VA, 1985.

[Dou98a] B.P. Douglass. *Doing Hard Time: Using Object-Oriented Programming and Software Patterns in Real Time Applications.* Addison-Wesley, Reading, 1998.

[Dou98b] B.P. Douglass. *Real-Time UML.* Addison-Wesley, Reading, 1998.

[Edw93] K. Edwards. *Real-Time Structured Method.* Wiley, Chichester, 1993.

[Ell94] J.R Ellis. *Objectifying Real-Time Systems.* SIGS Books, New York, 1994.

[FB93] E. Freund und H.-J. Buxbaum. Control of Robot-based Flexible Manufacturing Work Cells. In *Int'l Conference on Advanced Mechatronics*, Seiten 348 – 353, Tokyo, 1993.

[FGL88] K.S. Fu, R.C. Gonzalez, und C.S.G. Lee. *Robotics.* McGraw-Hill, New York, 1988.

[Fow97] M. Fowler. *Analysis Patterns, Reusable Object Models.* Addison-Wesley, Reading, 1997.

[FS97] M.E. Fayad und D.C. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10):32–38, Oktober 1997.

[FV98] A. Fink und S. Voß. Software-Wiederverwendung mittels Frameworks. *WiSt*, (10):535 – 538, Oktober 1998.

[Gal95] B.O. Gallmeister. *POSIX.4: Programming for the Real World.* O'Reilly, Sebastopol, 1995.

[Gam92] E. Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge.* Springer, Berlin, 1992.

[Ger90] D. Gershon. Parallel Process Decomposition of a Dynamic Manipulation Task: Robot Sewing. *IEEE Transactions On Robotics and Automations*, 6(3):357 – 367, Juni 1990.

[GGM93] M. A. Goodman, M. Goyal, und R.A. Massoudi. *Solaris Porting Guide.* SunSoft Press, Prentice Hall, Mountain View, 1993.

[GH70] S.C. Gupta und L. Hasdorff. *Fundamentals of Automatic Control.* Wiley, Chichester, 1970.

[GHJV94] E. Gamma, R. Helm, R. Johnson, und J. Vlissides. *Design Patterns: Abstractions and Reuse of Object-Oriented Software.* Addison-Wesley, Reading, 1994.

[GM96] K. Goldberg und M. Mascha. *Mercury Project, Robotic Tele-Excavation ,* `http://www.usc.edu/dept/raiders/`. University of Southern California, Los Angeles, USA, 1996.

[Gra93] I. Graham. *Object Oriented Methods.* Addison-Wesley, Wokingham, zweite Ausgabe, 1993.

[GW94] R.A. Grupen und R.S. Weiss. Integrated control for interpreting and manipulating the robot environment. *Robotica*, 12:165 – 174, 1994.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[Hei94] P. Heinrich. *IRL Programmieranleitung, Version 1.1 Teil 1.* Stuttgart, 1994.

[Hel96] R. Helm. Patterns, Architecture and Software. *ACM Sigplan Notices*, 31(1):2 – 3, Januar 1996.

[HH97] D. Henrich und Th. Hoeniger. Parallel processing approaches in robotics. In *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE'97)*, Seiten 702 – 707, Guimaraes, Portugal, 1997.

[Hir93] G. Hirzinger. Das Raumfahrt-Robotik-Experiment ROTEX – Konzepte, Erfahrungen und Perspektiven. *Intelligente Steuerung und Regelung von Robotern, VDI Berichte*, (1094):21 – 38, 1993.

[HN92] M. Henricson und E. Nyquist. *Programming in C++, Rules and Recommendations.* Ellemtel Telecommunication Systems Laboratories, `http://www.mgl.co.uk/people/kirit/cpprules.html`, Älvsö, Sweden, 1992.

[Hoc89] D. Hochgrefe. *Estelle, Lotos und SDL, Standard-Spezifikationssprachen für verteilte Systeme.* Springer, Berlin, 1989.

[Hol85] L.L. Hollingshead. Elements of Industrial Robot Software. In S.Y. Nof, Editor, *Handbook of Industrial Robotics*, Seiten 337 – 351. Wiley, New York, 1985.

[HOO95] HOOD User's Group, HOOD Technical Group, Brussels, Belgium. *HOOD Reference Manual, Release 4*, 1995.

[HSG96] B. Henderson-Sellers und I. Graham. OPEN: towards method convergence. *IEEE Computer*, 28(4), April 1996.

[Hun91] V.D. Hunt. *Understanding Robotics.* Academic Press, San Diego, 1991.

[Hüs95] T. Hüsener. *Objektorientierter Entwurf von nebenläufigen, verteilten und echtzeitfähigen Softwaresystemen.* Spektrum, Akademischer Verlag, Heidelberg, 1995.

[IEC92] *IEC 1131: Standard for programmable controllers Part 3: Programming languages*, 1992.

[IEE93] Institute of Electrical and Electronics Engineers (IEEE), 1-55937-375-X. *Portable Operating System Interface (POSIX)– Part 1: Application Programming Interface (API) [C Language]– Amendment: Realtime Extentions*, 1993.

[Jar84] J.F. Jarvis. Robotics. *IEEE Computer*, 17(10):283 – 292, Oktober 1984.

[JC95] I. Jacobson und M. Christerson. Modelling With Use Case, A confused world of OOA and OOD. *Journal of Object-Oriented Programming*, 8(5):15 – 20, September 1995.

[JCJ92] I. Jacobson, M. Christerson, und P. Jonsson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Wokingham, England, 1992.

[Joh92] R.E. Johnson. Documenting Frameworks using Patterns. In *Proceedings of OOPSLA'92, Seventh Annual Conference On Object-Oriented Programming Systems, Languages and Applications*, volume 27, Seiten 63 – 76. ACM SIGPLAN Notices, Oktober 1992.

[Kay93] A.C. Kay. The Early History of Smalltalk. *ACM Sigplan Notices*, 28(3):69 – 96, März 1993.

[KGL+97] W. Küchlin, G. Gruhler, T. Lumpp, A. Speck, und A. Rupp. HIGHROBOT: Telerobotics in the Internet. In *Proceedings of ETFA'97, Int'l IEEE Conference on Emerging Technologies and Factory Automation*, Seiten 115 – 120, Los Angeles, 1997. IEEE Computer Society Press.

[KGSL97a] W. Küchlin, G. Gruhler, A. Speck, und T. Lumpp. HIGHROBOT: A High-performance Universal Robot Control on Parallel Workstations. In *Proceedings of ECBS'97, Int'l IEEE Symposium and Workshop on Engineering of Computer Based Systems*, Seiten 444 – 451, Monterey, 1997. IEEE Computer Society Press.

[KGSL97b] W. Küchlin, G. Gruhler, A. Speck, und T. Lumpp. HIGHROBOT: Distributed Object-Oriented Real-Time Systems. In *Proceedings of ARCS'97, 14th international ITG/GI-Conference Architecture of Computer Systems*, Seiten 91 – 99, Rostock, 1997. VDI.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Volume 1, 2 and 3.* Addison-Wesley, Reading, 1973.

[Knu92] D.E. Knuth. *Literate Programming.* University of Chicago Press, Chicago, 1992.

[Kor85] Y. Koren. Numerical Control and Robotics. In S.Y. Nof, Editor, *Handbook of Industrial Robotics*, Seiten 203 – 231. Wiley, New York, 1985.

[Kot98] J. Kotula. Using Patterns To Create Component Documentation. *IEEE Software*, 47(2):84 – 92, März/April 1998.

[KP88] G.E. Krasner und S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26 – 49, August/September 1988.

[KPK95] P. Kopacek, R. Probst, und G. Kronreif. A modular control system for flexible robotic manufacturing cells. In *Proceedings of 26th, Int'l Symposium on Industrial Robots, Robotics '95 - Flexible Production Flexible Automation*, Seiten 533 – 539, Singapur, 1995.

[KR94] D. Kugelmann und G. Reinhart. Automatische Online-Generierung von Handhabungsprogrammen mit der 3D-Simulation. In P. Levi und T. Bräunl., Editoren, *10. Fachgespräch Autonome und Mobile Systeme (AMS '94)*, Seiten 349 – 360, Berlin, 1994. Springer.

[KSS96] S. Kleinman, D. Shah, und B. Smaalders. *Programming with Threads.* SunSoft Press, Prentice Hall, Mountain View, 1996.

[LA90] S.-T. Levi und A.K. Agrawala. *Real-Time System Design.* McGraw-Hill, New York, 1990.

[LC92] M. Linn und M. Clancy. The Case for Case Studies of Programming Problems. *Communications of the ACM*, 35(3):121 – 132, März 1992.

[LH89] K.J. Lieberherr und L.M. Holland. Assuming good style for object-oriented programs. *IEEE Software*, 38(5):38 – 48, September/Oktober 1989.

[Lip91] S.B. Lippman. *C++ Primer.* Addison-Wesley, Reading zweite Ausgabe, 1991.

[LMKQ89] S.J. Leffler, K. McKusick, M. Karels, und J. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley, Reading, 1989.

[LN95] D.B. Lange und Y. Nakamura. Interactive Visualization of Design Patterns Can Help in Framework Understanding. In *Proceedings of OOPSLA'95, Tenth Annual Conference On Object-Oriented Programming Systems, Languages and Applications*, Volume 30, Seiten 342 – 356, Austin, Oktober 1995. ACM SIGPLAN Notices.

[Loy90] P.H. Loy. A Comparison of Object-Oriented and Structured Development Methods. *ACM SIGSOFT Software Engineering Notes*, 15(1), Januar 1990.

[Luh85] J.Y.S. Luh. Design of Control Systems for Industrial Robots. In S.Y. Nof, Editor, *Handbook of Industrial Robotics*, Seiten 169 – 202. Wiley, New York, 1985.

[LW95] C. Laloni und E. Wahl. Principles of Robot Simulation and their Application in a PC-based Robot Simulation System. In W. Straßer und E. Wahl, Editoren, *Graphics and Robotics*, Seiten 1 – 30. Springer, Berlin, 1995.

[Lyo98] A. Lyons. *UML for Real-Time Overview.* ObjecTime Limited,
http://www.objectime.com/otl/technical/umlrt.html,
Rational Software Cooperation,
http://www.rational.com/uml/resources/whitepapers/index.jtmpl, 1998.

[Mar67] J. Martin. *Design of Real-Time Computer Systems.* Prentice-Hall, Englewood Cliffs, 1967.

[McI76] M. D. McIlroy. Mass-produced software components. In J.M. Buxton, P. Maur, und B. Randell, Editoren, *Software Engineering Concepts and Techniques, Proceedings of 1968 North Atlantic Treaty Organisation (NATO) Conference on Software Engineering Garmisch-Partenkirchen*, Seiten 88 – 98. New York, 1976.

[McK95] P.J. McKerrow. *Introduction to Robotics.* Addison-Wesley, Sydney, 1995.

[MCK97] M. Meusel, K. Czarnecki, und W. Köpf. A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext. In *Proceedings of ECOOP'97, European Conference on Object-Oriented Programming*, Jyväskylä, Finland, 1997, Springer LNCS 1241.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, Englewood Cliffs, 1988.

[MGGT96] D. Morris, E. Gareth, P. Green, und C. Theaker. *Object Oriented Computer Systems Engineering.* Springer, New York, 1996.

[Mic90] G. Michel. *Programmable Logic Controllers and Architecture and Application.* Wiley, New York, 1990.

[MM95] R. Maglica und N. Martenson. Teaching Robot Programming Using CAL. In *Proceedings of 26th, Int'l Symposium on Industrial Robots, Robotics '95 - Flexible Production Flexible Automation*, Seiten 111 – 116, Singapur, 1995.

[MN96] S. Moser und O. Nierstrasz. The effect of object-oriented frameworks on developer productivity. *IEEE Computer*, 29(9):45 – 51, September 1996.

[Moi99] D. Moitra. Software Engineering in the Small. *IEEE Computer*, 32(10):39 – 40, Oktober 1999.

[MTP95] *Multithreaded Programming Guide.* SunSoft Press, Prentice Hall, Mountain View, 1995.

[ND95] O. Nierstrasz und L. Dami. Component-oriented software technology. In O. Nierstrasz und D. Tsichritzis, Editoren, *Object-Oriented Software Composition.* Prentice Hall, Englewood Cliffs, 1995.

[NGT92] O. Nierstrasz, S. Gibbs, und D. Tsichritzis. Component-oriented software development. *Communication of the ACM*, 35(9):160 – 165, 1992.

[Nic94] E.J. Nicolson. Standardizing I/O for Mechatronic Systems SIOMS using Real Time UNIX Device Drivers. In *Proceedings of '94, International Conference on Robotics and Automation*, Seiten 3489 – 3494, San Diego, Mai 1994. IEEE Press.

[NSN95] T. Nakazato, K. Sakanashi, und I. Nagamatsu. Cooperative and Independent Control of Multiple Robots with Common Controlled Elements (Station-Twin Cooperative Robot System). In *Proceedings of 26th, Int'l Symposium on Industrial Robots, Robotics '95 - Flexible Production Flexible Automation*, Seiten 521 – 525, Singapur, 1995.

[NZ95] T.M. Nabhan und A.Y. Zomaya. Application of Parallel Processing to Robotic Computational Tasks. *International Journal of Robotics Research*, 14(1):76 – 86, Februar 1995.

[OHE96] R. Orfali, D. Harkey, und J. Edwards. *The Essential Distributed Object Survival Guide.* Wiley, New York, 1996.

[OMA98] OMAC API Work Group. *OMAC API SET, Version 0.18, Working Document*, Februar 1998.

[Ope98] Open Control Foundation. *Open Control Interface, Version 1.4*, Februar 1998.

[OQC97] G. Odenthal und K. Quibeldy-Cirkel. Using Patterns for Design and Documentation. In *Proceedings of ECOOP'97, European Conference on Object-Oriented Programming*, Jyväskylä, Finland, 1997, Springer LNCS 1241.

[OSA96] *OSACA Open Architecture for Controls within Automation Systems OSACA I & II Final Report, Project No EP 6379 & EP 9115.* EU Brussels, April, 1996.

[OYL97] C. Ou-Yang und T.S. Lin. Developing an Integrated Framework for Feature-Based Early Manufacturing Cost Estimation. *Int'l Journal of Advanced Manufacturing Technology*, 13(9):618 – 629, September 1997.

[Par72] D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053 – 1058, December 1972.

[Pet87] L. Peters. *Advanced Structured Analysis and Design.* Prentice Hall, Englewood Cliffs, 1987.

[Pet89] F.D. Petruzella. *Programmable Logic Controllers.* McGraw-Hill, New York, 1989.

[PKS99] E. Pulvermüller, H. Klaeren, und A. Speck. Aspects in Distributed Environments. In *Proceedings of the GCSE'99, First International Symposium on Generative and Component-Based Software Engineering*, Erfurt, September, 1999.

[PPSS95] W. Pree, G. Promberger, A. Schappert, und P. Sommerlad. Active Guidance of Framework Development. *Software–Concepts and Tools*, (16):136 – 145, 1995.

[Pre94] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, Reading, 1994.

[Pre97] W. Pree. *Komponentenbasierte Softwareentwicklung mit Frameworks.* dpunkt, Heidelberg, 1997.

[PRST99] D. Parsons, A. Rashid, A. Speck, und A. Telea. A "Framework" for Object Oriented Frameworks Design. In *Proceedings of TOOLS EUROPE'99, Objects, Components, Agents; Technologie of Object-Oriented Languages and Systems, 29th Int'l Conference and Exhibition*, Seiten 141 – 151, Nancy, Frankreich, Juni 1999. IEEE Computer Society Press.

[PSR00] E. Pulvermüller, A. Speck und R. Rashid.. Implementing collaboration-based Design using Aspect-Oriented Programming. In *Proceedings of Proceedings of TOOLS USA 2000, Technologie of Object-Oriented Languages and Systems, 34th Int'l Conference and Exhibition*, St. Barbara, Juli/August 2000.

[PTH97] G. Pritschow, T.L. Tran, und J. Hohenadel. Standalone PC-Controller on an open Platform. In *Proceedings of the 30th International Symposium on Automotive Technology & Automation*, Florenz, Juni 16 – 19 1997.

[PUD94] G. Pritschow, A. Uhl, und P. Demel. Flexibility and Cost Efficiency with an Open Multitasking Control Architecture for Robots. In *Proceedings of 25th, Int'l Symposium on Industrial Robots, Robotics '94 - Flexible Production Flexible Automation*, Seiten 395 – 402, Hannover, April 1994.

[PW97] C. Pelich und F.M. Wahl. ZERO++ An OOP Environment for Multiprocessor Robot Control. *International Journal of Robotics and Automation*, 12(2):49 – 57, Februar 1997.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, und W. Lornsen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[RIB99] J. Rumbaugh, Jacobson I., und G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, 1999.

[RJ98] D. Roberts und R. Johnson. Patterns for evolving Frameworks. In R. Martin, D. Riehle, und F. Buschmann, Editoren, *Pattern Languages of Program Design, Vol. 3*, Seiten 471 – 486. Addison-Wesley, Reading, dritte Ausgabe, 1998.

[RK92] W. Reis und E. Kroth. A ROBOTIC SYSTEM FOR COMPLEX COATING PROCESS. In *Proceedings of 23rd, Int'l Symposium on Industrial Robots, Robotics '92 - Flexible Production Flexible Automation*, Seiten 617 – 620, Barzelona, 1992.

[RNS93] U. Rembold, B.O. Nnaji, und A. Storr. *CIM: Computer Integrated Manufacturing and Engineering.* Addison-Wesly, Reading, 1993.

[Roy70] W.W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of WESCON*, Abschnitt A1, Seiten 1 – 9, August 1970.

[RS77] D.T. Ross und K.E. Schoman. Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering*, 3(1):6 – 15, Januar 1977.

[RS98] J. Rumbaugh und B. Selic. *Using UML for Modeling Complex Real-Time Systems.* ObjecTime Limited, http://www.objectime.com/otl/technical/umlrt.html, Rational Software Cooperation, http://www.rational.com/uml/resources/whitepapers/index.jtmpl, 1998.

[Rze93] H. Rzehak. Real-Time UNIX: What Performance can we expect? *Control Eng. Practice*, 1(1):65 – 70, 1993.

[SAL97] K. Schröer, S.L. Albright, und A. Lisounkin. Modeling Closed-Loop Mechanisms in Robots for Purposes of Calibration. *IEEE Transactions On Robotics and Automation*, 13(2):218 – 314, April 1997.

[San95] M. Santifaller. *TCP/IP und ONC/NFS in Theorie und Praxis.* Addison-Wesley, Reading, 1995.

[SB96] H.-J. Siegert und S. Bocioneck. *Robotik: Programmierung intelligenter Roboter.* Springer, New York, 1996.

[SB98] J. Smaragdakis und D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of ECOOP'98, European Conference on Object-Oriented Programming*, Seiten 550 – 570, Brüssel, Juli 1998, Springer LNCS 1445.

[SC95] L.M. Saw und C.L. Chuah. A Multitasking PC-based Controller for Flexible Automated Disc Drive Assembly. In *Proceedings of 26th, Int'l Symposium on Industrial Robots, Robotics '95 - Flexible Production Flexible Automation*, Seiten 615 – 620, Singapur, 1995.

[SC96] K.G. Shin und C.-C. Chou. Design and Evaluation of Real-Time Communication for Field Bus-Based Manufacturing Systems. *IEEE Transactions On Robotics and Automations*, 12(3):357 – 366, Juni 1996.

[Sch86] K.J. Schmucker. MacApp: An Application Framework. *Byte*, 11(8):189 – 193, August 1986.

[Sch89] A.-W. Scheer. *CIM: Der computergesteuerte Industriebetrieb*. Springer, Berlin, 1989.

[Sch91] A.-W. Scheer. *Architektur integrierter Informationssysteme*. Springer, Berlin, 1991.

[Sch92] M. Schiebe und S. Pferrer, Editoren. *Real-Time Systems*. Kluwer Academic Publishers, Boston, 1992.

[Sch93] D.C. Schmidt. The Design and Use of the ACE Reactor: an Object-Oriented Framework for Event Demultiplexing. *C++ Report*, 5(7):1 – 14, September 1993.

[Sch95] H.A. Schmid. Creating the Architecture of a Manufacturing Framework by Design Patterns. In *Proceedings of OOPSLA'95, Tenth Annual Conference On Object-Oriented Programming Systems, Languages and Applications*, Volume 30, Seiten 370 – 384, Austin, Oktober 1995. ACM SIGPLAN Notices.

[Sch96a] D.C. Schmidt. *A Family of Design Patterns for Application-Level Gateways, Theories and Practice of Object Systems.* Wiley, New York, 1996.

[Sch96b] D.C. Schmidt. *ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software,* `http://www.cs.wustl.edu/~schmidt/ACE-papers.html` . Washington University, New Orleans, 1996.

[Sch97a] H.A. Schmid. Systematic Framework Design by Generalization. *Communications of the ACM*, 40(10):48 – 51, Oktober 1997.

[Sch97b] D.C. Schmidt. Applying design patterns and frameworks to develop object-oriented communication software. In P. Salus, Editor, *Handbook of Programming Languages*, Volume 1. MacMillan Computer Publishing, 1997.

[Sch98] H.A. Schmid. Framework Design by Systematic Generalization: From Hot Spot Specification to Hot Spot Subsystem. In M. Fayad, D.C. Schmidt, und R. Johnson, Editoren, *Application Frameworks: Problems & Perspectives*. Wiley, New York, 1998.

[SE84] E. Soloway und K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, 10(5):595 – 605, September 1984.

[Sel96] B. Selic. An Architectural Pattern for Real-Time Control Software. In *PLoP96, The Joint Pattern Languages of Programs Conferences*, Illinois, September 1996.

[Sel99] B. Selic. Turning Clockwise: Using UML in the Real-Time Domain. *Communications of the ACM*, 42(10):46 – 54, Oktober 1999.

[SFJ96] D.C. Schmidt, M. Fayad, und R.E. Johnson. Software Patterns, Introduction. *Communications of the ACM*, 39(10):36 – 38, Oktober 1996.

[SGK98] A. Speck, G. Gruhler, und W. Küchlin. Object-Oriented Robot Control Framework. In *Proceedings of IECON'98, 24th Annual Conference of the IEEE Industrial Electronics Society, IES*, Seiten 1663 – 1666, Aachen, 1998. IEEE Press.

[SGW94] B. Selic, G. Gullekson, und P.T. Ward. *Real-Time Object-Oriented Modelling*. Wiley, 1994.

[Sha89] Y.-P. Shan. An Event-Driven Model-View-Controller Framework for Smalltalk. *SIGPLAN Notices*, 24(10):347 – 352, Oktober 1989.

[Shi91] K.G. Shin. Real-Time Communications in a Computer-Controlled Workcell. *IEEE Transactions On Robotics and Automations*, 7(1):105 – 113, Februar 1991.

[SK99] A. Speck und H. Klaeren. RoboSiM: Java 3D Robot Visualization. In *Proceedings of IECON'99 25th Annual Conference of the IEEE Industrial Electronics Society, IES*, Seiten 821 – 826, San Jose, 1999. IEEE Press.

[Som92] I. Sommerville. *Software Engineering*. Addison Wesley, Wokingham, vierte Ausgabe, 1992.

[SP00] A. Speck und E. Pulvermüller. Component Frameworks for Software Generators. In *Workshop des GI Arbeitrskreis 2.1.4 Programmiersprachen und Rechnerkonzepte, Schwerpunkt Softwarekomponenten*, Bad Honef, Mai, 2000.

[Spe96] Special Issue on Software Patterns. *Communications of the ACM*, 39(10):36 – 82, Oktober 1996.

[Spe98] A. Speck. Robot Simulation and Monitoring on Real Controllers (RoboSiM). In *Proceedings of the ESS'98, 10th European Simulation Symposium and Exhibition*, Seiten 482–489, Nottingham, 1998.

[Spe99] A. Speck. RoboSiM – eine Java-basierte Robotervisualisierung. In S. Maffeis, F. Toenniessen, und C. Zeidler, Editoren, *Erfahrungen mit Java*, Seiten 293 – 310. dpunkt, Heidelberg, 1999.

[Spe00] A. Speck. Component-based Control System. In *Proceedings of ECBS'2000, Int'l IEEE Symposium and Workshop on Engineering of Computer Based Systems*, Edinburgh, April, 2000. IEEE Computer Society Press.

[SPG91] A. Silberschatz und P.B. Galvin. *Operating System Concepts*. Addison Wesley, Reading, fünfte Ausgabe, 1998.

[SPM00] A. Speck, E. Pulvermüller und M. Mezini. Reusability of Concerns. In *Proceedings of ECOOP Workshop Aspects and Dimensions of Concerns 2000*, Sophia Antipolis, Cannes Juni, 2000

[SRPC99] A. Speck, A. Rashid, E. Pulvermüller und R. Chitchyan. Individual Software Development in Generative Programming. In *Proceedings GCSE99 Young Researchers Workshop*, Erfurt, September, 1999.

[SS85] W.P. Seering und V. Scheinman. Mechanical Design of an Industrial Robot. In S.Y. Nof, Editor, *Handbook of Industrial Robotics*, Seiten 29 – 43. Wiley, New York, 1985.

[SS95] D.C. Schmidt und P. Stephenson. Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms. In *Proceedings of ECOOP'95, European Conference on Object-Oriented Programming*, Aarhus, 1995, Springer LNCS 952.

[SSDB95] J.A. Stankovic, M. Spari, M. DiNatale, und G.C. Buttazzo. Implications on Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, 28(6):16 – 25, Juni 1995.

[Sto92] R. Stone. Virtual reality and telepresence. *Robotica, Cambridge University Press*, 10:461 – 467, 1992.

[Str93] B. Stroustrup. A History of C++: 1979-1991. *ACM Sigplan Notices*, 28(3):271 – 298, März 1993.

[Str94] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, zweite Ausgabe, 1994.

[STY94] J. Sulkanen, R. Tuokko, und J. Yliollitervo. Flexibility and Cost Efficiency with an Open Multitasking Control Architecture for Robots. In *Proceedings of 25th, Int'l Symposium on Industrial Robots, Robotics '94 - Flexible Production Flexible Automation*, Seiten 375 – 379, Hannover, 1994.

[Szy97] C. Szyperski. *Component Software.* Addison-Wesley, ACM-Press, New York, 1997.

[Tan89] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs, 1989.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Englewood Cliffs, 1992.

[Tay99] K. Taylor. *Project Telerobot: Australia's Telerobot On The Web,* `http://telerobot.mech.uwa.edu.au/`. University of Western Australia, Perth, 1999.

[TS93] R. Tuokko und J. Sulkanen. Commercial Industrial Robot as a Teleoperational Slave – Application Aspects for Model-Based Telerobotics. In *ISMCR'93, Third Intl' Symposium on Measurement and Control in Robotics*, Seiten II–5 – II–10, Turin, 1993.

[TT95] K. Taylor und J. Trevelyan. Australia's Telerobot On The Web. *26th International Symposium on Industrial Robots*, 26:37 – 44, 1995.

[VHK+92] P. Voho, J. Heilala, J. Kemiläinen, A. Pentti, und J. Kyllönen. ACCEMDEI. In *Proceedings of 23rd, Int'l Symposium on Industrial Robots, Robotics '92 - Flexible Production Flexible Automation*, Seiten 159 – 165, Barzelona, 1992.

[War89] P.T. Ward. How to Integrate Object Orientation with Structured Analysis and Design. *IEEE Software*, 38(10):74 – 82, März 1989.

[Wec96a] M. Weck. *Werkzeugmaschinen – Fertigungssysteme, Band 4.1.* VDI Verlag, Frankfurt, Main, vierte Ausgabe, 1996.

[Wec96b] M. Weck. *Werkzeugmaschinen – Robotersysteme, Band 4.2.* VDI Verlag, Frankfurt, Main, vierte Ausgabe, 1996.

[Weg87] P. Wegner. The object-oriented classification paradigm. In B. Shriver und P. Wegner, Editoren, *Research Directions in Object-Oriented Programming*, Seiten 479 – 560. MIT Press, Cambridge, 1987.

[WGM89] A. Weinand, E. Gamma, und R. Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2):63 – 87, April 1989.

[WHW98] C. Wurll, D. Henrich, und H. Woern. Parallel on-line motion planning for industrial robots. In *Proceedings of Robotics 98: The Third ASCE Specialty Conference on Robotics for Challenging Environments*, Albuquerque, 1998.

[Wil91] T. Williams. Real-Time UNIX Develops Multiprocessing Muscle. *Computer Design*, 30(5):26, März 1991.

[WM85] P.T. Ward und S. Mellor. *Structured Development for Real-time Systems.* Prentice Hall, Yourdon Press, Englewood Cliffs, 1985.

[WSW85] H.J. Warnecke, R.D. Schraft, und M.C. Wanner. Mechanical Design of the Robot System. In S.Y. Nof, Editor, *Handbook of Industrial Robotics*, Seiten 44 – 79. Wiley, New York, 1985.

[You94] E. Yourdon. *Object-Oriented Systems Design.* Yourdon Press, Prentice Hall, Englewood Cliffs, 1994.

[Zie97] C. Zieliński. Object-oriented robot programming. *Robotica*, 15(1):41 – 48, Februar 1997.

[Zie99a] J. Ziegler. private Communication, Juni 1999.

[Zie99b] C. Zieliński. The MRROC++ system. In *Proceedings of the RoMoCo 99, First Workshop On Robot Motion And Control*, Kiekrz, Polen, Juni 28 – 29 1999.

Abbildungsverzeichnis

Tabellenverzeichnis