Objektorientierter Entwurf verteilter eingebetteter Echtzeitsysteme auf Basis höherer Petri-Netze

Druckversion mit allen Formelzeichen hier:
http://www.theoinf.tu-ilmenau.de/~nuetzel/promo_nuetzel_print.pdf

Dissertation

zur Erlangung des akademischen Grades
Dr.-Ing.

vorgelegt
der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von Dipl.-Ing. Jürgen Nützel
geb. am 16.2.1967 in Schweinfurt

Gutachter: Prof. Dr.-Ing. habil. Wolfgang Fengler, TU Ilmenau (wiss. Betreuer)
Prof. Dr.-Ing. habil. Hans-Michael Hanisch, Uni Magdeburg
Dr.-Ing. Roger Knorr, Alcatel, USA

Eingereicht am: 18.12.1998
Verteidigt am: 25.5.1999
Prädikat: „summa cum laude“

Autor: Jürgen Nützel

Titel: Objektorientierter Entwurf verteilter eingebetteter Echtzeitsysteme auf Basis höherer Petri-Netze

Kurzbeschreibung:
Derzeit erfolgt der Entwurf, die Validierung und die Implementierung eines verteilten eingebetteten Echtzeitsystems meist völlig voneinander getrennt. Der durchgängige Ablauf wird für eingebettete Systeme praktisch nicht durchgeführt. Die wachsende Komplexität und die steigenden Sicherheitsanforderungen von Echtzeitsystemen werden allerdings in naher Zukunft den Einsatz von integrierenden Entwurfsmethoden zwingend erforderlich machen.

Ziel dieser Arbeit ist es, eine Entwurfsmethode – die Objektnetz-Methodik – zu entwickeln und vorzustellen, die den geforderten durchgängigen Entwurf für verteilte eingebettete Echtzeitsysteme ermöglicht. Die Objektnetz-Methodik verbindet hierzu die graphische Notationsform einer datenflußorientierten Moduldarstellung mit der zustandsorientierten Darstellung hierarchischer erweiterter Zustandsmaschinen. Es werden hierarchische Strukturierungsmöglichkeiten sowohl für die Modulebene als auch für die Zustandsebene angeboten. Für die schrittweise Konkretisierung wird auf das objektorientierte Konzept der Vererbung zugegriffen. Zur formalisierten Notation wichtiger, sicherheitsrelevanter Anforderungen wurde die Objektnetz-Constraint-Sprache entwickelt. Die Objektnetz-Methodik bietet für Constraints, die in dieser Sprache formuliert wurden, den Zugang zu einer automatischen Verifikation.

Damit der Vorgang der Implementierung rückwirkungsfrei erfolgen kann, wurde ein objektorientiertes Framework zur Abstraktion verteilter Rechnerplattformen eingeführt. Über die Klassen dieses Frameworks können alle Spezifika eines verteilten eingebetteten Rechnersystemen gekapselt werden. Durch die Kapselung bleiben plattformabhängige Ansteuerungsdetails der unterschiedlichsten Peripheriekomponenten verborgen.

Die Validierung einer Objektnetz-Spezifikation, die in die zu implementierende Steuerung und in die Umgebung zerfällt, erfolgt auf Basis einer Verhaltensbeschreibung mit höheren Petri-Netzen; Objektnetz-Spezifikationen lassen sich hierzu vollständig mit einer speziellen höheren Petri-Netz-Klasse darstellen. Durch eine Simulation, die mit einer graphischen Animation der gesteuerten Umgebung gekoppelt wird, erfolgt die Validierung der prinzipiellen Funktion. Durch die vollständige Simulation, bei der alle Varianten des Umgebungsmodells durchgespielt werden, lassen sich die Constraints, deren Verhalten ebenfalls als Petri-Netz beschreibbar ist, verifizieren.

Der durchgängige Entwurfsvorgang der Objektnetz-Methodik wird durch die automatische Generierung der Software für die Steuerung abgeschlossen.

Mit dem Object System Specification Inventory (OSSI) wurde begleitend zu dieser Arbeit eine Tool-Unterstützung realisiert.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fachgebiet Rechnerarchitektur des Institutes für Theoretische und Technische Informatik der Technischen Universität Ilmenau. Mit diesem vorangestellten Kapitel möchte ich allen danken, die zum Zustandekommen dieser Arbeit maßgeblich beigetragen haben.

An erster Stelle möchte ich meinem wissenschaftlichen Betreuer Prof. Dr.-Ing. habil. Wolfgang Fengler danken. Er hat in mehrfachem Sinne dazu begetragen, daß diese Arbeit in der vorliegenden Form zustande kam. Mitte 1990 konnte er mich dazu bewegen, parallel zu meiner Industrietätigkeit ein Fernstudium in Ilmenau zu beginnen. Diese postgraduale Studium schloß ich 1994 mit dem Diplom ab. Daraufhin offerierte er mir eine Stelle in seinem Fachgebiet. Auf dieser Mitarbeiterstelle konnte ich mich ab Oktober 1994 hauptamtlich mit der in dieser Arbeit behandelten Thematik befassen. Er ließ mir dabei stets die benötigte Unterstützung zukommen und gewährte mir bei der Bearbeitung einen großzügigen Freiraum.

Zu besonderem Dank bin ich Dr. Thomas Böhme verpflichtet. Trotz seiner Habilitation stand er mir in langen Abenden und Nächten mit seinem mathematischen Sachverstand zur Seite. Seine Art und Weise komplexe Probleme zu analysieren, ermöglichte es mir, den Petri-Netz-Abschnitten die notwendige Schärfe und Klarheit zu verleihen. Die Zusammenarbeit mit Thomas brachte nicht nur in der bearbeiteten Thematik interessante Resultate, auch meine prinzipielle Herangehensweise bei der Bearbeitung ähnlicher Fragen konnte dabei eine Verbesserung erleben. Ebenso möchte ich Dr. Bernd Däne danken. Er stand mir jederzeit mit Rat und Tat zur Seite. Frau Dr.-Ing Susan Schwuchow gilt gleichfalls mein Dank. Sie lieferte mir wertvolle Tips, die mir an entscheidenden Stellen weiter halfen.

Nicht vergessen werden soll die Mühe, die sich Prof. Dr.-Ing. habil. Hans-Michael Hanisch von der Universität Magdeburg gemacht hat. Prof. Hanisch – mein zweiter Gutachter – gab mir im Rahmen mehrerer persönlicher Gespräche eine Vielzahl sehr entscheidenter Hinweise. Gleichzeit möchte ich Dr.-Ing. Roger Knorr von Alcatel USA dafür danken, daß er sich bereiterklärt hat, neben seiner industriellen Tätigkeit die von mir eingereichte Dissertationsschrift zu begutachten.

Den Herren Berger und Niklas von der Firma Mühlbauer möchte ich für ihren Anstoß zu dieser Arbeit danken. Herr Berger initiierte eine Studie, die in Folge die Entwicklung der Objektnetz-Methodik auslöste.

Schließlich möchte ich allen Studenten danken, die im Rahmen der von mir betreuten Studien- oder Diplomarbeiten bzw. während einer hilfswissenschaftlichen Tätigkeit zur Objektnetz-Methodik beigetragen haben. Den Auftakt lieferten die beiden Diplomarbeiten von Ralf Schulze und Thomas Richter. Ralf erforschte den Stand der Technik. Thomas legte die Grundlagen für die Plattformabstraktion. Zuvor arbeiteten Ralf und Thomas zusammen mit Jens Hösel im Rahmen ihrer Studienarbeiten an der CAN-Bus-Thematik. Die Studienarbeit von Mario Holbe lieferte das Datenklassen-Konzept mit den Value-Checkern. Die Studienarbeit von Dirk Kahnert realisierte die Datenbankankopplung, die für jedes moderne Entwicklungs-werkzeug zwingend ist. Alexander Schmidt verfaßte eine Studienarbeit, die die Auflösung der Hierarchie von Objektnetzen zur Aufgabe hatte. Aktuell arbeitet er an einer Diplomarbeit, die sich mit heuristischen Mapping-Verfahren befaßt. Marko Langbein erstellte als Studienarbeit den graphischen Editor für hierarchische Zustandsmaschinen. In seiner angeschlossenen Diplomarbeit befaßt er sich mit Sequenzdiagrammen zur graphischen Darstellung von Constraints. Wolfgang König realisiert als wissenschaftliche Hilfskraft den hierarchischen Objektnetz-Editor. In einer angeschlossenen Diplomarbeit wird er sich mit der Systematik bei der Erstellung von graphischen Umgebungsanimationen befassen. Alexander Fleischer hat begonnen, die Code-Generierung zu realisieren.

Ebenso sollten meinem Kollegen Klaus-Dieter Fritz ein besonderer Dank gelten. Er war maßgeblich daran beteiligt, daß die Objektnetz-Methodik im Rahmen eines durch das Land Thüringen geförderten Vorhabens eine Weiterführung findet.

Last but not least will ich an dieser Stelle meiner Frau meinen besonderen Dank aussprechen. Nicht nur weil sie mit mir den Kampf gegen den Fehlerteufel und meine Schachtelsätze aufgenommen hat, sondern weil sie auch in besonders anstrengenden und schwierigen Phasen zu mir und meiner Arbeit gehalten hat. Mit der Geburt unserer Tochter Laura (1996) und unseres Sohnes Sebastian (1998) machte sie die Promotionsphase zu einem ganz besonders fruchtbaren Lebensabschnitt. Hierbei muß ich mich noch bei Laura entschuldigen, daß ich in der Endphase der Promotion so wenig Zeit für sie hatte *„Papa ´pielen! Papa ´hause!”*.

Inhaltsverzeichnis

Abkürzungsverzeichnis

- **Eigene Abkürzungen:**

AONC	*Abstract Object Net Class*	abstrakte Objektnetz-Klasse
ARP	*Asynchronous Receive Port*	asynchroner Empfangsport
ASC	*Actuator/Sensor Class*	Aktor/Sensor-Klasse
ASP	*Asynchronous Send Port*	asynchroner Sendeport
CFU	*Core Function Unit*	Rechnerkernfunktionseinheit
CAI	*Communication Abstraction Interface*	Kommunikationsabstraktionsinterface
CIC	*Communication Interface*	Kommunikationsschnittstelle
CON	*Concurrent Object Net*	
EONC	*Elementary Object Net Class*	elementare Objektnetz-Klasse
HONC	*Hierarchical Object Net Class*	hierarchische Objektnetz-Klasse
ML	*Message Link*	Nachrichtenverbindung
ON	*Object Net*	Objektnetz
ONCL	*Object Net Constraint Language*	Objektnetz-Constraint-Sprache
ONI	*Object Net Instance*	Objektnetz-Instanz
OPNTcl	*Object Petri Nets based on Tcl*	
PA	*Port Assignment*	Portzuordnung
PAF	*Platform Abstraction Framework*	Framework zur Plattformabstraktion
PE	*Port Export*	Portexportierung
PIC	*Process Interface Coupler*	Prozeßschnittstellenkoppelelement
PP	*Parameter Port*	Parameterport
SRP	*Synchronous Receive Port*	synchroner Empfangsport
SSP	*Synchronous Send Port*	synchroner Sendeport
TC	*Transition Condition*	Transitionsbedingung
TF	*Transition Function*	Transitionsfunktion
TONI	*Textual Object Notation Interface*	Textuelle Objektnotationsschnittstelle
TOPN	*Timed Object Petri Net*	zeiterweitertes Objekt-Petri-Netz
WT	*Waiting Time*	Wartezeit
ZM		Zustandsmaschine

• Fremde Abkürzungen (Deutsch):

EPN	elementares Petri-Netz
EZA	Echtzeitautomat
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug

• Fremde Abkürzungen (Englisch):

ADC	*Anlog-Digtal-Converter*
ANSI	*American National Standard Institute*
ASI	*Actuator-Sensor-Interface*
CAN	*Controller Area Network*
CSMA/CA	*Carrier Sense Multiple Access with Collision Avoidence*
CPU	*Cental Processing Unit*
DAC	*Digital-Analog-Converter*
FIP	*Factory Instrumentation Protocol*
FMS	*Field Bus Message Specification*
LON	*Local Area Network*
MAP	*Manufacturing Automation Protocol*
MSC	*Message Sequence Chart*
MSI	*Medium Scale Integration*
OMG	*Object Management Group*
OMT	*Object Modeling Technique*
ROOM	*Real-Time Object-Oriented Modeling*
RTOS	*Real-Time Operating System*
SQL	*Structured Query Language*
Tcl	*Tool Command Language*
TCTL	*Temporal Computation Tree Logic*
Tk	*Toolkit*
UML	*Unified Modeling Language*
VHDL	*VHSIC (Very High Speed Integrated Circuit) Hardware Description Language*

1. Kapitel

Einführung und Motivation

Der Entwurf eingebetteter Echtzeitsysteme verläuft häufig sehr unsystematisiert und mit einer viel zu geringen Produktivität ab. Bestimmte Schritte im Entwurfsablauf erfolgen strukturiert. Andere, wie z.B. die werkzeugunterstützte Validierung, fehlen dafür oft gänzlich. Der Kontakt zur Firma Mühlbauer machte dem Autor diese Problematik überaus deutlich. Die Entwicklung von Steuerungssoftware für Chip-Bonder erfolgt bei Mühlbauer mit Hilfe von Zustands-automaten. Diese Zustandsautomaten werden direkt in ANSI-C codiert und über einfache Bit-Merker gekoppelt. Systeme mit einer wachsender Anzahl an Zustandsautomaten werden allerdings rasch unübersichtlich. Sie sind ohne den Einsatz von geeigneten, zusätzlichen Hilfsmitteln kaum noch beherrschbar. Insbesondere bereitet die Erkennung von Neben-läufigkeiten und Verklemmungen Probleme. Eine Möglichkeit zur automatischen Analyse solcher Systeme besteht im Einsatz von Petri-Netzen.

Dies beschreibt die Ausgangslage bei der Erstellung einer Studie [NütBöh 96] für die Firma Mühlbauer. Die Arbeit an dieser Studie brachte verschiedene Erkenntnisse: Automaten eignen sich für den praktischen Entwurf komplexer eingebetteter Echtzeitsysteme; für das tiefere Verständnis innerer Zusammenhänge können Petri-Netz gewinnbringend eingesetzt werden. Sie bleiben somit speziellen Teilschritten im Entwurfsablauf vorbehalten.

Ausgehend von diesen Aussagen, setzte sich der Autor das Ziel eine Entwurfsmethodik für verteilte eingebettete Echtzeitsysteme zu entwickeln. Die Methodik soll den Anwender vom Entwurf über die Validierung bis zur Implementierung begleiten. Eines der wesentlichen Anliegen war dabei die Durchgängigkeit der Methodik. Dem Anwender muß ein Pradigmen-wechsel z.B. beim Übergang zur Implementierung erspart bleiben. Der Entwurf soll über eine leicht verständliche modul- und zustandsorientierte graphische Notation erfolgen. Diese Notationsformen versprechen bei den Hardware-orientierten Entwicklern eingebetteter Systeme die größte Akzeptanz. Die Hierarchie und Modulorientierung halten dabei komplexe Entwürfe überschaubar. Die Objekt-Technologie steuert moderne und effektive Mechanismen zur Wiederverwendung und Abstraktion bei. Module werden als mehrfach verwendbare Klassen entworfen. Die Schnittstellenelemente dieser „Modulklassen” besitzen eine leicht verständliche graphische Repräsentation.

Neben dem Entwurf der Steuerungssoftware wird die Erstellung komplexer Testszenarien für die Umgebung ermöglicht. Zum Zwecke der Validierung erfolgt eine gemeinsame Simulation von Steuersoftware und Modellumgebung. Die graphische Simulation beschränkt sich nicht auf die Notationsebene; eine zusätzliche Animation der Modellumgebung ist ebenfalls möglich. Zusätzlich können wichtige, sicherheitsrelevante Anforderungen einfach formalisiert werden. Ihre Verifikation erfolgt automatisiert. Die gesamte Validierung ruht auf einem bekannten mathematischen Apparat – den Petri-Netzen. Die Abstraktion der Spezifika unterschiedlicher Plattformen ermöglicht eine automatische Code-Generierung. Eine optimierte Code-Generierung wird durch die Integration eines Framework ermöglicht, welches die plattformspezifisch erstellten Ansteuerungssequenzen verwaltet. Dieser, noch teilweise während der Forschungsarbeiten modifizierte Forderungskatalog formuliert die Aufgabenstellung für die vorliegende Arbeit.

Der Hauptteil der Arbeit beginnt mit Kapitel 2, in welchem der Leser in die verwendete Begriffswelt eingeführt wird. Es werden zuerst die betrachteten Zielplattformen vorgestellt. Darauf aufbauend werden softwaretechnische Basistechniken diskutiert, die sich für den Entwurf verteilter Systeme eignen. Aus der sich anschließenden Betrachtung unterschiedlicher konventioneller und objektorientierter Methoden wird der bereits erwähnte Forderungskatalog abgeleitet. In Kapitel 3 erfolgt die Umsetzung dieser Forderungen; die Objektnetz-Methodik wird entwickelt. Nach der Diskussion der Entwurfsprinzipien erfolgt die Vorstellung der Entwurfselemente. Kapitel 3 endet mit der Beschreibung des Frameworks zur Kapselung unterschiedlicher Zielplattformen. In Kapitel 4 werden die Vererbungs- und Hierarchiekonstrukte der Objektnetze aufgelöst, bevor in Kapitel 5 das Verhalten dieser verflachten Objektnetz-Spezifiktionen mit Petri-Netzen formal beschrieben wird. Die formale Darstellung der Objektnetze mit einer speziellen höheren Netzklasse ermöglicht eine Validierung der vollständigen Spezifikation. Die generierten Petri-Netze spiegeln das detaillierte zeitliche Verhalten auf der Zielplattform wider, da sie das Abarbeitungsmodell und die spezifischen Zeitparameter berücksichtigen. Abgeschlossen wird Kapitel 5 durch die Petri-Netz-Darstellung der formalen und applikationsspezifischen Anforderungen. Neben der Verifikation dieser Constraints über die sogenannte vollständige Simulation, erfolgt in Kapitel 6 die Verifikation applikationsunabhängiger Anforderungen: speziell für verteilte Steuerungen muß deterministisches Verhalten gewährleistet werden. Zum Abschluß von Kapitel 6 wird die Objektnetz-Simulation mit Animation vorgestellt. Kapitel 7 stellt schließlich das Funktionsprinzip der automatischen Code-Generierung für Objektnetze vor; Zustandsmaschinen werden zusammen mit einem automatisch generierten Scheduler in ANSI-C umgesetzt.

Nach der Zusammenfassung findet der Leser im Anhang die Beschreibung der realisierten Entwurfsumgebung. Abgeschlossen wird der Anhang mit zwei Anwendungsbeispielen.

2. Kapitel

Der Entwurf von verteilten eingebetteten Echtzeitsystemen

Die Definition des Begriffes Echtzeit bildet den thematischen Einstieg für das vorliegende Kapitel. Begleitend hierzu werden der Aufbau allgemeiner Echtzeitsysteme skizziert und ihre Bedeutung für sicherheitskritische Anwendungen unterstrichen. Der zweite Abschnitt erweitert diese Betrachtung um verteilte und eingebettete Echtzeitsysteme. Es folgt die Analyse typischer Echtzeitsysteme dieser Kategorie. Hierbei werden die betrachteten Systeme in das Ebenenmodell der Automatisierung eingeordnet und weiter klassifiziert. Die Klassifikation umfaßt die Hardware-Plattform der Echtzeitsysteme mit ihren eingebetteten Rechnerknoten und dem verbindenden Kommunikationsnetzwerk. Als Kommunikationsnetzwerk werden typische Vertreter der Feldbusse erläutert, wie CAN und PROFIBUS.

Den Abhandlungen über die verteilte Hardware eingebetteter Echtzeitsysteme folgt im dritten Abschnitt die Analyse der softwaretechnischen Grundprinzipien zu ihrer Umsetzung. Im Detail werden dort die Begriffe Prozeß, Prozeßkommunikation und Prozeßsynchronisation diskutiert.

Der vierte Abschnitt bildet mit einer Bewertung bekannter Entwurfsmethoden den Schwerpunkt des zweiten Kapitels. Als Einstieg werden die betrachteten Entwurfsverfahren in das vereinfachte Phasenmodell der Software-Entwicklung eingeordnet. Es wird dabei ein Spektrum von konventionellen und objektorientierten Methoden vorgestellt, welches in seiner Breite eine Vielzahl wünschenswerter Eigenschaften für den durchgängigen Entwurf verteilter eingebetteter Echtzeitsysteme bereitstellt.

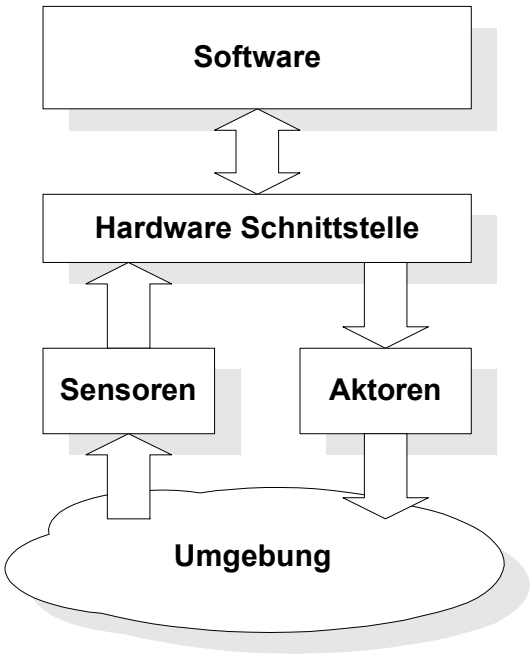
Das zweite Kapitel wird mit den Schlußfolgerungen und den daraus abgeleiteten Zielen für diese Arbeit abgeschlossen. Hierbei werden alle bis dorthin vorgebrachten Aspekte erneut aufgegriffen, mit der Absicht, das Anforderungsprofil für eine moderne und durchgängige Entwurfsmethode für verteilte eingebettete Echtzeitsysteme abzuleiten.

2.1 Echtzeitsysteme

Echtzeitsysteme sind informationstechnische Anwendungen, wie sie in vielen Lebensbereichen zu finden sind. Dort steuern bzw. regeln sie technische Prozesse oder kontrollieren sicherheitskritische Systeme, wie sie z.B. in der Medizin oder in der Fahrzeugtechnik vorkommen. Die Anwendungsgebiete mit spezifischem Sicherheitsaspekt verdienen dabei eine besondere Beachtung. Ein Versagen des Echtzeitsystems kann einen hohen Schaden an Mensch und Material nach sich ziehen. Folglich muß der Entwurf von sicherheitskritischen Anwendungen mit großer Sorgfalt und Methodik durchgeführt werden.

Abbildung 2.1 zeigt schematisch den Aufbau eines allgemeinen Echtzeitsystems. Eine Softwareanwendung arbeitet auf einer Rechnerplattform, die in speziellen Anwendungsfällen auch verteilt sein kann. Diese Plattform bildet die Hardware-Schnittstelle zu den Sensoren und Aktoren, die wiederum den Kontakt zur Umgebung herstellen. Die Umgebung wird durch den technischen Prozeß charakterisiert, welcher durch das Echtzeitsystem gesteuert oder geregelt wird.

Abbildung 2.1
Die Grundelemente eines Echtzeitsystems



Neben verschiedenen strengen Definitionen über den Echtzeitbetrieb von Softwaresystemen z.B. [DIN 44300] findet sich in [Laplante 93] eine allgemeiner gehaltene Begriffsfestlegung von Echtzeitsystemen, die auch im folgenden vertreten wird:

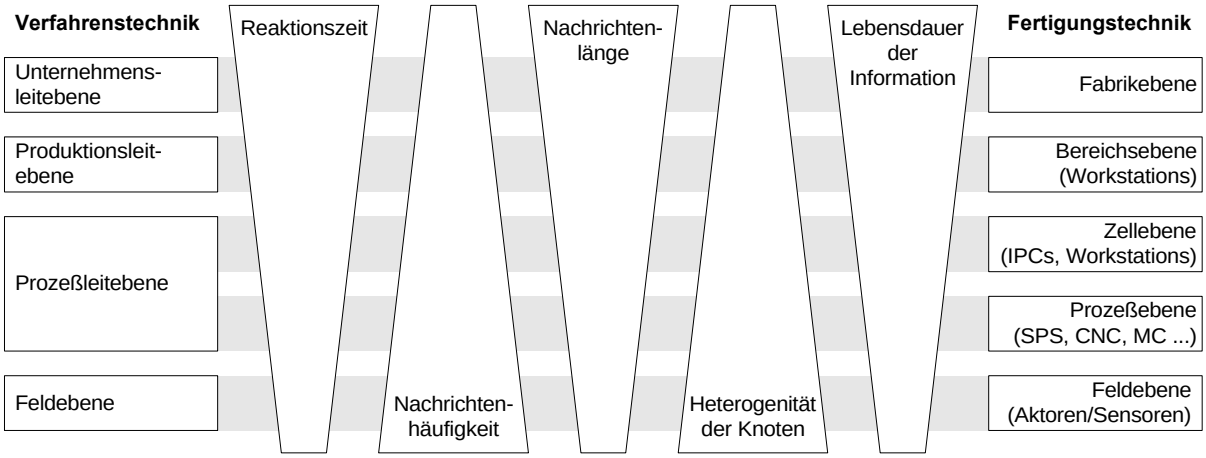
> *„A real-time-system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure. A failed system is a system which cannot satisfy one or more of the requirements laid out in the formal system specification."*

Dieses Zitat unterstreicht die Bedeutung der Rechtzeitigkeit von Systemreaktionen auf externe Signale als das zentrale Merkmal eines Echtzeitsystems. Kapitel 2.3 greift dieses Merkmal bei der Erläuterung von softwaretechnischen Grundprinzipien für die Erstellung von Echtzeitsystemen erneut auf.

2.2 Verteilte eingebettete Echtzeitsysteme

Die in der vorliegenden Arbeit behandelten verteilten, eingebetteten und heterogenen Echtzeitsteuerungssysteme, die in einer Vielzahl unterschiedlicher Anwendungsdomänen auftreten, werden im folgenden in ein Ebenenmodell eingeordnet. Abbildung 2.2 [Arnold 93] zeigt zwei Ebenenmodelle für die Automatisierung, die linke Seite aus Sicht der Verfahrenstechnik [Polke 94], die rechte Seite aus der Sicht der Fertigungstechnik. Die weiteren Betrachtungen beschränken sich exemplarisch auf das weiter verbreiterte Ebenenmodell der Fertigungstechnik. Auf jeder Ebene dieses Modells sind Automatisierungsfunktionen realisiert, die sich entsprechend der hierarchischen Anordnung in der Komplexität und der Art des Informationsflusses unterscheiden. Der Informationsfluß erstreckt sich von der Feldebene über die übergeordneten Ebenen bis hin zur Fabrik- bzw. Unternehmensleitebene. Innerhalb einer Ebene erfolgt der Informationsfluß anwendungsoptimiert.

Abbildung 2.2
Ebenenmodell der Automatisierung aus Sicht der Verfahrens- und Fertigungstechnik

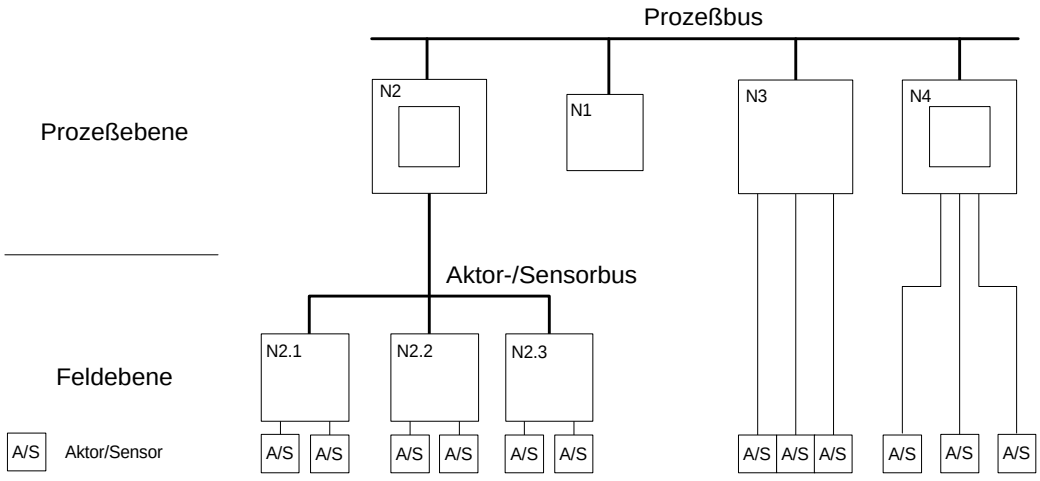


Die höchste Hierarchieebene ist die Fabrikebene. Sie stellt die Verbindung zum kaufmännischen Bereich, zur Entwicklung oder zur Qualitätssicherung her. Auf dieser Ebene werden sehr große Datenmengen verwaltet. Die realisierten Funktionen sind zeitunkritisch und azyklisch. Die Kommunikation erfolgt typischerweise über MAP (*Manufacturing Automation Protocol*) bzw. LAN (*Local Area Network*). In der Bereichsebene werden mehrere Automatisierungszellen zusammengeführt. Hier liegt der Schwerpunkt in der Überwachung, Lenkung und Optimierung von Automatisierungsanlagen. Die Kommunikationsstruktur entspricht der der Fabrikebene, wobei der Datenumfang auf dieser Ebene geringer ist. Einrichtungen, wie z. B. Montage- und Fertigungszellen, sind in der Zellebene zusammengefaßt. Dabei fallen mittlere Datenmengen an, deren Übertragung (überwiegend azyklisch) durch spezielle Prozeßbussysteme erfolgt, wie beispielsweise durch den PROFIBUS [Bender 92], [DIN 19245], [Knorr 94]. Zelleinrichtungen bestehen aus vernetzten Aktor/-Sensormodulen. Die Verwaltung dieser Module vollzieht sich auf der Prozeßebene. Zwischen den Aktor-/Sensorbaugruppen werden geringe Datenmengen unter Einhaltung bestimmter Zeitschranken ausgetauscht. Hergestellt wird die

Verbindung zum technischen Prozeß über die Feldebene. Als typische Kommunikationsmedien auf der Prozeß- und Feldebene haben sich Feldbussysteme, wie z. B. CAN (*Controller-Area-Network*) [Etschberger 94], [Lawrenz 98], [Blume 95] oder ASI (*Actuator-Sensor-Interface*) [KrMaBe 95] etabliert.

Eine weitere Aufteilung der Feldbussysteme in spezielle Prozeß- bzw. Aktor/Sensor-Busse (vgl. Abbildung 2.5) ist zweckmäßig. Abbildung 2.3 zeigt beispielhaft die Struktur eines typischen verteilten eingebetteten Echtzeitsystems. Die beteiligten Rechnerknoten kommunizieren über den Prozeßbus bzw. einen Aktor-/Sensorbus. Einige Knoten (N1und N2) besitzen keine Aktoren bzw. Sensoren. Sie erfüllen reine Verarbeitungsaufgaben.

Abbildung 2.3
Struktur einer verteilten eingebetteten Echtzeitsteuerung mit hierarchischer Struktur



Aus der noch folgenden Definition für eingebettete Systeme lassen sich für das Ebenenmodell verteilter eingebetteter Systeme diverse Einschränkung ableiten. Die Fabrik- und Bereichsebene entfallen, da die Rechner in diesen Ebenen keine eingebetteten Systeme sind; Die Verarbeitungseinheiten eingebetteter Systeme treten für den Anwender nicht nach außen in Erscheinung. Des weiteren ist ein verteiltes eingebettetes System typischerweise räumlich stark begrenzt (z.B. ein Straßenfahrzeug). Es kann folglich auf eine Zelle innerhalb des Ebenen-modells konzentriert werden. Eine separate Zellebene ist somit ebenfalls nicht notwendig.

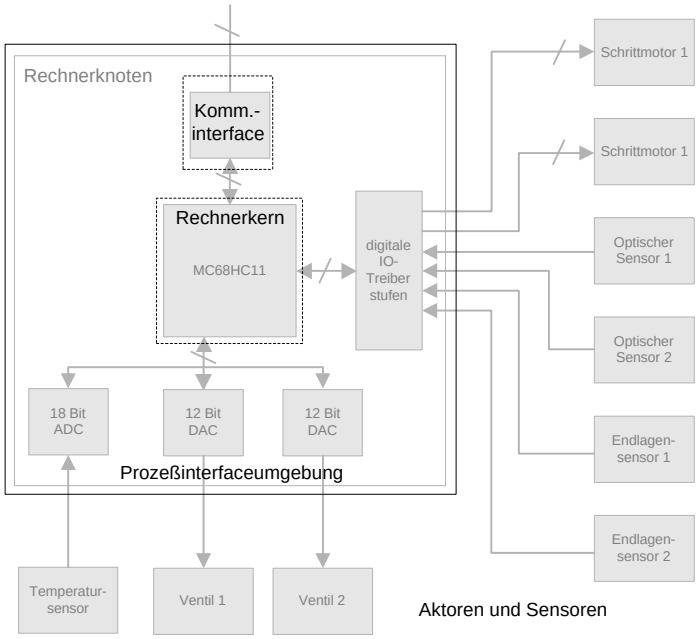
2.2.1 Klassifikation der Rechnerknoten

So wie das gesamte verteilte Echtzeitsystem ein eingebettetes System darstellt, so sind auch alle beteiligten Rechnerknoten der Kategorie *embedded systems* zuzuordnen. Eine allgemeine Definition für *embedded systems* wird in [Cooling 91] formuliert: „Ein *embedded system* ist dadurch charakterisiert, daß die zentrale Verarbeitungseinheit nicht als solche für das System kennzeichnend ist, sondern nur ein Teil des Systems darstellt". Da diese Aussage sehr allgemein

gefaßt ist, wird im folgenden die tiefergehende Beschreibung aus dem *Free On-Line Dictionary of Computing* [FOLDOC 98] für die weiteren Betrachtungen zugrunde gelegt:

> *„Hardware and software which forms a component of some larger system and which is expected to function without human intervention. A typical embedded system consists of a single-board microcomputer with software in ROM, which starts running some special purpose application program as soon as it is turned on and will not stop until it is turned off (if ever).”*

Abbildung 2.4
Unterteilung eines Rechnerknotens (ein Beispiel) der Kategorie *embedded system*



Aufbauend auf diese Beschreibung läßt sich eine genauere Untergliederung der Rechnerknoten einführen. Als zentrale Verarbeitungseinheit existiert ein **Rechnerkern**, typischerweise realisiert durch Mikrocontroller-, Signalprozessor- (DSP) oder Mikroprozessorschaltkreise. Mit weniger Intelligenz ausgestattete Spezialcontrollerschaltkreise sind ebenfalls denkbar. Der Rechnerkern ist in eine **Prozeßinterfaceumgebung** eingebettet, und besitzt mindestens ein **Kommunikationsinterface**.

- **Rechnerkern:**

Das Herzstück des Rechnerkerns bildet der Controllerschaltkreis (in Abbildung 2.4 z.B. ein 8-Bit-Controller MC68HC11 von Motorola [MOTOROLA 98]). Er ist optional mit Speicher- und Peripheriemodulen ausgestattet. Die Kopplung dieser Baugruppen mit dem Controller entscheidet über ihre Zuordnung zum Rechnerkern oder zur Prozeßinterfaceumgebung. Peripherieschaltkreise, die über den Adreß- und Datenbus mit dem Controller verbunden sind, werden dem Rechnerkern zugeordnet. Controllerexterne Speichermodule (RAM, ROM, EEPROM, Flash-EPROM usw.) sind ebenso dem Rechnerkern zugeordnet. Peripherie-

schaltkreise, deren Steuerregister in den Adreßraum des Controllers eingeblendet sind (*memory-mapped* E/A), zählen ebenfalls zum Rechnerkern [HenPat 93].

Der Umfang der Funktionalität des jeweiligen Rechnerkernbereiches ist stark realisierungsabhängig. Oft besitzt der Rechnerkern E/A-Kanäle für eine direkte Sensor- und Aktoransteuerung. Außerdem kann der Kern Kommunikationsschnittstellen besitzen. Die möglichen Verlagerungen der Funktionalität in den Rechnerkern soll in Abbildung 2.4 durch die Umrandungen mit unterbrochenen Linien verdeutlicht werden.

- **Prozeßinterfaceumgebung:**

Die Prozeßinterfaceumgebung ist gekennzeichnet durch die Anpassung der zum Rechnerkern hinführenden bzw. vom Rechnerkern ausgehenden Signale entsprechend den E/A-Spezifikationen der anzusteuernden Aktor-/Sensorelemente. Hardwaretechnisch kann ein komplexes Prozeßinterface als eine Kette, bestehend aus Umsetz-, Verstärker- und Kanalverwaltungsgliedern, beschrieben werden. Die Reihenfolge und die Existenz jedes einzelnen Gliedes ist stark von der zu realisierenden Prozeßschnittstelle abhängig. Zur Realisierung werden vorwiegend MSI- (*Medium Scale Integration*) Schaltkreise eingesetzt, wie z. B. ADC, DAC oder Multiplexer (MUX). Mit dem Rechnerkern erfolgt die Kopplung der Schaltkreise der Prozeßinterfaceumgebung über explizit als E/A-Schnittstellen ausgewiesene Ports des Kerns. Derartige E/A-Schnittstellen sind z. B. E/A-Ports, ADC-Eingänge, serielle Interfaces u. a.. Die Prozeßinterfaceumgebung ist der Schaltungsteil, der verstärkt Änderungen während des Entwurfs neuer Aktor-/Sensorsysteme unterliegt. Die E/A-Kopplung ermöglicht dabei eine flexible Anpassung.

- **Kommunikationsinterface:**

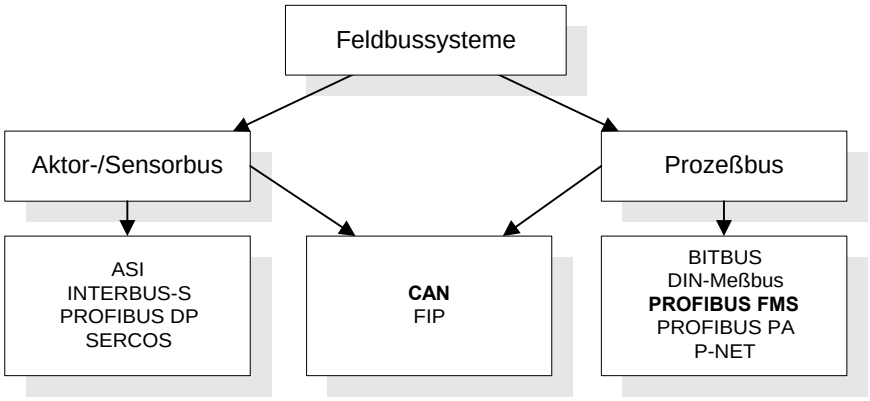
Da die beteiligten Rechnerknoten Komponenten eines verteilten, eingebetteten Systems‘ sein können, besitzen diese eine oder mehrere Kommunikationsschnittstellen. Als Interfaces werden typischerweise Feldbusadapter (z.B. für den CAN-Bus [Etschberger 94]) eingesetzt. Sie stellen die Verbindung zum Kommunikationsnetzwerk her. Ihre Komplexität differiert sehr stark. Das Spektrum erstreckt sich über einfache Treiberbausteine bis hin zu speziellen Kommunikations-controllern, die z. T. Hardware-Implementierungen der jeweiligen Schicht-1- und Schicht-2-Protokolle bereitstellen [NütFen 95b].

2.2.2 Klassifikation des Kommunikationsnetzwerkes

Besteht ein verteiltes System aus mehr als zwei Rechnerknoten, so sind einfache serielle 1:1-Verbindungen zwischen den Knoten (z.B. nach RS-232 oder RS-485) ungeeignet, da für jeden Kommunikationspartner ein eigenes Kommunikationsinterface notwendig werden würde. Feldbussysteme bieten für verteilte Echtzeitsysteme eine kostengünstige Lösung. Das Schema in Abbildung 2.5 zeigt eine Aufstellung der verbreitesten Feldbussysteme und ihre Zuordnung in die Kategorien Aktor-/Sensorbus bzw. Prozeßbus nach [Busse 96].

Der Aktor-/Sensorbus ist gekennzeichnet durch kurze definierte Reaktionszeiten. Es werden Datenblöcke geringer Länge übertragen. Master-Knoten verwalten zentral mehrere Slave-Knoten, welche nur mit der notwendigen Intelligenz zur Ansteuerung von Aktoren und Sensoren ausgestattet sind. Die dienstorientierte Kommunikation wird auf ein Minimum reduziert. Als typisches Aktor-/Sensor-Bussystem sei hier der ASI-Bus [KrMaBe 95] erwähnt.

Abbildung 2.5
Einordnung der Feldbussysteme



Durch eine dienstorientierte Kommunikation zwischen mehreren intelligenten Rechnerknoten (Master-Knoten) ist der Prozeßbus gekennzeichnet. Dabei werden größere Datenmengen als bei einem Aktor-/Sensorbus ausgetauscht. Auf Basis dieser Bussysteme lassen sich dezentrale intelligente Steuerungssysteme aufbauen. Als Beipiel hierfür läßt sich der PROFIBUS-FMS (FMS – *Field Bus Message Specification*) anführen.

CAN und FIP (*Factory Instrumentation Protocol*) [FIP 88] lassen sich aufgrund ihrer Eigenschaften sowohl auf der Aktor-/Sensor- als auch auf der Prozeßebene einsetzen. Da LON (*Local Operating Network*) [Dietrich 98] vorwiegend im Bereich der Gebäudeleittechnik eingesetzt wird, wird dieses System in der Klassifizierung nicht berücksichtigt.

Der Ursprung des CAN in der Automobilindustrie liefert dem Anwender ein breites Angebot an preisgünstigen Schnittstellenbausteinen und Controllern mit integrierter Kommunikationsschnittstelle [Lawrenz 98]. Diese gute Hardware-Unterstützung macht den CAN zum bevorzugten Bussystem für verteilte eingebettete Echtzeitsysteme. Folglich werden CAN und PROFIBUS-FMS, der in der Automatisierungstechnik starke Verbreitung gefunden hat, an dieser Stelle beispielhaft näher erläutert.

- **Topologie:**

CAN besitzt physikalisch und logisch (auf Protokollebene) eine Bustopologie. Im Gegensatz dazu bildet der PROFIBUS-FMS physikalisch eine Bus- und logisch eine Ringstruktur.

- **Buszugriffssteuerung:**

Beide Systeme stellen physikalisch einen Bus dar, bei denen im Unterschied zu RS-232-Systemen eine Buszugriffssteuerung notwendig ist. Die Buszugriffsart der im System vorhandenen Rechnerknoten stellt ein wichtiges Kriterium bei der Generierung von Kommunikationssoftware dar. Die wichtigsten Zugriffsmechanismen auf Feldbusebene werden im folgenden erläutert:

- Dezentrale Buszuteilung am Beispiel des PROFIBUS-FMS ➪ deterministischer Zugriff: Mehrere Master- und mehrere Slave-Knoten können als Teilnehmer angeschaltet sein. Die Master-Knoten bilden einen logischen Ring. Das Senderecht wird zyklisch in Form eines Tokens zwischen den Master-Knoten weitergegeben (*token passing*). In dem festgelegten Zeitraum, die sogenannte Tokenhaltezeit, kann der Master, der aktuell das Token besitzt, mit den Slave-Knoten kommunizieren. Die Kommunikation kann nur durch die Master-Knoten initiiert werden; Slave-Knoten reagieren entsprechend. Diese Beziehung zwischen Master- und Slave-Knoten wird auch als Client-Server-System bezeichnet.

- Dezentrale Buszuteilung am Beispiel des CAN-Bus ➪ stochastischer Zugriff: Physikalisch und logisch bildet der CAN-Bus eine Busstruktur. Als Teilnehmer sind mehrere Master-Knoten, aber auch Slave-Knoten möglich. Sogenannte SLIO (*Serial Linked Input Output*) werden als Slave-Knoten eingesetzt. Sie sind nur mit einer Intelligenz zur Beantwortung von Master-Anforderungstelegrammen ausgestattet. Der Buszugriff beruht auf dem CSMA/CA-Konzept (*Carrier Sense Multiple Access with Collision Avoidance*). Durch bitweises Arbitrieren über den Nachrichtenidentifier, durch den unterschiedliche Nachrichtenprioritäten realisiert werden, wird eine Kollision vermieden. Der Vorteil dieses Verfahrens ist es, daß ein sendebereiter Knoten jederzeit senden kann, ohne auf die Erteilung des Senderechts warten zu müssen. Aufgrund dieses quasi stochastischen Buszugriffs sind exakte Aussagen über die Zeitdauer der Rahmenübertragung nur mit erhöhten Aufwand möglich [WaLuSt 92], [Staub 95]. Durch die Einbindung einer über-geordneten Zeitsteuerung wird ein determiniertes Verhalten auf Applikationsebene ermöglicht [NüBlFe 97].

2.3 Konzepte verteilter nebenläufiger Echtzeitsysteme

Nach der Klassifikation typischer Hardware-Plattformen für verteilte eingebettete Echtzeitsysteme folgt die Betrachtung der elementaren softwaretechnischen Grundprinzipien für deren Realisierung. In [Hüsener 94] werden die Begriffe **Prozeß**, **Prozeßkommunikation** und **Prozeßsynchronisation** als etablierte und zweckmäßige Grundkonzepte für nebenläufige Echtzeitsysteme vorgestellt. Die Kenntnis dieser Grundkonzepte bildet die Basis für das Verständnis von bestehenden bzw. noch zu entwerfenden Entwurfsmethoden für verteilte Echtzeitsysteme.

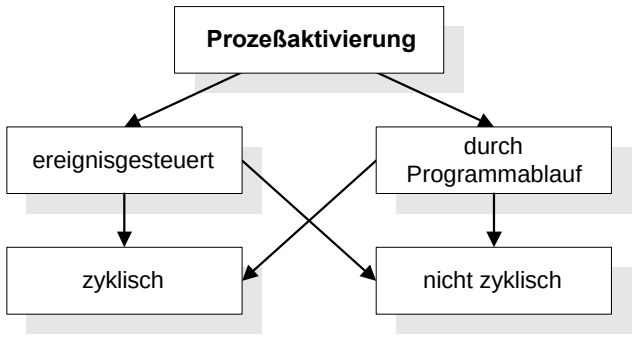
2.3.1 Prozesse und ihre Aktivitäten

Typischerweise werden Abläufe, die wiederholt abgearbeitet werden, softwareseitig durch sogenannte Abschnitte realisiert. Diese Abschnitte lassen sich in nicht weiter zerlegbare atomare Einheiten, die Aktivitäten genannt werden, zerlegen. Mehrere Aktivitäten werden in der Regel zu einem Prozeß zusammengefaßt. In verteilten eingebetteten Echtzeitsystemen finden im allgemeinen mehrere Aktivitäten nebenläufig zueinander statt. Nebenläufigkeit besteht, wenn die Aktivitäten sowohl parallel als auch in einer beliebigen Folge sequentiell ausgeführt werden können. Die Nebenläufigkeit der Aktivitäten wird durch die Kommunikation und Synchronisation der Prozesse, die die Aktivitäten enthalten, beschränkt.

Prozesse [Hüsener 94] in Echtzeitsystemen sind durch mehrere Randbedingungen ausgezeichnet. Zu nennen sind zum einen die Art und Weise ihrer Aktivierung bzw. Erzeugung, zum anderen das Vorhandensein von Zeitschranken für ihre Aktivierung, Ausführung, Unterbrechung und Terminierung.

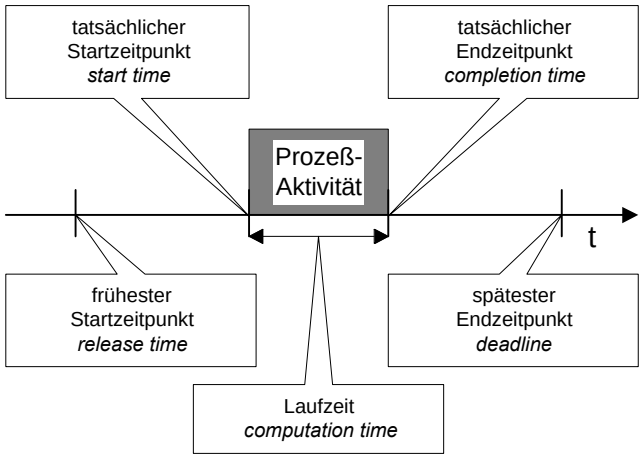
Die Aktivitäten der Prozesse können durch ein Ereignis bzw. durch den Programmablauf gestartet werden (vgl. Abbildung 2.6). Sie können zyklischer bzw. nicht zyklischer Natur sein. Zyklische Prozesse werden innerhalb einer vorgegebenen Periodendauer genau einmal immer wieder aktiviert. Ein Prozeß, der z.B. jede 20 ms ein bestimmtes Sensorsignal aufbereiten und zur Verfügung stellen muß, wird als zyklischer ereignisgesteuerter (durch das Signal des Sensors) Prozeß bezeichnet.

Abbildung 2.6
Die Aktivierung von Prozessen



Innerhalb eines Echtzeitsystems existiert mindestens ein Prozeß, welcher speziellen zeitlichen Restriktionen genügen muß. Abbildung 2.7 zeigt alle relevante Zeiten [HerHom 89], für die Ausführung einer Aktivität innerhalb eines solchen Prozesses.

Abbildung 2.7
Relevante Zeiten für die Ausführung einer Aktivität



Der früheste Startzeitpunkt markiert den Zeitpunkt, an dem der Prozeß frühestens (durch ein Ereignis oder den Programmablauf) aktiviert werden kann. Der späteste Endzeitpunkt ist der Zeitpunkt, an dem der Prozeß vollkommen ausgeführt sein muß. Er muß somit so aktiviert werden, daß seine tatsächliche Laufzeit zwischen diesen beiden Zeitschranken liegt. Muß ein Prozeß nur im Mittel diese Zeitschranken einhalten,

so spricht man von weichen Zeitschranken. Interaktive Benutzeroberflächen sind z. B. durch solche weiche Zeitschranken gekennzeichnet. Harte Zeitschranken liegen vor, wenn ein Prozeß die vorgegebenen Schranken auf keinen Fall verletzen darf. Sicherheitskritische Anwendungen stehen und fallen oft mit der strikten Einhaltung bzw. Verletzung dieser harten Zeitschranken.

Ist der tatsächliche Zeitbedarf für die Abarbeitung der Aktivitäten unbekannt, so ist es unmöglich, präzise Aussagen über das zeitliche Verhalten des Gesamtsystems zu machen. In diesen Fall kann mit einer Abschätzung (*worst case*) der Abarbeitungszeiten, die ausreichend Reserve beinhalten, gearbeitet werden. Bei dieser Verfahrensweise ist jedoch ein höherer Ressourcenbedarf zu erwarten.

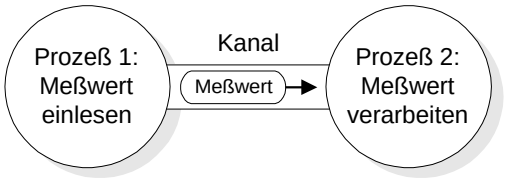
2.3.2 Kommunikation und Synchronisation von Prozessen

Typisch für Echtzeitsysteme ist, daß zwischen nebenläufigen Vorgängen Abhängigkeiten generiert werden. Dies wird als Synchronisation bezeichnet [Engesser 88]. Werden z.B. Meßwerte von einem Prozeß eingelesen und von einem zweiten verarbeitet, so müssen die bei-den beteiligten Prozesse sich synchronisieren und miteinander kommunizieren. Der einlesende Prozeß übergibt dabei den Meßwert an den verarbeitenden Prozeß (⇨ Kommunikation). Dieser Prozeß wird folglich nur dann aktiviert, wenn er einen neuen Wert erhalten hat (⇨ Synchronisation). Dieses kurze Beispiel verdeutlicht, daß Synchronisation und Kommunikation nicht isoliert betrachtet werden können.

Bei der Auswahl einer für verteilte eingebettete Systeme problemgerechten Kommunikations- und Synchronisationsart stehen die Konzepte

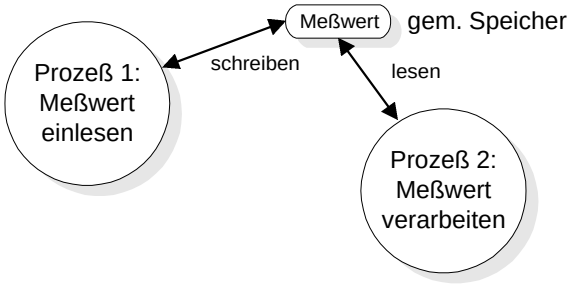
- **gemeinsamer Speicher** (*shared memory*) und
- **Botschaften** (*message passing*) zur Auswahl.

Abbildung 2.8
Botschaftenkonzept



Beim Botschaftenkonzept senden sich die beteiligten Prozesse Infor-mationen zu. Die Informations-übertragung erfolgt über einen Kanal (vgl. Abbildung 2.8). Es kann dabei zwischen **synchroner** und **asynchroner** Kommunikation unterschieden werden. Bei synchroner Kommunikation erfolgt eine Synchronisation zwischen Sender und Empfänger derart, daß der Sender solange blockiert ist, bis er die Empfangsbestätigung des Empfängers erhalten hat. Bei asynchroner Kommunikation ist der Sender unmittelbar nach dem Absenden der Nachricht wieder frei. Die Kommunikation zwischen verteilten Rechnerknoten der bereits vorgestellten Plattformen erfolgt hardwaretechnisch gesehen nach diesem Botschaftenkonzept (z.B. durch CAN). Es wäre daher naheliegend, dieses Konzept für alle, auch für die lokal kommunizierenden Prozesse anzuwenden.

Abbildung 2.9
Kommunikation über gemeinsamen Speicher



Beim Konzept mit gemeinsamen Speicher (z.B. durch Semaphoren [Dijkstra 68]) wird davon ausgegangen, daß alle beteiligten Prozesse auf einen gemeinsamen Speicher schreibend und lesend zugreifen können. Kom-munikation und Synchronisation erfolgen dadurch, daß die beteiligten Prozesse gemeinsame Daten besitzen, die sie lesen und im wechselseitigen Ausschluß beschreiben (vgl Abbildung 2.9). Wird der gemeinsame Speicher durch die Hardware nicht zur Verfügung gestellt, so kann eine Software-Emulation des Speichers erfolgen. In diesem Fall werden die Synchronisation und Kommunikation über gemeinsame Speicherzellen auf das Botschaftenkonzept abgebildet. Umgekehrt kann bei Vorhandensein gemeinsamen Speichers auch das Botschaftenkonzept emuliert werden.

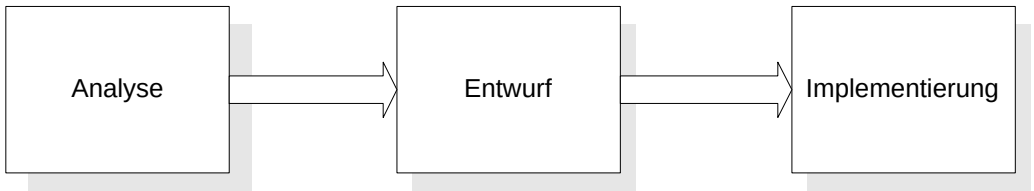
Nach [Leler 90] sind allerdings Systeme, die mit dem Konzept des gemeinsamen Speichers arbeiten, aufgrund des auftretenden Nichtdeterminismus schwieriger zu handhaben:

> *„In general, the shared memory model is somewhat easier to program, but the resulting programs are more likely to contain unwanted nondeterminism, while the distributed memory model is harder to program but more likely to run correctly."*

2.4 Gegenüberstellung bekannter Entwurfsmethoden

Nach den verschiedenen Betrachtungen zu Echtzeitsystemen, deren Hardware-Plattformen und softwaretechnischen Grundprinzipien, folgt eine Gegenüberstellung von bekannten Methoden zur Software-Entwicklung. Nach [Stein 97] besteht eine Methode aus einem Beschreibungs-mittel (graphisch oder textuell) und einem zugeordneten Vorgehensmodell.

Abbildung 2.10
Vereinfachtes Phasenmodell der Software-Entwicklung

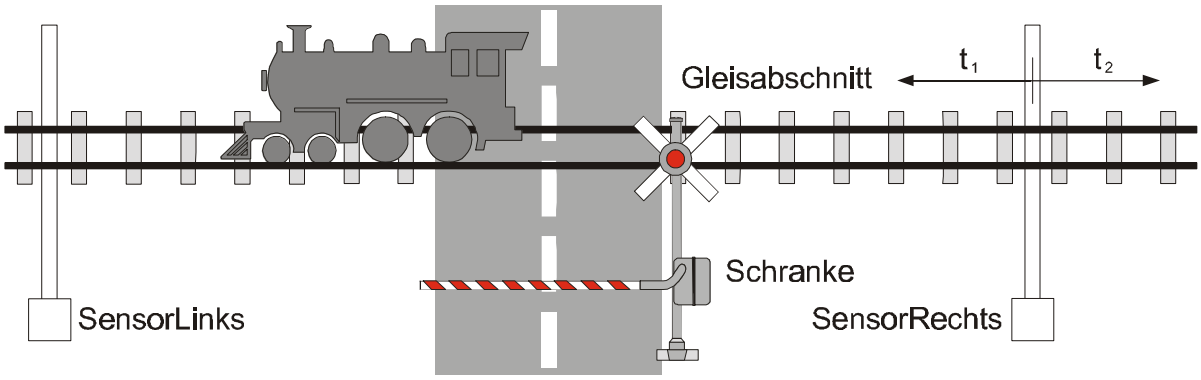


Die vorgestellten Methoden haben ihren Schwerpunkt in der Entwurfsphase (vgl. Abbildung 2.10). Das ursprüngliche Wasserfallmodell nach [Royce 70] bildet die Basis für das gezeigte Phasenmodell. Die Analyse- und Implementierungsphase werden nur tangiert. In der Analysephase, die der Entwurfsphase vorangestellt ist, versucht der Software-Designer die

teilweise noch sehr wagen informellen Anforderungen (das *Was*) in geeigneter Form zu notieren. Diese Notationen bilden schließlich den Ausgangspunkt für die Entwurfsphase (das *Wie*), in der alle spezifischen Details eingebracht und schließlich so weit verfeinert werden, daß eine Implementierung möglich wird.

Das aus der Literatur bekannte Beispiel des *Generalized Railroad Crossing* [HeLy 94], welches ein Standardbeispiel für den Entwurf von Echtzeitsystemen ist, soll die Gegenüberstellung verschiedener Entwurfsmethoden begleiten.

Abbildung 2.11
Ein Gleisabschnitt, in dem sich ein beschrankter Bahnübergang befindet



Das Sollverhalten für die Steuerung einer Schranke (vgl. Abbildung 2.11) wurde in der Analysephase durch den folgenden informellen Wortlaut festgelegt:

*„In einem Teilabschnitt einer Eisenbahnstrecke befindet sich ein beschrankter Bahnübergang. Züge fahren in beiden Richtungen durch den Abschnitt. Sensoren stellen jede Ein- und Ausfahrt eines Zuges fest. Die Aufgabe besteht nun in der Entwicklung einer Steuerung für die Schranke, so daß zwei Forderungen eingehalten werden. Die Schranke ist geschlossen, solange sich ein Zug auf dem Bahnübergang befindet (**Sicherheit** des Systems). Die Schranke ist geöffnet, wenn sich keine Züge im Bereich des Übergangs befinden (**Lebendigkeit** des Systems). Diese Forderungen werden durch zwei Zeitschranken t_1 und t_2 genauer festgelegt. Wenn ein Zug in den Teilabschnitt fährt, muß spätestens nach t_1 Zeiteinheiten die Schranke geschlossen sein. Wenn der Zug den Abschnitt verläßt, muß spätestens nach t_2 Zeiteinheiten die Schranke wieder geöffnet sein."*

Die Gegenüberstellung unterschiedlicher Methoden, soll in ihrer Breite eine Vielzahl wünschenswerter Eigenschaften vorstellen, welche für den Entwurf verteilter eingebetteter Echtzeitsysteme bereitstehen. Auch wenn einige der vorgestellten Methoden nicht primär für die hier behandelte Domäne entwickelt wurden, so besitzen sie dennoch Eigenschaften, die in einen durchgängigen Entwurf eingebetteter Systeme aufgenommen werden sollten.

Bei der Methodenauswahl wurde berücksichtigt, inwieweit die ausgewählten Methoden die Konzepte Nebenläufigkeit, Kommunikation und Synchronisation unterstützen. Zusätzlich wurde darauf Wert gelegt, daß die Methode ein graphisches Beschreibungsmittel bereitstellt. Die

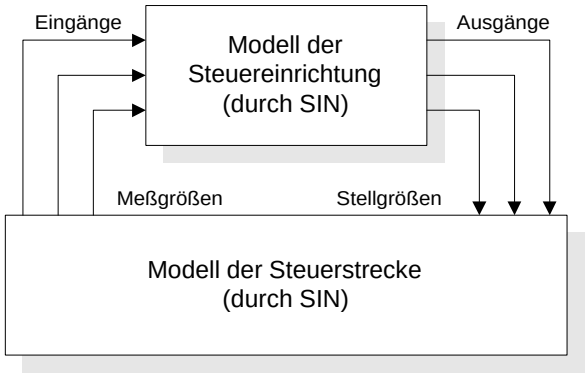
Gegenüberstellung beginnt mit den konventionellen, nicht-objektorientierten Methoden und endet mit den modernen objektorientierten Methoden.

2.4.1 Konventionelle Methoden

- **Steuerungstechnisch interpretierte Petri-Netze:**

„C.A. Petri schlug vor, Prozesse als Systeme von Situationen und Ereignissen zu diskutieren; d.h. die Situationen und Ereignisse als Grundbegriffe zu betrachten [Petri 62]. Ereignisse setzen einerseits die Gültigkeit bestimmter (ihnen zugeordneten) Situationen voraus; andererseits verändern eingetretene Ereignisse die Gültigkeit von Situationen, d.h., sie produzieren (ihnen nachgeordnete) Folgesituationen. Damit kann man einen Prozeßverlauf als eine Folge von Situationen auffassen, die nacheinander eingenommen werden. Die Änderungen der Situation werden durch Ereignisse herbeigeführt." [KönQuä 88]

Abbildung 2.12
Modell aus Steuerung und Steuerstrecke

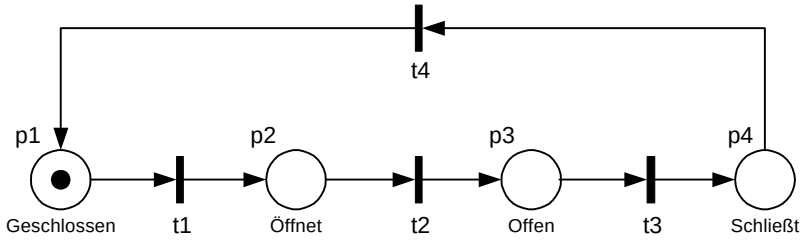


Die Modellierung technischer Systeme (z.B. ein System aus Steuereinrichtung und Steuerstrecke, vgl. Abbildung 2.12) mit Petri-Netzen setzt voraus, daß die Petri-Netz-Klasse über ein Eingangs- und Ausgangsverhalten verfügt. Steuerungstechnisch interpretierte Petri-Netze (SIN) [KönQuä 88] stellen eine Erweiterung des klassischen Petri-Netz-Konzeptes um dieses Ein- und Ausgangsverhalten dar. Den Plätzen werden Boole'sche Variablen zugeordnet, die mit den binären Ausgängen korrespondieren. In [FenPhi 91] wird ein ähnliches Konzept beschrieben, bei dem alle zugeordneten Variablen – während der Platz mit mindestens einer Marke markiert ist – den Wert Eins besitzen. Den Transitionen werden Boole'sche Gleichungen mit logischen Verknüpfungen der binären Eingänge zugeordnet. Die Transition schaltet, wenn sie Konzession hat und die logische Verknüpfung erfüllt ist.

Abbildung 2.13 zeigt das Petri-Netz des Streckenmodells einer vereinfachten Bahnschranke. Es stellt das formalisierte Verhaltensmodell, der z.B. im Pflichtenheft verbal formulierten Verhaltensbeschreibung dar. In [Quäck 92] wird schrittweise beschrieben, wie aus dem Modell der Strecke (Umgebung), das Petri-Netz-Modell der Steuerung abgeleitet wird. Beide Netzmodelle werden über eine dritte Netzebene gekoppelt. In dieser Koppelebene werden die binären Sensoren und Aktoren durch sogenannte EPN (elementare Petri-Netze [KönQuä 88]) modelliert. Ein EPN besitzt genau zwei Plätze, die mit „O" und „1" bezeichnet sind. Die Markierung der Plätze im Steuerungs- bzw. Streckennetz beeinflußt die Markierung der EPN, die wiederum die Schaltfähigkeit der Transitionen im Strecken- bzw. Steuerungsnetz beeinflussen.

Einen anderen Ansatz verfolgen Hanisch und Rausch [Rausch 96], [HaLöRa 96], indem sie aus dem Petri-Netz-Modell der Strecke das Modell der Steuerung automatisch ableiten.

Abbildung 2.13
SIN-Streckenmodell der Bahnschranke



Das resultierende Netz des Gesamtsystems bildet schließlich die Grundlage für die simulative und analytische [Starke 90] Validierung, die Auskunft über verschiedene Systemeigenschaften geben sollen. Hierzu zählen z.B. Lebendigkeit und Konfliktfreiheit. Für die Modellierung von Echtzeitsystemen können z.B. die Transitionen mit zusätzlichen Schaltintervallen [Merlin 74] oder Schaltdauern [Ramchandani 74] bewertet werden. Ebenso können die Kanten mit Zeitintervallen beschriftet werden [Hanisch 92].

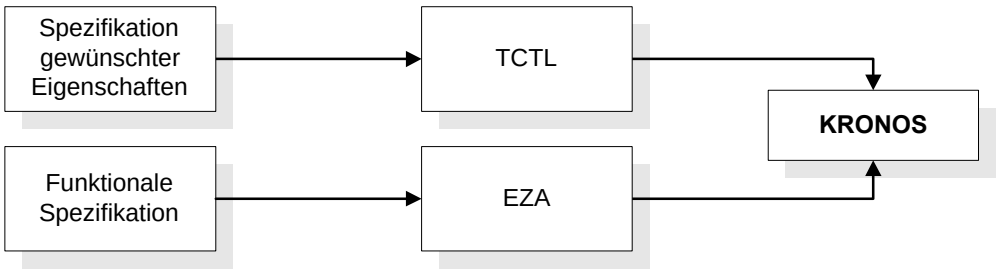
Sollen Systeme entworfen werden, die einen höheren Informationsverarbeitungsanteil besitzen, so eignen sich die vorgestellten Petri-Netze nur bedingt. Sogenannte High-Level-Netze [Jensen 92], [GenLau 79] besitzen durch ihre Markenattributierung die Möglichkeit, zusätzlich zum Steuerfluß auch Datenhaltung und Datenfluß zu modellieren. High-Level-Netze lassen sich in der Regel in Platz-Transitionsnetze transformieren, wodurch die dort entwickelten analytischen Untersuchungen wieder eingesetzt werden können.

Bewertung: Petri-Netze überzeugen durch ihre einfache graphische Darstellung und ihrem mächtigen mathematischen Analyseapparat. Der hohe Grad der Abstraktion vom Zielsystem verbunden mit dem Aufwand für die Implementierung, reduzieren jedoch erheblich die Akzeptanz bei Embedded-Anwendungen.

- **Echtzeitautomaten:**

Echtzeitautomaten (EZA) [AlDi 94] sind eine Erweiterung der endlichen (abstrakten) Automaten [Krapp 88], [Philippow 87] um reellwertige Variablen, die Uhren darstellen und gleichmäßig ansteigen. Diese Uhren können durch Übergänge des Automaten zurückgesetzt und gestartet werden. Zusätzlich können die Zustandsübergänge von den Uhrenständen abhängig gemacht werden. Zur Analyse von EZA werden in [KoPrEn 97] drei leistungsfähige Rechnerwerkzeuge genannt: HyTech [HYTECH 98], KRONOS [KRONOS 98] und UPPAAL [UPPAAL 98]. KRONOS – als exemplarischer Vertreter ausgewählt – verifiziert die gewünschten Eigenschaften, die in TCTL (*Timed Computation-Tree Temporal Logic* [HNSY 94]) formuliert werden, gegenüber der funktionalen Spezifikation des Systems, welche mittels EZA modelliert wurde (vgl. Abbildung 2.14).

Abbildung 2.14
Verifikation der funktionalen Spezifikation gegenüber den gewünschten Eigenschaften



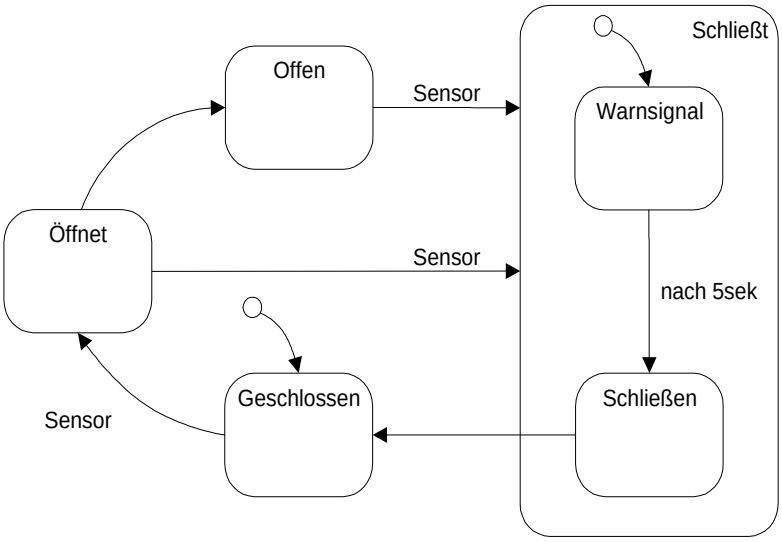
Die Sicherheitseigenschaft und die Lebendigkeitseigenschaft für das Bahnschrankenbeispiel werden durch zwei TCTL-Ausdrücke im KRONOS Format angegeben. Die in den beiden Ausdrücken referenzierten Systemzustände beziehen sich auf den, durch Komposition der einzelnen Automaten der funktionalen Spezifikation (für den Zug, die Steuerung und die Schranke) entstandenen EZA. Seine Zustände, Zustandsübergänge und Invarianten (logische Ausdrücke über die Uhrenvariablen, die erfüllt sein müssen, solange sich das System in dem zugeordneten Zustand befindet), sind in der zweiten KRONOS-Eingabedatei notiert. Die Eingabedatei und die Erläuterung der Zustände sowie eine ausführliche Beschreibung des gesamten Bahnschranken-beispiels findet sich in [KoPrEn 97].

Bewertung: Die Verbindung einer automatenbasierten funktionalen Spezifikation mit einer formalen Spezifikation für applikationsspezifische Anforderungen ist für den Entwurf und die Verifikation sicherheitsrelevanter Echtzeitsysteme zwingend. Die relativ geringe Mächtigkeit der Echtzeitautomaten und die Kompliziertheit der TCTL-Formeln reduzieren jedoch die praktische Einsatzfähigkeit des Verfahrens. Der direkte Bezug der Formel auf die Automatenzustände verhindert zusätzlich, daß vor der Existenz der vollständigen funktionalen Spezifikation applikationsspezifische Anforderungen formalisiert werden können.

- **Statecharts:**

Statemate [Harel 90] der Firma i-Logix [ILOGIX 98] ist eine Software-Entwicklungsumgebung für den Entwurf komplexer, reaktiver Systeme. Das grundlegende Konzept in Statemate ist die Modellierung eines reaktiven Systems durch *Activitycharts*, *Modulecharts* und *Statecharts*. Activitycharts – die Datenflußdiagramme [DeMarco 78] in Statemate – beschreiben hierbei die funktionale Sicht auf das System. Als Sprachmittel dienen *Data Stores*, *Activities* und *Control Activities*. Das Verhalten der *Control Activities* wird durch sogenannte *Statecharts* beschrieben. Zur Beschreibung der physikalischen Sicht des Systems, also aus welchen Software- bzw. Hardware-Komponenten das System besteht, sowie deren Zusammenhänge werden in Statemate *Modulecharts* eingesetzt. *Modulecharts* besitzen nur deskriptive aber keine operationale Semantik. Besondere Bedeutung über die *Activitycharts* und *Modulecharts* hinaus haben die von David Harel [Harel 87] zusammen mit seinen Mitarbeitern am Weizmann Institut in Israel entwickelten Statecharts.

Abbildung 2.15
Geschachteltes Zustandsdiagramm der Bahnschranke nach Harel



Mit den *Statecharts* (vgl. Abbildung 2.15) wird das Verhalten des Systems beschrieben. Den *Control Activities* der *Activitycharts* wird jeweils ein Statechart zugeordnet, das das *Wie* der *Control Activities* beschreibt. *Statecharts* sind eine sehr weitgehende Verallgemeinerung der Darstellung endlicher Automaten als Zustandsübergangsdiagramme. *Statecharts* erweitern gewöhnliche Zustandsautomaten um hierarchische und parallele Zustände sowie Broadcasting-Kommunikation. Neben Zuständen und Transitionen stehen *Events*, *Conditions* und strukturierbare Variablen als weitere Sprachelemente zur Verfügung. Die Details des Entwurfs, die nicht direkt in den Charts beschrieben werden können, werden in Formulare (*Forms*) eingetragen und im *Data-Dictionary* abgelegt. *Statecharts* und Zustände können rekursiv geschachtelt werden. D.h., ein Zustand bzw. der Automat, den dieser Zustand enthält, kann in einem neuen *Statechart* beschrieben werden.

Bewertung: Statecharts eignen sich hervorragend, komplexe reaktive Systeme graphisch zu modellieren. Sie ermöglichen durch die hierarchische Verfeinerung von Zuständen eine kompakte Darstellung auch von großen Systemen. Die Kopplung der *Control Activities* auf der Ebene der *Activitycharts* erfolgt allerdings direkt über Bezeichner, die in den *Statecharts* an Zustandsübergängen definiert wurden. Es wird keinerlei graphische Repräsentation dieser Kopplung angeboten. Dieser Mangel reduziert sehr stark die Anschaulichkeit der *Activitycharts*.

2.4.2 Objektorientierte Methoden

Das objektorientierte Paradigma durchzieht seit den frühen Neunzigern sämtliche Bereiche der Informatik. Das Prinzip, ein komplexes System durch eine Anzahl kleinerer Komponenten – die Objekte – aufzubauen, hat sich als praktikable und realitätsnahe Sicht erwiesen. Bei dieser Herangehensweise wird versucht, Objekte mit gleichen Aufgaben zu Klassen zusammenzufassen. Dabei wird die eigentliche Realisierung der Funktionalität des Objekts, welche in seiner Klasse

hinterlegt ist, vollständig durch die Klassenschnittstelle abstrahiert. Die Interaktion eines Objektes mit seiner Umwelt (die anderen Objekte des Systems) erfolgt ausschließlich über diese Schnittstelle. Durch diese Schnittstelle wird die gesamte Funktionalität eines Objekts den Objekten seines Umfelds zur Verfügung gestellt. Der Abruf einer speziellen Funktionalität erfolgt durch das Senden einer definierten Botschaft (bzw. durch Aufruf einer Methode aus der Schnittstelle) an das bereitstellende Objekt. Dieses Abstraktionsprinzip ermöglicht es z.B., die interne Realisierung einer speziellen Funktionalität zu ändern bzw. zu erweitern (z.B. durch Überschreiben), ohne daß dies Einfluß auf das Umfeld eines Objekts hat. Weitere Vorzüge und Charakteristika des objektorientierten Paradigmas können in [Booch 91] oder [Booch 95], welche hier als exemplarische Vertreter der unüberschaubaren Literatur auf diesem Gebiet stehen, nachgelesen werden.

Die speziellen Anforderungen, die sich aus dem Bereich der verteilten Echtzeitsysteme ableiten lassen, können nicht durch alle objektorientierten Entwurfsmethoden zufriedenstellend gelöst werden. Es sind die speziellen Eigenschaften der zu modellierenden Prozesse, die ihr Gegenstück in der gewählten Methode finden müssen. Das Hauptaugenmerk sollte auf die Möglichkeiten zur Modellierung dynamischer Aspekte gerichtet werden. Wobei zwischen Objekt- und Systemdynamik zu unterscheiden ist.

Unter Objektdynamik ist die Modellierung eines Objektlebenszyklus, und die damit verbundenen Sachverhalte zu verstehen. Der Objektlebenszyklus beschreibt, wie in Abhängigkeit von der Historie des Objekts, auf Ereignisse und Botschaften zu reagieren ist. Es gilt zu unterscheiden, wie der Botschaftenempfang organisiert ist und wie sich Aspekte der Nebenläufigkeit in der Methode wiederfinden. Die Interobjektparallelität, die Nebenläufigkeit zwischen verschiedenen Objekten, und die Intraobjektparallelität, die Nebenläufigkeit innerhalb eines Objekts, sollen für die verschiedenen Methoden diskutiert werden [Schulze 97]. Zusätzlich zu den im Anschluß aufgeführten Methoden bzw. Modellen findet sich in [Schulze 97], welche in Vorbereitung dieser Arbeit entstand, die Diskussion weiterer Verfahren: *Object-Oriented Analysis and Design* (OOAD) von Booch [Booch 95], *Objectlifecycles* (OL) von Shlaer und Mellor [ShlMel 92] und *Concurrent Object Oriented Design* (COOD) von der Ruhr-Universität Bochum [Hüsener 95]. Für einen weiteren sehr umfassenden Überblick über die Vielzahl an objektorientierten Methoden kann zusätzlich [Stein 97] herangezogen werden. Eine Darstellung vieler nichtobjektorientierter Basismethoden findet sich in [AbeLem 98].

- **Object Modeling Technique (OMT):**

OMT wurde am *General Electric Research and Development Center* durch die Autoren des Buches *„Object-Oriented Modeling and Design"* [Rumbaugh 91] entwickelt und veröffentlicht. Die langjährigen Erfahrungen im Bereich der relationalen Datenbanken finden sich in den genutzten Datenflußdiagrammen wieder. Die Methode zeichnet sich durch eine reichhaltige und aussagekräftige Notation aus [Schulze 97]. In [AwKuZi 96] dient die OMT als Basis für die OCTOPUS Methode, welche ein praxisorientierte Umsetzung des objektorientierten Entwurfsparadigmas speziell für Echtzeitsysteme darstellt.

Grundsätzlich versteht die OMT die objektorientierte Herangehensweise als eine Teilung in **Analyse**, **Systemdesign**, **Objektdesign** und **Implementation**. Die Analyse beinhaltet die Abstraktion des zu lösenden Problems der realen Welt in eine Form von Objekten und Beziehungen. Das Ergebnis der Analyse ist die Beschreibung, *was* das zu entwickelnde System machen soll, nicht *wie* dies geschehen soll. Im Systemdesign entsteht eine Architektur des Zielsystems auf einem hohen Level. Weiterhin gehören zum Systemdesign die Unterteilung in Subsysteme und das Hinzufügen weiterer Wunscheigenschaften. Ziel des Objektdesigns ist die Ausfüllung der gefundenen Klassen mit Datenstrukturen und Algorithmen, die der Lösung des Problems dienen. Die Implementierung ist die Übersetzung des modellierten Systems in eine Programmiersprache, Datenbank oder Hardware und schließt den Systemmodellierungsvorgang ab.

In [Booch 91] wird ebenfalls die Durchgängigkeit des objektorientierten Software-Entwicklungszyklus (*software development life cycle for object-oriented design*) gefordert. Booch betont dabei, daß die beiden Phasen der Analyse und des Designs (Entwurf) iterativ verlaufen und stark verzahnt geschehen. Er leitet dabei auf der Seite 201 die Forderung ab, daß die Ergebnisse aus der Anforderungsanalyse direkt in die Entwurfsphase übertragbar sein müssen:

> *„Practically speaking, this means that the product of object-oriented analysis may be used almost directly at the start of the object-oriented design process. The object-oriented designer refines these products by inventing new abstractions and mechanisms that use these classes and objects in clever way."*

Auch die OMT verwendet ähnlich wie Booch [Booch 91] zur Modellierung der Objektdynamik Zustandsdiagramme nach Harel (vgl. Abbildung 2.15). Der Zustandsgraph ist der Klasse zugeordnet und beschreibt somit das Verhalten aller Objekte dieser Klasse. Es finden sich alle Eigenschaften, wie sie auch bei Booch beschrieben werden. Transitionen markieren den Zustandsübergang, Events lösen diese aus, Aktionen können Transitionen oder Zuständen zugeordnet sein, und es existiert die Unterscheidung in Aktion und Aktivität. Für Zustände lassen sich getrennte Aktionen für das Erreichen (*entry*) und das Verlassen (*exit*) des Zustandes definieren. Befindet sich das Objekt in einem Zustand, kann für diesen eine Aktivität festgelegt werden (*do*). Transitionen können durch *guard functions* bewertet werden. Interne Aktionen innerhalb eines Zustandes entsprechen Eigenschleifen unter Auslassung der entry- und exit-Aktionen. Events, die eine interne Aktion auslösen, bewirken das Ausführen der zugeordneten Aktion, aber keinen Zustandswechsel. Events werden grundsätzlich als Broadcasting-Sendungen verstanden. Objekte, die für dasselbe Event sensitiv sind, reagieren alle auf das aufgetretene Ereignis. Ein Event, das im aktuellen Zustand des Objektes keine Wirkung hat, wird ignoriert. Aktivitäten können durch Events unterbrochen werden. Aktionen haben keinen Zeitverbrauch und müssen so auch nicht gesondert berücksichtigt werden. Automatische Transitionen (*lambda transitions*) haben kein Triggerevent. Sie werden ausgeführt, wenn die Aktivität des Vorzustands abgeschlossen ist und eine eventuell zugeordnete *guard function „true*" ergibt.

Alle Objekte im System werden als nebenläufig untereinander beschrieben. Nebenläufigkeit innerhalb eines Objektes wird zugelassen und ist mit Hilfe eines verzweigten Zustandsgraphen darstellbar. Im funktionalen Modell findet sich die Unterscheidung nach Aktor und Speicher. Ein Aktor produziert und konsumiert Daten, auf die Daten eines Speichers kann man nur zugreifen.

Bewertung: Die Durchgängigkeit der OMT, von der Analyse über das Design bis zur Implementierung, ist sehr stark ausgeprägt. Das objektorientierte Paradigma bietet dabei dem Anwender größtmögliche Anschaulichkeit. Die Möglichkeit der Validierung gegenüber einer Testumgebung ist allerdings nicht gegeben. Ebenso ist ein analytischer Nachweis sicherheitskritischer Anforderungen nicht verfügbar. Ein weiterer Nachteil besteht in der fehlenden Unterstützung bei der Implementierung auf verteilten Plattformen.

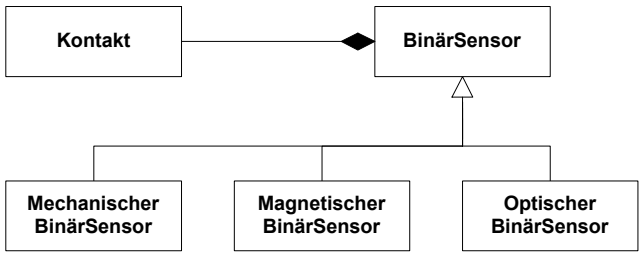
● Unified Modeling Language (UML):

In der Firma Rational Software Corporation haben sich Grady Booch, Jim Rumbaugh und Ivar Jacobson zusammengefunden, um einen Standard für den Bereich der objektorientierten Analyse und Design zu setzen [UML 97], [Oesterreich 97], [Burkhardt 97]. Die Zusammenführung ihrer drei Methoden und die Einbeziehung der Erfahrungen aus der Anwendung bilden hierfür die Grundlage. Die dieser Arbeit zugrundeliegende Version 1.1 ist seit Ende 1997 ein durch die OMG (*Object Management Group*) [OMG 98] akzeptierter Standard für die graphische Notation objektorientierter Software.

Die UML enthält keine Aussage über das Vorgehen der Methode. Als Hinweis kann hier nur der Verweis auf die zugrundeliegenden Methoden [Booch 95], [Rumbaugh 91] und [Jacobson 92] gegeben werden. Innerhalb der UML sind eine Reihe von unterschiedlichen Entwurfsdiagrammen spezifiziert. Folgende Diagramme werden im *Notation Guide* [UML 97] als zentral beschrieben:

- Das Klassendiagramm (*class diagram*) zeigt die logische statische Sicht auf die Struktur eines Systems mit den Beziehungen (*association*) der Klassen untereinander. Die UML mißt den statischen Strukturdiagrammen (*static structure diagrams*) eine besondere Bedeutung zu. Sie geht sehr detailliert auf die graphische Notation von Klassen und deren Beziehungen (z.B. Vererbung und Aggregation, siehe z.B. Abbildung 2.16) ein.

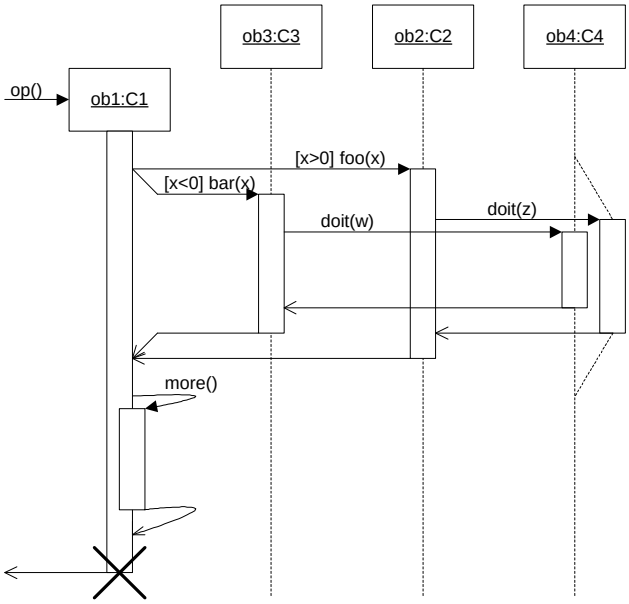
Abbildung 2.16
Klassendiagramm für Binärsensoren



- Ein Anwenderfalldiagramm (*use case diagram*) wird genutzt, um das Verhalten des Systems als Ganzes oder von abgeschlossenen Teilsystemen zu spezifizieren. Es zeigt die Interaktion zwischen externen Akteuren und dem System/Teilsystem. Ein Szenario ist die Instanz eines Anwendungsfalls und zeigt seinen Ablauf. Ein Anwendungsfall besteht normalerweise aus mehreren Szenarien.

- Das Sequenzdiagramm (*sequence diagram*) stellt neben dem Kollaborationsdiagramm die Kommunikation der einzelnen Objekte untereinander dar. Die Abfolge der Events, die Objektkreierung und -terminierung ist ebenfalls modellierbar (vgl Abbildung 2.17). Eine Darstellung des Kontrollflusses ist möglich. Die Sequenzdiagramme wurden mit kleinen Modifikationen aus den *Message Sequence Charts* [UIT-T 92] abgeleitet. Sequenzdiagramme dienen der Identifizierung der Events und ihrer Abfolge. Nach ihnen werden die Zustandsdiagramme der einzelnen Objekte entworfen. Erfolgt der Entwurf der Zustandsgraphen nicht direkt aus den Sequenzdiagrammen, so können sie z.B. in der Simulationsphase als Kontrollreferenz eingesetzt werden.

Abbildung 2.17
Sequenzdiagramm [UML 97]



- Das Zustandsdiagramm (*state diagram*), welches bis auf Kleinigkeiten sich an Harels Statecharts anlehnt, beschreibt das Verhalten eines Objekts der Klasse. Der Zustandsgraph ist der Klasse zugeordnet.

- Kollaborationsdiagramme (*collaboration diagram*) stellen die existierenden Objekte und Links vor Beginn und nach einer Operation dar. Links bewerkstelligen den Nachrichtenfluß zwischen den Objekten. Gezeigt wird, wie verschiedene Objekte bei der Ausführung einer Operation involviert sind. Das Kollaborationsdiagramm ist ein Szenario-Diagramm.

- Aktivitätsdiagramme (*activity diagram*) dienen der abstrakten Darstellung der Implementation einer Operation. Mit Hilfe von Operationen und Suboperationen wird die Gesamtheit der Funktionalität einer Operation dargestellt. Die Beschreibung erfolgt in Form eines Zustandsgraphen, der allerdings nichts mit dem *state diagram* der Klasse zu tun.

- Komponentendiagramme (*component diagram*) repräsentieren die Entwicklungssicht auf ein System. Es zeigt die physikalische Struktur, die Zusammenfassung logischer Einheiten zu Modulen.

- Einsatzdiagramme (*deployment diagram*) stellen die physikalischen Gegebenheiten dar, auf denen das zu implementierende Softwaresystem ausgeführt werden soll. Es besteht aus den Knoten (*nodes*), die die Prozessoren und Geräte (*devices*) des Systems darstellen.

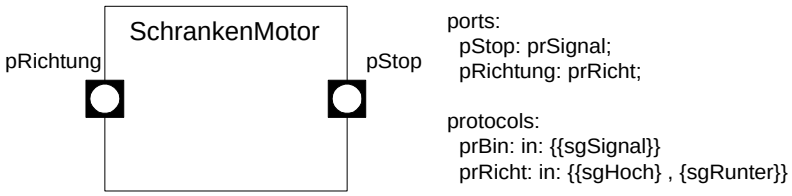
Bewertung: Die statischen Klassendiagramme ermöglichen ein klare Darstellung von Vererbungsbeziehungen. Durch die Vielzahl unterschiedlichster Diagramme in der UML, die teilweise einen sehr abweichenden Abstraktionsgrad besitzen, bleibt eine durchgängige

Modellierung jedoch nur auf wenige Bereiche beschränkt. Ebenso wird die Aussagekraft z.B. der Klassendiagramme durch das Bestreben der UML nach größtmöglicher Allgemeingültigkeit wieder reduziert. Durch eine definierte Beschränkung auf wenige Diagrammtypen mit einem reduzierten Freiheitsgrad können Teile der UML für den Entwurf eingebetteter Systeme gewinnbringend eingesetzt werden.

- **Real-Time Object-Oriented Modeling (ROOM):**

Die in [SeGuWa 94] ausführlich beschriebene objektorientierte Methode wird in [Schulze 97] ausgiebig analysiert und bewertet. Eine knappere Beschreibung findet sich in [BenTom 97]. Die ereignisorientierte ROOM-Methode wurde vor ca. 10 Jahren für die Modellierung von Anlagen und Vorgängen im Bereich der Telekommunikation von *Bell-Northern Research* in Kanada, entwickelt. Ihr liegen ebenfalls hierarchische Automaten, die *ROOMcharts,* zugrunde. Ein ablauffähiges Modell wird als *Aktor* bezeichnet. *Container*-Aktoren können mehreren Aktoren zusammenfassen. Aktoren können miteinander *Nachrichten* über sogenannte *Ports* austauschen. Jeder Port besitzt ein *Protokoll*, welches seine Nachrichten definiert (Signale mit Nutzdaten). Damit zwei Ports sich verbinden lassen, müssen sie die gleichen Protokolle besitzen (vgl. Abbildung 2.18). Der dynamische Aspekt, das *Verhalten* eines Aktors, wird mit Hilfe der *ROOMchart* beschrieben. Somit besitzt ein Aktor eine endliche Anzahl von Zuständen und Zustandsübergängen. Zustandsübergänge werden durch die an den Ports eintreffenden Nachrichten ausgelöst, indem jeder Kante eine sogenannte *Transitionsbedingung* zugewiesen wird. Diese legt fest, welche Nachrichten den Zustandsübergang bewirken. Im Gegensatz zu den Harel-Statecharts kann immer nur ein Zustand auf einer Hierarchieebene aktiv sein.

Abbildung 2.18
Struktur des ROOM-Aktors *SchrankenMotor* mit Port- und Protokolldefinition



Bewertung: Die ROOM-Methode verbindet geeignete komponenten- und datenflußorientierte Darstellungsformen, wie sie bei Ingenieuren üblich sind, mit den zustandsorientierten Statecharts. An- und abgehende Nachrichten werden durch die sogenannten Ports veranschaulicht. Allerdings exisitiert, wie bei der OMT, keine deutliche Trennung zwischen dem Modell der Steuerung und dem Modell der Umgebung. Dieser Mangel verhindert, daß sich der Systemvalidierung eine automatische Implementierung anschließen läßt.

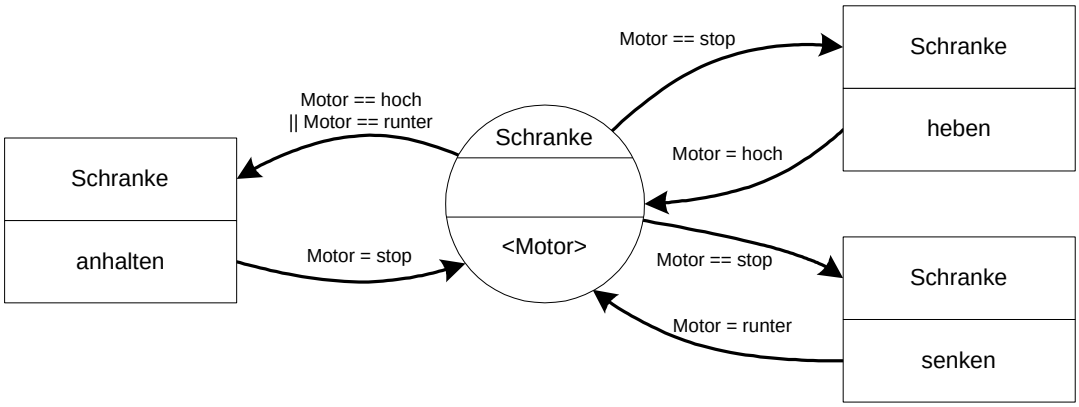
- **Objekt-Prozeß-Modell (OPM):**

Das von Burkhardt 1994 an der Technischen Universität Ilmenau entwickelte objektorientierte Objekt-Prozeß-Modell (OPM) ist eine graphischen Beschreibungssprache, die die spezielle

Berücksichtigung dynamischer Modellierungsaspekte zum Ziel hat [Burkhardt 94]. Das zugehörige 10-stufige Vorgehensmodell behandelt auf den ersten fünf Stufen die Erstellung von Objekt-Prozeß-Diagrammen (OPD) und auf den letzen fünf Stufen weitere Modelldiagramme (sämtliche aus der UML), die die statische Modellsicht repräsentieren.

Das OPM wurde zwar nicht speziell für verteilte Echtzeitsysteme entworfen, enthält aber dennoch dafür nutzbare Eigenschaften. Der definierte Modellierungsprozeß geht vom Kernelement des dynamischen Aspekts aus – dem Prozeß. Von diesen Prozessen aus, welche als Rechtecke dargestellt werden, werden Objekte definiert, die als Kreise dargestellt werden. Die Objekte stellen über ihre aktuelle Attributbelegung Vor- und Nachbedingungen für die Aktivierung der beteiligten Prozesse dar. Im Sinne der Objekt-Technologie ist ein Prozeß die Instanz einer Methode (z.B. *heben*), die einer Klasse (z.B. *Schranke*) zugeordnet wird (vgl. Abbildung 2.19). Ebenso sind die Objekte eines OPD Klassen zuzuordnen.

Abbildung 2.19
OPD für die Ansteuerung des Schrankenmotors



Das OPM bietet die Möglichkeit zur statischen- und dynamischen Modellüberprüfung. Die statische Überprüfung wird auf Methoden der Petri-Netz-Analyse zurückgeführt (z.B. Test auf Konfliktfreiheit). Zur dynamischen Überprüfung wird in den aus den OPD generierten C++-Code zusätzlicher Prüfcode integriert, welcher zur Laufzeit überwacht, ob die durch das OPD gesetzten Vor- bzw. Nachbedingungen beim Methodenaufruf eingehalten wurden. In [Burkhardt 97] wird ausführlich dargestellt, wie sich das OPM in das Umfeld der UML einordnet. Burkhardt zeigt die Zweckmäßigkeit für die Verwendung der statischen Modellierungsmittel aus der UML auf und begründet die Verwendung der Objekt-Prozeß-Diagramme zur Modellierung der dynamischen Aspekte. In [Mühlphort 98] wurden die OPD konsequent erweitert.

Bewertung: Das OPM verwendet konsequent die UML für die Modellierung statischer Modellaspekte. Die OPD bieten eine anschauliche Möglichkeit, notwendige Vor- und Nachbedingungen (Constraints) für einen Methodenaufruf graphisch zu notieren. Die Aktivierung einer Methode, im Sinne der Statecharts, kann mit diesem Formalismus allerdings nicht ausgedrückt werden.

2.5 Schlußfolgerungen und Ziele

2.5.1 Eigenschaften einer durchgängigen Entwurfsmethodik

Die im Rahmen der vorliegenden Arbeit zu entwickelnde Entwurfsmethode für eingebettete verteilte Echtzeitsysteme soll im besonderen Maße die Durchgängigkeit – vom Entwurf über die Validierung bis zur Implementierung – betonen. Aus den Ausführungen des zweiten Kapitels wurden hierfür eine Reihe von Eigenschaften abgeleitet. Die geeignete Integration dieser – im folgenden dargestellten – Eigenschaften gewährleistet schließlich die Durchgängigkeit innerhalb der betrachten Anwendungsdomäne.

- **Allgemeine Eigenschaften:**

  - *Anschaulichkeit der Entwürfe*. Im Sinne der Anschaulichkeit sollte eine Entwurfsmethode möglichst graphisch orientierte Beschreibungsmittel einsetzen. Die hohe Akzeptanz der Statecharts (nach Harel) unterstreicht dabei die starke Bereitschaft der Entwickler, die graphische Software-Entwicklung auf Basis erweiterter Zustandsmaschinen durchzuführen.

  - *Objektorientierung für einen durchgängigen Entwurf*. Es hat sich in der Praxis gezeigt, daß das objektorientierte Paradigma ein hervorragendes Mittel zur Strukturierung und Wiederverwendung von komplexen Entwürfen ist. Die in der Analysephase erfolgte Spezifikation des *Was* sollte immer in die Spezifikation des *Wie* mit einfließen. Das Prinzip der Vererbung eignet sich ideal dazu, das abstrakte *Was* in ein konkretes *Wie* zu überführen.

  - *Bezug zu akzeptierten Standards*. Die *Unified Modeling Language* (UML) stellt einen akzeptierten Standard zur graphischen Notation von objektorientierter Software dar. Eine Entwurfsmethodik, die eine möglichst hohe Akzeptanz erreichen will, kann diesen Standard nicht vollständig ignorieren. Dabei muß allerdings nicht jede in der UML definierte Notationsform auch Verwendung finden. Ebenso können auch zusätzliche Diagrammarten, die für eine spezielle Anwendungsdomäne neu entwickelt wurden, eingesetzt werden.

  - *Komponenten- und datenflußorientierte Darstellung*. Die ROOM-Methode greift den naheliegenden Gedanken auf, Nachrichten, die ein Objekt erreichen bzw. verlassen, über sogenannte Ports zu leiten. Dies ist eine sehr anschauliche Möglichkeit die Kommunikation zwischen den Objekten darzustellen. Soll eine Entwurfsmethode in der traditionell durch Elektroingenieure geprägten Embedded-Domäne seine Anwender finden, so ist eine komponenten- und datenflußorientierte Darstellung der Objekte und ihrer dynamischen Beziehungen sehr vorteilhaft. Sie ist stark an die Darstellungsform digitaler Schaltungen angelehnt, in der die integrierten Schaltkreise und die sie verbindenden Signalleitungen die bestimmenden Elemente sind.

- **Eigenschaften für Entwurf von Echtzeitsystemen:**

  - *Garantie zeitlicher Schranken und logischer Bedingungen*. Dies sind wohl die wichtigsten Eigenschaften, die ein Echtzeitsystem besitzen muß. Die Reaktionen der Steuerung auf

Ereignisse, welche durch die Umgebung ausgelöst werden, müssen innerhalb festgelegter zeitlicher Schranken und in einer definierten Reihenfolge ablaufen. Diese zeitlichen und logischen Anforderungen sollten als temporal-logische Ausdrücke, die sich formal gegenüber der funktionalen Spezifikation verifiziert lassen, formalisiert werden können. Damit diese Ausdrücke (die Constraints) sich bereits in einer frühen Entwurfsphase notieren lassen, dürfen sie sich nicht auf interne Details der Objekte beziehen.

- *Validierung der funktionalen Spezifikation*. Die Validierung der Spezifikation einer Echtzeitsteuerung erfolgt über die Kopplung mit einer Spezifikation der Umgebung. Dabei ist es von Vorteil für den Anwender, wenn die Modellierung dieser Umgebungs-spezifikation nach der gleichen Methodik erfolgt. Für die Validierung und die Verifikation der Constraints sollte die Spezifikation der Steuerung zusammen mit der Umgebung in eine formale, ausführbare Form (z.B. Petri-Netze) überführt werden können.

- *Validierung mit realen Zeiten*. Die Validierung ist näher an der Realität der späteren Implementierung, wenn reale Zeiten für die Ausführung von Software-Aktivitäten und für die Kommunikation zwischen verteilten Rechnerknoten eingesetzt werden können; spezielle Parameter der Zielplattform müssen sich hierzu in die Validierung der Spezifikation einbeziehen lassen.

● Eigenschaften für die Implementierung verteilter eingebetteter Systeme:

Da verteilte eingebettete Systeme oft heterogen und im hohen Maße proprietär für die spezielle Anwendung konzipiert werden, muß eine Entwurfsmethode für die Phase der Implementierung im besonderen den Hardware-Aspekt berücksichtigen. Die Hardware-Plattform wird bei eingebetteten Systemen aus Kostengründen häufig stark an der speziellen Anwendung ausgerichtet. Um diese projektspezifischen Zielarchitekturen von der höheren Entwurfsebene abzuschirmen, muß ein Konzept zur Spezifikation und Abstraktion der Hardware-Funktionalitäten in die Entwurfsmethode integriert sein. Konzepte, wie sie durch die virtuelle Java-Maschine [Venners 97] verfolgt werden, können dabei ausgeschlossen werden, da sie versuchen, die Zielplattformen an der Entwurfsmethode (bzw. Programmierparadigma) auszurichten und nicht umgekehrt.

Ist ein System vollständig entworfen, so daß alle Funktionsdetails im Modell realisiert und validiert wurden, so kann die Implementierung auf einer verteilten Hardware-Plattform erfolgen. Für diese Implementierung sollte eine Entwurfsmethode folgende Eigenschaften besitzen:

- *Trennung von logischer und physikalischer Realisierung*. Die Entwurfsmethodik sollte unterhalb der objektorientierten Entwurfsebene eine zusätzliche logische Realisierungsebene bereithalten, um eine Implementierung auf unterschiedlichen Plattformen mit unterschiedlichen Echtzeitbetriebssystemen und unterschiedlichen Kommunikationsnetzwerken zu ermöglichen. Als ein geeignetes Mittel zur Umsetzung von Nebenläufigkeit und Verteilung haben sich Systeme kommunizierender Prozesse herausgestellt. Die Kommunikation und Synchronisation der nebenläufig und gegebenenfalls auch verteilt arbeitenden Prozesse durch das Botschaftenkonzept hat sich dabei als zweckmäßigster Ansatz etabliert. Durch diese

logische Realisierungsschicht wird neben den verschiedenen Knotenarchitekturen auch das unterlegte Kommunikationsnetzwerk (z.B. PROFIBUS oder CAN) gekapselt. Es existiert auf dieser Ebene für den Entwurf kein Unterschied zwischen lokal oder verteilt operierenden Prozessen. Die Abstraktion dieser Realisierungsdetails ermöglicht eine leichtere Portier- und Wiederverwendbarkeit.

- *Portier*- und *Verteilbarkeit*. Die Ansteuerung der Aktoren und Sensoren sollte über eine zusätzliche Plattformabstraktionsebene erfolgen. Diese Ebene kapselt die spezifischen Ansteuerungsdetails der beteiligten Aktoren und Sensoren für unterschiedliche Rechnerknoten und stellt ihre Funktionalität in abstrahierter Form zur Verfügung. Durch diese Trennung von Funktionalität, welche auf der höheren Entwurfsebene sichtbar ist, und den Ansteuerungsdetails, die für die Implementierung benötigt wird, bleiben die Entwürfe portierbar. Diese Portierbarkeit ermöglicht es, daß eine freie Zuordnung von Funktionalitäten auf die verteilte Plattform erfolgen kann.

- *Maximale Hardware-Nähe*. Trotz der Kapselung der Hardware-Details (*wrapping*) sollte die eigentliche Ansteuerung so direkt und effektiv wie möglich erfolgen. Die Methodenimplementation der Klassen der Abstraktionsebene, die direkten Hardware-Zugriff haben, erfolgt hierzu in einer der Plattform originären Codierungsform. Dies kann z.B. auch Assemblercode einschließen.

2.5.2 Integration der Eigenschaften durch die Objektnetz-Methodik

Die im folgenden Kapitel vorgestellte Objektnetz-Methodik wurde unter dem Aspekt konzipiert, die zuvor formulierten Eigenschaften in einer durchgehenden Form innerhalb einer Entwurfsmethodik zusammenzuführen. Die Objektnetze besitzen die allgemeinen Eigenschaften aus Sicht des Entwicklers, die für Echtzeitsysteme notwendigen Eigenschaften sowie die Eigenschaften, die für die Impementierung verteilter eingebetter Systeme notwendig sind.

3. Kapitel

Der Entwurf mit Objektnetzen

Im folgenden Kapitel wird übersichtlich und kompakt eine neue graphische Entwurfsmethodik für verteilte eingebettete Echtzeitsysteme vorgestellt. Diese neue Methodik nennt sich *Concurrent Object Nets* oder kurz *Objektnetze* und betont somit ihre objektorientierten Konzepte und ihre Wurzeln in der Petri-Netz-Theorie. Die Methodik der Objektnetze bildet den zentralen Punkt der vorliegenden Arbeit. Die folgenden Abschnitte sollen den Leser einerseits von der Zweckmäßigkeit der Objektnetze für den Entwurf von verteilten Echtzeitsystem überzeugen, und andererseits ihn in die Lage versetzen, die neue Entwurfsmethode einzusetzen.

Einleitend erfolgt vereinfachend die Einordnung der Objektnetz-Methodik in das Phasenmodell. Verschiedene Sichten auf das zu entwerfende System werden vorgestellt. Danach wird der dreistufige Entwurfsprozeß für Objektnetze eingeführt. Es werden die Prinzipen der einzelnen Stufen anhand eines überschaubaren Beispiels diskutiert. Abstrakte Objektnetz-Klassen bilden den Einstieg für die erste Stufe. Diese abstrakten Klassen werden in der zweiten Stufe durch Vererbung schrittweise konkretisiert. Auf der dritten Stufe werden die vollständig konkretisierten und noch plattformunabhängigen Objektnetz-Modelle durch das Hinzufügen weiterer Informationen für die automatische Implementierung auf einer verteilten Plattform vorbereitet.

Nach der Erläuterung des Entwurfsprozesses werden schließlich schrittweise die einzelnen Modellierungselemente der Objektnetz-Methodik und ihres Frameworks zur Plattformabstraktion eingeführt. Zur Darstellung ihrer Zusammenhänge wird ein Meta-Klassen-Modell definiert. Dieses nach den Regeln der *Unified Modeling Language* [UML 97] erstellte Klassen-Modell ist eine leicht verständliche formalisierte grafische Notation, die neben dem Zwecke der Erläuterung die Basis für die Tool-Erstellung bildet [Kahnert 98].

3.1 Objektnetze

3.1.1 Einordnung

Die Schlußfolgerungen des zweiten Kapitels lieferten die Anregungen für eine neue objektorientierte Entwurfsmethode. Die im folgenden vorgestellten Objektnetze (*Concurrent Object Nets*) sind ein neuartiger Ansatz für den Entwurf, die Validierung und die Implementierung von verteilten eingebetteten Echtzeitsystemen. Objektnetze verbinden geeignet verschiedene Entwurfskonzepte, die aus unterschiedlichen Bereichen stammen, zu einer leistungsfähigen Entwurfsmethode. Ziel dieser neuen Methode ist es, visuelle und objektorientierte Entwurfsverfahren für die in der Regel sehr hardwarenahe arbeitenden Entwickler eingebetteter Echtzeitsysteme bereitzustellen. Objektnetze lösen den bestehenden Widerspruch zwischen der abstrahierten Herangehensweise innerhalb des objektorientierten Paradigmas und der von den Entwicklern eingebetteter Systeme geforderten effektiven Hardware-Ansteuerung [NüDäFe 98]. Speziell für den Entwurf von sicherheitskritischen Systemen, welche häufig Echtzeitbedingungen genügen müssen, liefern Objektnetze den Zugang zur Verifikation sicherheitsrelevanter Eigenschaften. Die Bereitstellung einer formalen Verhaltensbeschreibung auf Basis von Petri-Netzen liefert die Grundlage für die Validierung.

Abbildung 3.1
Einordnung der Objektnetz-Methodik in das Phasenmodell

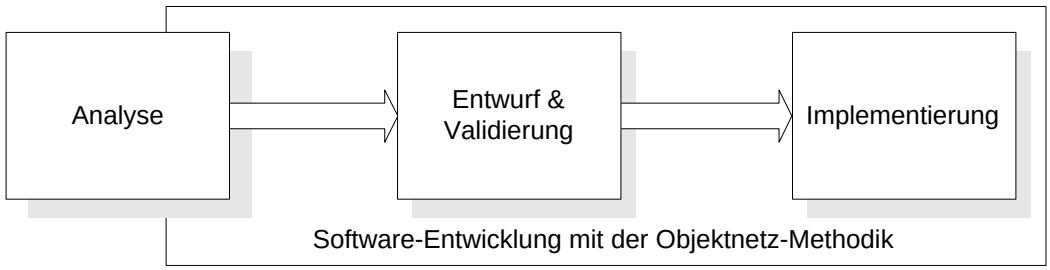


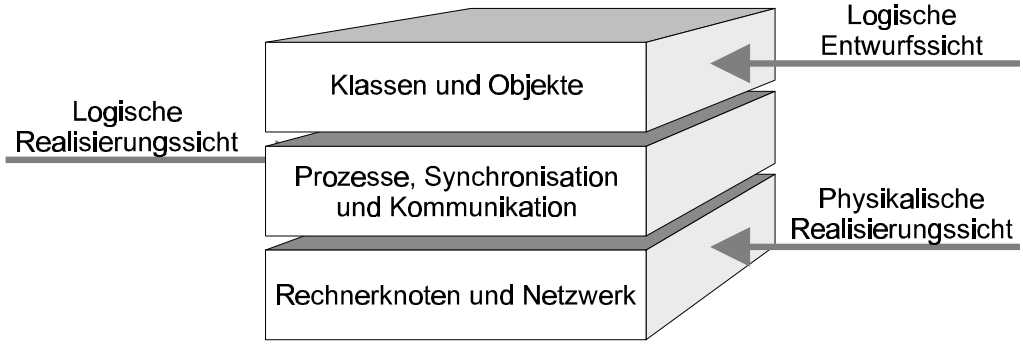
Abbildung 3.1 zeigt den Schwerpunkt der Software-Entwicklung mit Objektnetzen. Die Analysephase, in der die Spezifikation des *Was* erfolgt, wird nur teilweise durch die Objektnetz-Methodik unterstützt. Es wurde bewußt darauf verzichtet ein aufwendiges Regelwerk zu formulieren, welches dem Designer vorschreibt, wie die realen Objekte der Welt auf die Software-Objekte abgebildet werden (*„how to find the objects"*).

3.1.2 Verschiedene Systemsichten

Es ist unmöglich, Echtzeitsysteme, die einen immer höheren Komplexitätsgrad erreichen, unabhängig davon, ob sie verteilt oder konzentriert realisiert werden, nur aus einem Blickwinkel zu erfassen (vgl. [Booch 91, Seite 155]). Erst die verschiedenen Betrachterstandpunkte in ihrer Gesamtheit mit ihren unterschiedlichen Sichtweisen ermöglichen es, ein vollständiges Modell für ein komplexes Systems zu erstellen.

Folglich muß eine Entwurfsmethodik für verteilte eingebettete Echtzeitsysteme eine Reihe von verschiedenen Sichtweisen bereithalten (vgl. Abbildung 3.2). Zum einen muß sie die logische Sichtweise des Auftraggebers und des Designers auf das System unterstützen, deren Anschauung in der Regel noch völlig unbeeinflußt von späteren Realisierungsdetails ist. Zum anderen sollte sie die Sicht auf die physikalische Realisierung nicht vernachlässigen, da es andernfalls nahezu unmöglich ist, die erstellten Spezifikationen ressourcenoptimiert zu implementieren. Die optimale Ausnutzung der meist vorgegebenen Hardware-Plattform ist ein entscheidendes Merkmal für eingebettete Systeme.

Abbildung 3.2
Drei Modellebenen und die Sichten der Objektnetz-Methodik



Die logische Realisierungsebene verbindet die beiden sich widersprechenden Sichten. Sie stellt das Bindeglied zwischen der durch das objektorientierte Paradigma geprägten logischen Entwurfssicht, und der durch die Struktur der Zielplattform geprägten physikalischen Realisierung dar. Abbildung 3.2 zeigt die drei sich ergänzenden Modellebenen mit den unterschiedlichen Sichten der Objektnetz-Methodik.

● Die logische Entwurfssicht:

Die logische Sicht des Entwerfers auf das System erfolgt innerhalb der Objektnetz-Methodik natürlich objektorientiert. In Kapitel 2 wurden bereits die Vorteile des objektorientierten Paradigmas herausgestellt und betont, daß sie sich generell auch auf den Entwurf von Echtzeitsystemen übertragen lassen.

Die Objektnetz-Methodik trennt, wie es bereits in [Burkhardt 94] vorgeschlagen wurde, zwischen den statischen und den dynamischen Aspekten des Modells. Die statischen Modellaspekte werden durch die Klassen und ihre Vererbungsbeziehungen ausgedrückt. Die Objektnetz-Methodik definiert hierzu drei getrennte Klassenhierarchien: Datenklassen, Objektnetz-Klassen und Aktor/Sensor-Klassen. Die dynamischen Aspekte werden durch die Nachrichten, die zwischen den Objekten ausgetauscht werden (die Interobjektdynamik), und die Art und Weise, wie die Objekte intern auf diese Nachrichten reagieren (die Intraobjektdynamik), widergespiegelt. Der Schwerpunkt liegt dabei bei der Modellierung der dynamischen Aspekte

innerhalb der Objektnetz-Klassen. Tabelle 3.1 faßt die statischen und dynamischen Aspekte der Objektnetz-Methodik zusammen.

Tabelle 3.1
Die statischen und dynamischen Aspekte der Objektnetz-Methodik

Statische Aspekte	Dynamische Aspekte
Datenklassen	Interobjektynamik
Objektnetz-Klassen	☐ Nachrichten
☐ Hierarchische Objektnetze	☐ Nachrichtenverbindungen
☐ Elementare Objetnetze	Intraobjektdynamik
Aktor/Sensor-Klassen	☐ Erweiterte Zustandsmaschine
Vererbung	☐ Wartezeiten
☐ Konkretisierung von Klassen	Zeitliche und logische Constraints

Für die graphische Spezifikation statischer Aspekte, z.B. die Vererbungsbeziehungen der Klassen, greift die Objektnetz-Methodik auf die Standards der UML [Fowler 97], [Oesterreich 97] zurück. Dies ermöglicht es, die statischen Aspekte stark verkürzt zu behandeln. Der Schwerpunkt dieses Kapitels gehört den dynamischen Aspekten.

- **Die logische Realisierungssicht und ihr Basismodell:**

Als Bindeglied zwischen der logischen Entwurfsebene und der physikalischen Realisierungsebene fungiert die logische Realisierungsebene. Sie erfüllt die Aufgabe eines virtuellen Echtzeit-Betriebssystems (*Real-Time Operating System* – RTOS). Durch diese Zwischenebene kann das Verhalten der logischen Entwurfsebene von den Spezifika verschiedener Zielbetriebssysteme abgekoppelt werden. Der Entwickler muß sich nicht mit den Details des jeweiligen Betriebssystems, z.B. der Prozeßverwaltung, befassen.

Das Basismodell der Objektnetze beruht aus logischer Realisierungssicht auf dem Modell der asynchron kommunizierenden erweiterten endlichen Zustandsautomaten [Hogrefe 89]. Diese Zustandsautomaten wurden um lokale Variablen, die Attribute, und um Timer-Funktionen (Wartezeiten) erweitert. Die Kommunikation und somit auch die Synchronisation der Automaten untereinander erfolgen asynchron über feste Kommunikationskanäle, die im folgenden Nachrichtenverbindungen genannt werden. Synchrone Kommunikation wird auf asynchrone Kommunikation zurückgeführt. Im Gegensatz zu den klassischen Automatensystemen [Krapp 88] befindet sich am Endpunkt der Nachrichtenverbindung ein FIFO-Empfangspuffer. Dieser Puffer ist einem oder mehreren Zustandsübergängen in den Automaten zugeordnet. Die Zustandswechsel der Automaten werden entweder durch den Empfang spezieller Nachrichten oder durch Ablauf eines internen Timers ausgelöst.

Die logische Entwurfsebene stellt die konzeptionelle Weiterentwicklung der logischen Realisierungsebene um objektorientierte Entwurfsprinzipien dar.

- **Die physikalische Realisierungssicht:**

Die direkte physikalische Sicht auf die verteilte Zielplattform sollte in der Regel einem Hardware-Spezialisten überlassen bleiben. Dieser realisiert einmalig die Kapselung der jeweiligen Plattformspezifika. Die Objektnetz-Methodik definiert hierfür ein Framework [NüDäFe 98] zur Plattformabstraktion (PAF – *Platform Abstraction Framework*). Frameworks sind domänenspezifische, typischerweise objektorientierte, Codebiliotheken mit zugehörigen Entwurfsrichtlinien [Pree 95], [Pree 96], [PhiIva 97], die die vereinfachte Entwicklung neuer Applikationen mit ähnlichen Charakteristika ermöglichen. Das Plattformabstraktionsframework der Objektnetz-Methodik ermöglicht einem mit den Hardware-Details vertrauten Entwickler in kurzer Zeit spezielle Klassen, die den Ansteuerungscode für die Aktoren und Sensoren der beteiligten Plattformen kapseln, zu erstellen. Durch diese plattformabhängige Ebene wird die Plattformunabhängigkeit von Objektnetz-Spezifikationen gewährleistet.

3.2 Der Software-Entwicklungsprozeß mit Objektnetzen

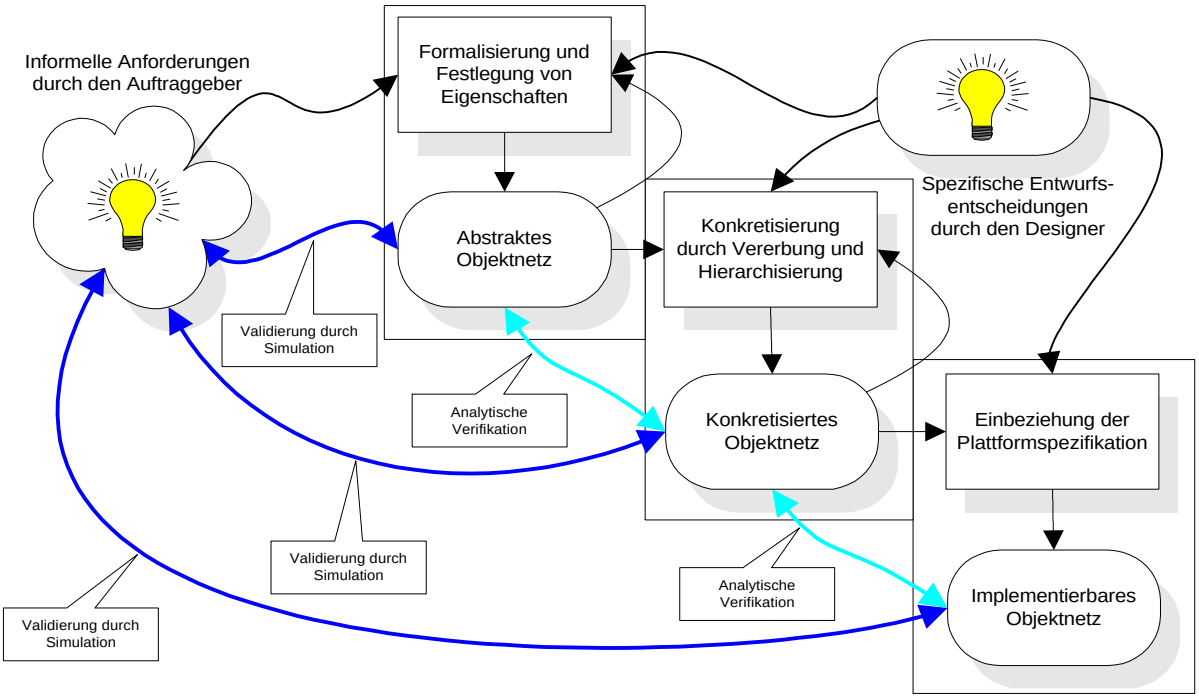
Die Entwurfsmethodik der Objektnetze hält neben einer speziellen Entwurfsnotation einen definierten Entwicklungsprozeß mit drei Hauptstufen bereit (vgl. Abbildung 3.3). Der Forderung von Booch nach einer einheitliche Notationsform für Analyse- und Entwurfsphase wird dabei Rechnung getragen. Der dreistufige Software-Entwicklungsprozeß transformiert die informell vorliegenden Anforderungen des Auftraggebers schrittweise bis zu einer implementierbaren Spezifikation. Auf jeder Stufe dieses Prozesses erweitert der Software-Designer die Objektnetz-Spezifikation durch seine spezifischen Entwurfsentscheidungen. Spezielle analytische Verifikationsverfahren, welche in Kapitel 6 beschrieben werden, garantieren dabei die Einhaltung der auf der ersten Stufe (während der Anforderungsspezifikation) des Entwurfsprozesses festgelegten und formalisierten wichtigen Solleigenschaften.

Da ein erstes abstraktes Objektnetz noch Spezifikationsfreiräume enthalten kann, muß es so lange konkretisiert werden, bis der Auftraggeber sämtliche Anforderungen umgesetzt sieht. Um Interpretationsfehler durch den Designer auszuschließen, kann auf jeder Stufe eine Validierung der Objektnetz-Spezifikation gegenüber den informellen Anforderungen des Kunden durchgeführt werden. Diese Validierung erfolgt typischerweise durch Simulation.

Die erste Stufe des Entwurfsprozesses wird noch in enger Zusammenarbeit mit dem Kunden durchlaufen. Durch die Zuarbeit des Designers entsteht dabei das abstrakte Objektnetz als formale Anforderungsspezifikation. Das abstrakte Objektnetz bildet einen ersten (eingeschränkt) simulierbaren Prototyp. Die zweite Stufe bildet die sukzessive Konkretisierung des abstrakten Objektnetzes unter Verwendung sämtlicher Entwurfsmittel der Objektnetz-Methodik. Am Ende dieser zweiten Stufe steht die Software-Architekturspezifikation in Form eines vollständig konkretisierten und plattformunabhängigen Objektnetzes. Die letzte Prozeßstufe bezieht

schließlich die Plattformspezifikation ein; sie ermöglicht eine automatische Implementierung auf einer speziellen verteilten Zielplattform.

Abbildung 3.3
Der dreistufige Software-Entwicklungsprozeß mit Objektnetzen



Auf den ersten Blick handelt es sich bei der Objektnetz-Methodik um einen *Top-Down*-Entwurf, da vom Allgemeinen und Übergeordneten ausgehend, schrittweise erweitert und verfeinert wird. Der Designer bedient sich dabei der Verfahren der Vererbung und Hierarchisierung. Existieren jedoch bereits konkretisierte Objektnetz-Klassen aus älteren Projekten mit validierten Eigenschaften, so können diese im Rahmen eines *Bottom-Up*-Vorgehens wiederverwendet werden. Dieses Vorgehen ermöglicht es, die ersten beiden Stufen zu einem Schritt zusammenzuziehen. Ebenso kann die Plattformspezifikation eines zurückliegenden Projektes, welches auf der gleichen Ziel-Hardware implementiert wurde, wiederverwendet werden.

3.2.1 Prinzip der abstrakten Objektnetz-Spezifikation

Ausgehend von den informellen Anforderungen des Auftraggebers wird am Beginn des Objektnetz-Entwurfsprozesses eine abstrakte Spezifikationshülle erstellt. Dieses erste Objektnetz enthält noch kaum Realisierungsdetails, sollte aber bereits alle sicherheitskritischen Anforderungen des Kunden aufnehmen. Diese erste Stufe ist der bei weitem kritischste Schritt im gesamten Entwurfsablauf. Kann hier der Software-Experte nicht bereits sämtliche wichtigen Eigenschaften des Systems in das Objektnetz integrieren, so ist zu befürchten, daß dies auf einer späteren Stufe nur noch mit sehr hohem Aufwand gelingen kann.

Das auf Seite 14 eingeführte Bahnschrankenbeispiel verdeutlicht das Prinzip einer abstrakten Objektnetz-Spezifikation. Um die Übersichtlichkeit zu erhalten, wird auf die Modellierung des Fehlerverhaltens verzichtet. Neben den bereits gezeigten Sensoren (*SensorLinks*, *SensorRechts*), die die Ein- und Ausfahrt eines Zuges feststellen, existieren zwei weitere Sensoren (*SensorOben*, *SensorUnten*), die den Zustand der Schranke signalisieren. Das Öffnen und Schließen der Schranke erfolgt über die Ansteuerung eines Aktors (*SchrankenMotor*).

Abbildung 3.4
Die erste (abstrakte) Stufe der Objektnetz-Spezifikation

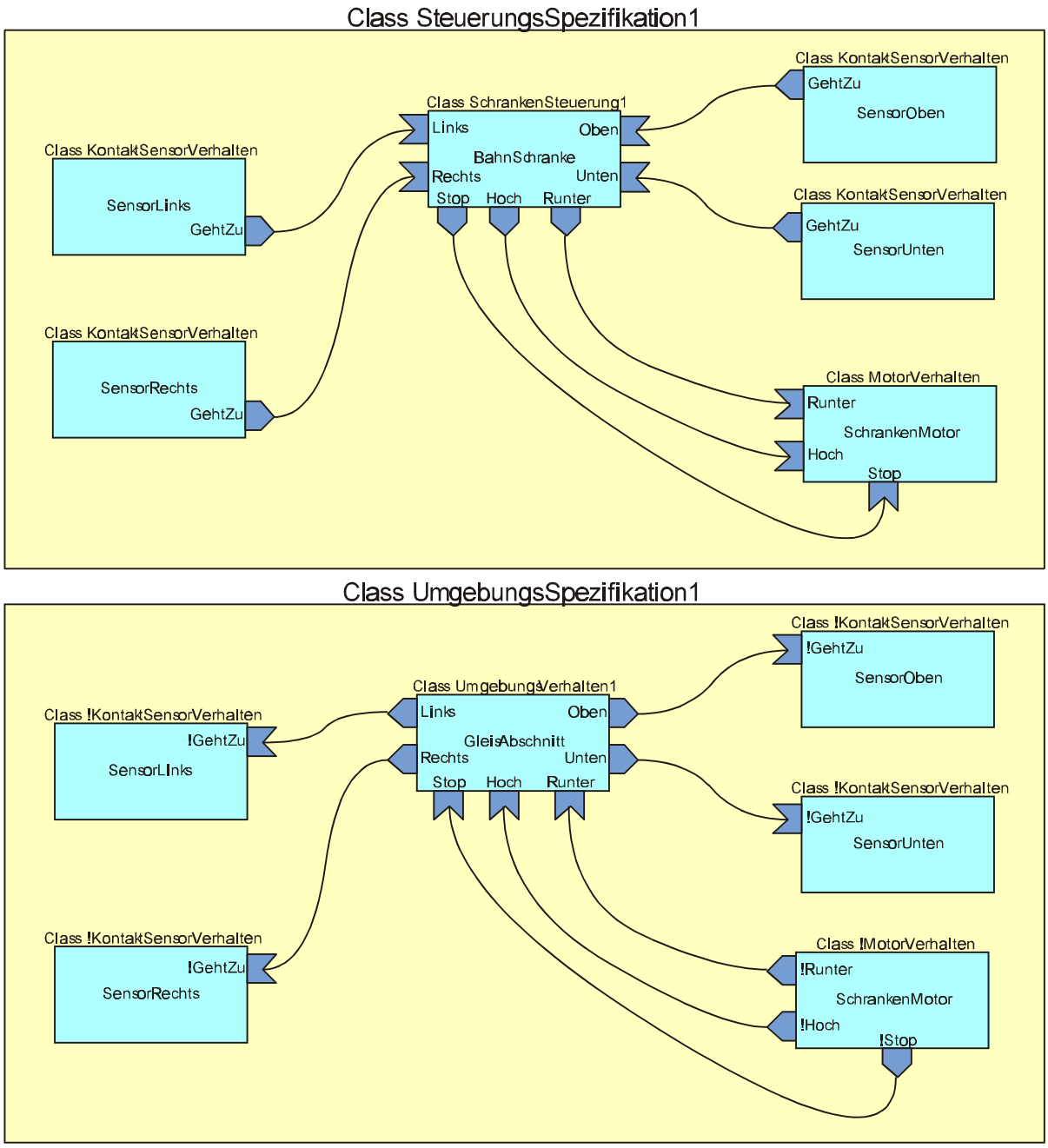


Abbildung 3.4 zeigt die erste (abstrakte) Stufe der Gesamtspezifikation. Sie zerfällt grundsätzlich immer in zwei getrennte Teile. Der erste Teil modelliert die eigentliche Steuerung

(bzw. die Steuerungssoftware), die durch die Instanz einer schnittstellenlose Objektnetz-Klasse (*SteuerungsSpezifikation1*) modelliert wird. Der zweite Teil, der das Verhalten der Umgebung modelliert, wird ebenfalls durch eine Instanz einer schnittstellenlosen Klasse (*Umgebungs-Spezifikation1*) verkörpert. Die Kopplung zwischen Steuerung und Umgebung erfolgt über Aktor/Sensor-Objekte (ASO). Diese in Abbildung 3.4 nicht sichtbaren AS-Objekte werden von den Instanzen spezieller Klassen (*KontaktSensorVerhalten*, *MotorVerhalten*) referenziert; sie modellieren das dynamische Verhalten der Aktoren bzw. Sensoren. AS-Objekte, die von der Steuerung als Aktoren angesprochen werden, werden von der Umgebung als komplementäre Sensoren betrachtet. Für Sensoren der Steuerung verhält es sich entsprechend umgekehrt. Die Zuordnung von Aktor zu komplementären Sensor erfolgt über die Instanznamen (*SensorLinks*, *SensorRechts* usw.).

Die Klasse *SteuerungsSpezifikation1* enthält eine Instanz (*BahnSchranke*) der abstrakten Objektnetz-Klasse *SchrankenSteuerung1*. Diese Klasse besitzt außer ihrer Schnittstellen-definition (die Ports: *Links*, *Rechts*, *Stop*, *Hoch*, *Runter*, *Oben*, *Unten*) keine interne Realisierung. Das gleiche gilt für die Klasse *UmgebungsVerhalten1*. Erst der Schritt der Konkretisierung durch Vererbung kann eine Unterklasse von *SteuerungsSpezifikation1* liefern, die diese Instanz einer abstrakten Klasse konkretisiert (überschreibt).

Vervollständigt wird die abstrakte Klasse *SchrankenSteuerung1* durch die Angabe des für den Anwender wichtigen Sollverhaltens. Dieses Sollverhalten des Echtzeitsystems, welches bereits auf Seite 14 informell notiert wurde, kann innerhalb der Objektnetz-Methodik durch sogenannte Constraints formalisiert notiert werden. Auf der Ebene einer abstrakten Objektnetz-Klasse dienen Constraints der Festlegung gewünschter Ereignisse und deren Abfolge. Während der Phase der Vererbung liefern sie den Entwurfsrahmen, innerhalb dessen die einzelnen Klassen zu konkretisieren sind. Constraints werden innerhalb der Validierung auf ihre Einhalt-ung überprüft. Diese kann durch analytische Methoden oder während der Simulation erfolgen.

Die Objektnetz-Methodik definiert einen textuellen Formalismus (ONCL – *Object Net Contraint Language*) in dem durch einfache Englische Sätze solche Zwänge formal notiert werden können. Die Klasse *SchrankenSteuerung1* besitzt vier solcher Constraints.

Abbildung 3.5
Zwei *Constraints* der Klasse *SchrankenSteuerung1*

```
Constraint Sicherheit1
  { receive Unten between 0 t1 after Links && Oben occurred }
Constraint Sicherheit2
  { receive Unten between 0 t1 after Rechts && Oben occurred }
```

Constraint *Sicherheit1* z.B. formuliert den Zwang, daß spätestens nach *t1* Zeiteinheiten nachdem an *Links* eine Nachricht eintraf, in Verbindung mit dem Eintreffen einer Nachricht an *Oben*, eine Nachricht an *Unten* eintreffen muß. Die vollständige Syntax der ONCL wird in Kapitel 3.3 im Zusammenhang mit der detaillierten Beschreibung abstrakter Objektnetz-Klassen eingeführt. In Kapitel 5.3 wird demonstriert, wie sich das Verhalten der Constraints durch Petri-

Netze beschreiben läßt. Das vollständige Bahnschranken-Beispiel – mit zwei weiteren Constraints – findet sich im Anhang der Arbeit.

3.2.2 Prinzip der Konkretisierung durch Vererbung

Die Prozeßstufe der Konkretisierung überführt die abstrakte Spezifikation der Steuerung und der Umgebung in einem oder mehreren Entwurfsschritten in ein vollständig spezifiziertes Objektnetz-Modell. Dieses Modell kann bereits mit allen Funktionsdetails simuliert werden. Um das Modell auch auf einer speziellen Zielplattform zu implementieren, muß eine zusätzliche Plattformspezifikation einbezogen werden. Bis zu diesem Schritt bleibt das Objektnetz jedoch vollständig plattformunabhängig.

3.2.2.1 Fundamentale Vererbungsregeln

Die Objektnetz-Methodik definiert für den Prozeß der Konkretisierung ausschließlich den Mechanismus der Vererbung. Die hierfür definierten Regeln legen fest, wie eine neu zu erstellende Objektnetz-Klasse aussehen muß, damit sie eine gültige Unterklasse einer bereits bestehenden Klasse, und somit eine zulässige Konkretisierung ist. Angelehnt an die Definitionen von Basten und van der Aalst [AaBa 97] wird für Objektnetze die Gültigkeit einer Konkretisierung durch Vererbung über das beobachtbare Verhalten und zusätzlich über die Constraints der Klasse festgelegt.

Das beobachtbare Verhalten einer Klasse ist geprägt durch Nachrichten, die über ihre Ports abgegeben bzw. aufgenommen werden. Constraints schränken bereits für abstrakte Klassen diese beobachtbare Kommunikation mit dem Umfeld ein. Constraints der Oberklasse gelten unverändert in den Unterklassen weiter. Auf Basis dieser Aussagen werden die Regeln zur Vererbung von Objektnetz-Klassen festgelegt. Sie sorgen primär dafür, daß Unterklassen grundsätzlich immer in dem Umfeld eingesetzt werden können, in dem auch ihre Oberklasse eingesetzt wurde (➪ statischer Polymorphismus).

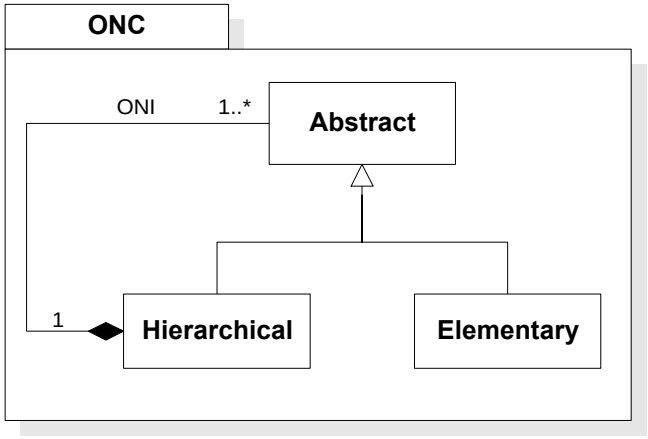
Das Prinzip der Konkretisierung durch Vererbung ermöglicht es, daß eine Instanz einer allgemeineren Oberklasse durch eine konkretere Instanz einer Unterklasse ersetzt bzw. überschrieben wird. Grundsätzlich wäre es nun wünschenswert, daß die im Rahmen einer Validierung für das spezielle Umfeld überprüften Constraints der Oberklasse durch den Konkretisierungsvorgang nicht durchbrochen werden können. Da die Vererbungsregeln jedoch aus pragmatischen Gründen nie so eng gefaßt werden können, so daß dies eingehalten werden kann, muß eine konkretere Objektnetz-Spezifikation in der Regel immer erneut auf die Einhaltung ihrer Constraints geprüft werden.

3.2.2.2 Objektnetz Meta-Klassen

Die Objektnetz-Methodik definiert drei Objektnetz-Meta-Klassen: Die Abstrakte Objektnetz-Klasse (AONC – *Abstract Object Net Class*), die hierarchische Objektnetz-Klasse (HONC – *Hierarchical Object Net Class*) und die elementare Objektnetz-Klasse (EONC – *Elementary Object Net Class*). Eine hierarchische Objektnetz-Klasse (HONC – *Hierarchical Object Net*

Class) enthält dabei mehrere Objektnetz-Instanzen (ONI), welche entweder Instanzen einer abstrakten, einer hierarchischen oder einer elementaren Objektnetz-Klasse sind. Abstrakte Objektnetz-Klassen werden eingesetzt, wenn der grundsätzliche Konkretisierungsweg noch offen ist; sie können folglich als Verallgemeinerung von HONC und EONC aufgefaßt werden. Abbildung 3.6 zeigt den Zusammenhang zwischen den drei Objektnetz-Meta-Klassen als UML-Klassendiagramm. Die Notation erfolgt nach den Regeln der *Unified Modeling Language* Version 1.1 [UML 97]. Eine neue Objektnetz-Klasse erbt immer nur von einer Oberklasse (*single inheritance*). Kapitel 3.3 behandelt alle drei Meta-Klassen differenziert.

Abbildung 3.6
Die Objektnetz-Meta-Klassen

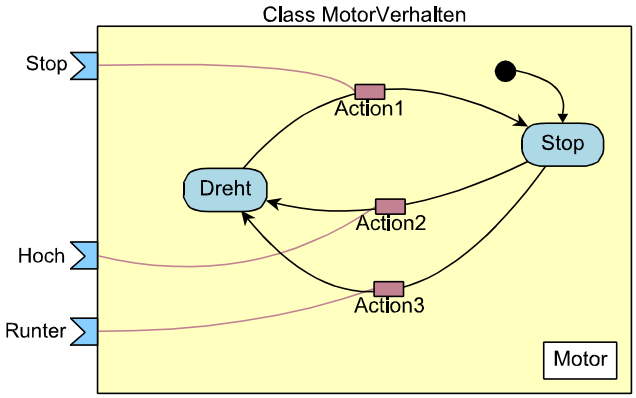


Zeigt sich im Laufe des Entwurfsprozesses, daß eine Klasse keine Nebenläufigkeiten enthält, so kann die Konkretisierung durch eine elementare ON-Klasse erfolgen.

Den elementaren ON-Klassen liegt das Modell erweiterter endlicher Zustandsmaschinen zugrunde [Hogrefe 89]. Sie spiegeln das Objektnetz-Basismodell, welches auf der logischen Realisierungsebene das elementare Abarbeitungsprinzip darstellt, direkt in die logische Entwurfsebene. Neben dem Objektverhalten, welches durch die erweiterte Zustandsmaschine festgelegt ist, sind die Interface-Elemente – die Ports – auf Klassenebene definiert. Vollständig wird eine EONC-Definition durch die Zuordnung der Ports zu den Zustandsübergängen. Die Manipulation von lokalen Attributen und der Zugriff auf Aktor/Sensor-Klassen erfolgt innerhalb von Software-Aktionen, die den Zustandsübergängen zugeordnet sind.

Abbildung 3.7 zeigt die elementare ON-Klasse *MotorVerhalten*, die das dynamische Verhalten eines Motors modelliert. Die erweiterte Zustandsmaschine verhindert, daß der Motor ohne eine vorherige Stop-Nachricht seine Drehrichtung umkehren kann. Die Aktionen *Action1*, *Action2* und *Action3* greifen jeweils direkt auf die drei Methoden (*MotorStop*, *MotorLinksLauf* und *MotorRechtsLauf*) der abstrakten Aktor/Sensor-Klasse *Motor* zu. Kapitel 3.3.3 geht detailliert auf den Aufbau und die Eigenschaften von elementaren ON-Klassen ein.

Abbildung 3.7
Elementare ON-Klasse, die das dynamische Verhalten des Motors modelliert



USES ASO: Motor		
TIMESCALE: 1 ms		
Attribute	Action	Guard
	Action1	
	Action2	
	Action3	

Zeigt der Entwurfsprozeß jedoch, daß sich ein Objekt durch Nebenläufigkeit auszeichnet, so kann eine abstrakte ON-Klasse durch eine hierarchische ON-Klasse konkretisiert werden. Hierarchische Klassen besitzen zwar die gleiche Port-Schnittstelle wie abstrakte oder elementare ON-Klassen, haben aber einen spezifischen internen Aufbau. Hierarchische Klassen bilden einen Container für mehrere ON-Instanzen. Wird z.B. die abstrakte Klasse *Schranken-Steuerung1* durch eine hierarchische ON-Klasse konkretisiert, so erbt die neue Unterklasse die Ports und die Constraints der abstrakten Klasse. Ausgewählte Ports der internen ON-Instanzen werden durch sogenannte Portexportierungen mit den geerbten Ports der Schnittstelle verschmolzen. Kapitel 3.3.4 beschreibt alle Aspekte hierarchischer ON-Klassen.

3.2.3 Prinzip der Plattformabstraktion

Aus bereits erläuterten Gründen sollen Objektnetz-Spezifikationen unabhängig von den schaltungstechnischen Details der Zielplattform entworfen werden. Bekanntermaßen zieht der Zugriff auf Aktoren und Sensoren innerhalb eines Objektnetz-Entwurfs bei der Implementierung die direkte Ansteuerung von plattformspezifischen Peripherie-Komponenten (wie z.B. D/A- oder A/D-Wandler) nach sich. Dies ist der Grund dafür, warum die Funktionalität der jeweiligen Aktoren und Sensoren in einer abstrahierten Form der logischen Entwurfsebene bereitgestellt werden.

Neben der Kapselung der Hardware-Ansteuerung repräsentieren die abstrakten Aktor/Sensor-Klassen (AS-Klassen) die Koppelelemente zwischen der Steuerungs- und Umgebungsspezifikation. Für die Implementierung der Steuerungsspezifikation müssen diese abstrakten AS-Klassen durch konkrete ersetzt werden. Diese konkreten Klassen stellen, unter der gleichen Schnittstelle, den jeweiligen plattformspezifischen Ansteuerungscode für den konkreten Aktor bzw. Sensor bereit.

Über diese Abstraktion hinaus hält eine Plattformspezifikation Informationen über das Kommunikationsnetzwerk bereit. Diese bilden unter anderem die Grundlage für die Zuordnung

(*mapping*) der Steuerung auf die verteilte Zielplattform. Für eine effektive Erstellung von Plattformspezifikationen stellt die Objektnetz-Methodik ein spezielles Framework (PAF) bereit; seine Beschreibung erfolgt in Kapitel 3.4.

3.3 Elemente und Entwurfsregeln der Objektnetze

Der folgende Abschnitt beschreibt schrittweise die Elemente, welche für die Erstellung von abstrakten und konkretisierten Objektnetzen eingesetzt werden. Neben den Elementen werden die dazugehörigen Entwurfsregeln vorgestellt. Zur Präzisierung der Ausführungen wird schrittweise das Meta-Modell der Objektnetz-Methodik (vgl. Abbildung 3.6) erweitert.

3.3.1 Datenklassen

Elementare Objektnetz-Klassen können durch lokale Variablen – die Attribute – erweitert werden. Ebenso können Nachrichten, die über Nachrichtenverbindungen zwischen den Instanzen ausgetauscht werden, Datenattribute mit sich führen. Innerhalb der Objektnetz-Methodik sind lokale Attribute und Nachrichtendaten immer von einem definierten Typ – infolge als Datenklasse bezeichnet. Die möglichen Wertebelegungen der Attribute sind durch die innerhalb der Datenklasse definierte Wertbelegungsmenge genau spezifiziert.

Für eine speicherplatzoptimierte Implementierung und effektive analytische Untersuchungen (vgl. Kapitel 7 und 6) ist eine möglichst große Einschränkung der Wertebe-legungsmenge von Bedeutung. Die konkrete Definition von Datenklassen erfolgt über eine der Objektnetz-Methodik zugeordnete Programmiersprache. Diese Programmiersprache muß allerdings die Definition von Datentypen unterstützen.

3.3.1.1 Vordefinierte Basisdatenklassen

Um mit speziellen Konstrukten beliebige Datenklassen mit einer definierten Wertbelegungsmenge definieren zu können, muß die eingesetzte Programmiersprache eine Reihe von vordefinierten Basisdatenklassen (vgl. Tabelle 3.2) bereitstellen. Die Festlegungen der Programmiersprache sorgen dafür, daß diese Datenklasse eine definierte Wertbelegungsmenge bzw. Wertebereich besitzen. Diese bilden die Grundlage für abgeleitete bzw. komplexe Datenklassen.

Die Datenklasse *generic* spielt bei der simulativen Validierung von Objektnetz-Modellen eine Sonderrolle. Soll auf einer Spezifikationsstufe ein Nachrichtentyp noch unspezifiziert bleiben, so kann diese Datenklasse eingesetzt werden. Nachrichtenattribute und lokale Attribute von dieser Datenklasse können somit während der Simulation mit beliebigen Werten belegt werden, unabhängig davon, ob es sich um Zeichenketten oder Zahlen handelt.

Tabelle 3.2
Mögliche vordefinierte Basisdatenklassen

Datenklasse	Belegung	Wertebereich
generic	Zeichenketten	beliebige Zeichenketten
string	Zeichenketten	beliebige Zeichenketten
char	ein Zeichen	1-Byte-Zeichen
int	natürliche Zahlen	4-Byte-Integerzahlen
double	rationale Zahlen	8-Byte-Realzahlen
boolean	Boole'sche Werte	*false*, *true*, 0, 1

Diese polymorphe Datenhaltung wird dadurch erzielt, daß auf Modellebene – nicht für die Implementierung auf der Zielplattform – eine Programmiersprache eingesetzt wird, die die interne Speicherung von Daten als Zeichenkette realisiert. Neben dieser Eigenschaft ist für eine vollständige Validierung der Objektnetz-Modelle, ohne zusätzlicher Compilierungsvorgang, die Interpretierbarkeit von Ausdrücken der Programmiersprache bedeutsam. Die applikationsspezifisch erweiterbare Skriptsprache Tcl (*Tool Command Language*) [Ousterhout 95], [Welch 97], [TCL 98] verbindet diese beiden Anforderung. Die im Standard-Tcl-Interpreter fehlenden Konstrukte zur Definition von Datenklassen wurden von [Holbe 97] als Tcl-Erweiterungsmodul (*package*) ergänzt. Als alternativ einsetzbare Skriptsprache ist Python [LöwFis 97] zu nennen.

Alle weiteren Datenklassen (*int*, *double*, *char*, *boolean*) sind intern auch nur Zeichenketten, jedoch mit einer eingeschränkten Menge von zugelassenen Zeichenfolgen. Somit besteht bezüglich ihrer Verwendung kein Unterschied zwischen den Datenklassen *string* und *generic*. Durch einfache Vererbung können aus den in Tabelle 3.2 gezeigten vordefinierten Basisdatenklassen weitere applikationsspezifische Basisdatenklassen abgeleitet werden.

Nachrichten ohne Dateninhalt, d.h. Nachrichten, die reine Steuerinformation besitzen, werden durch *control* gekennzeichnet. Es wird also nur dargestellt, ob eine Nachricht vorliegt. *Control* ist somit keine weitere Datenklasse, sondern der Bezeichner für Steuerinformationen. Wird z.B. einem Port (z.B. *Stop*, *Hoch*, *Runter* in Abbildung 3.7) keine spezielle Datenklasse zugeordnet, so wird automatisch ein Steuerport (*control*) impliziert.

3.3.1.2 Definition abgeleiteter Basisdatenklassen

Innerhalb der Objektnetz-Methodik kann eine neue applikationsspezifische Basisdatenklasse durch Vererbung aus einer oder mehreren bereits bestehenden Basisdatenklassen abgeleitet werden. Bei diesem Vorgang übernimmt die neue Datenklasse additiv die definierten Wertebereiche der Oberklassen. Durch zusätzliche optionale Angaben – den *Valuecheckern* – kann der neue additiv entstandene Wertebereich weiter eingeschränkt werden. Der Mechanis-mus der Vererbung von Wertebereichen und deren Einschränkung durch sogenannte *Valuechecker* erfolgt durch das von [Holbe 97] erstellte Erweiterungsmodul, welches integraler Bestandteil des

Simulators OPNTcl (*Object Petri Nets based on Tcl*) [OPNTCL 98], [UnDäNü 98] ist. OPNTcl bietet die Möglichkeit Objektnetz-Modelle ohne Compilierung auszuführen.

Abbildung 3.8
Das Kommando DataClass zur Erstellung neuer Basisdatenklassen

```
DataClass classname superclasses ?valuechecker? ?initvalue?
```

Die Datenklassendefinition ist ein Skript-Kommando (*DataClass*, vgl. Abbildung 3.8) dieses Erweiterungsmoduls und folgt somit ebenfalls den Syntaxregeln von Tcl.

Abbildung 3.9
Beispielhafte Erstellung neuer Basisdatenklassen mit erweitertem Tcl als Modellierungssprache

```
# Beispielklasse mit den Eigenschaften von string
DataClass text string

# Beispielklasse mit nur vier Farbnamen
DataClass colours string {enum {red green blue yellow}}

# Klasse mit Integerwerten im Bereich 0-255
DataClass byte int {rangeint {0 to 255}}

# Klasse mit 0 und nur negative double Werten
DataClass negativeDouble double {rangedouble {to 0}}

# Beispielklasse mit Strings mit einer minimalen
# Laenge von 3 und einer maximalen Laenge von 5 Zeichen
DataClass shortString string {rangelength {3 to 5}}

# Klasse mit nur geraden Integer-Werten
DataClass even int {basetcl {
   if {!($value % 2)} {return 1}
     return 0
}}
```

Das Argument *classname* spezifiziert den Namen der neuen Datenklasse. Das Argument *superclasses* ist eine Liste von bereits bestehende Klassen, deren Eigenschaften die neue Klasse erbt. Das optionale Argument *valuechecker* beschreibt den der Klasse zugeordneten Valuechecker, der aus zwei Listenelementen besteht. Das erste Element legt die Art des Valuecheckers fest (z.B. *enum* für einfache Aufzählungen), das zweite Element enthält die spezifischen Definitionskomponenten (z.B. eine Aufzählung von Wertbelegungen), die hier nur beispielhaft (vgl. Abbildung 3.9) betrachtet werden. Genauere Angaben hierzu liefert die Online-Dokumentation von OPNTcl [OPNTCL 98] und [Holbe 97]. Das ebenfalls optionale Argument *initvalue* ermöglicht die Angabe einer Default-Wertbelegung.

3.3.1.3 Definition komplexer Datenklassen

Abbildung 3.10
Beispielhafte Erstellung komplexer Datenklassen

```
# Komplexe Klasse zur Speicherung von Datumsangaben
DataClass mydate {{int day} {string month} {int year}}

# Erweiterung der Klasse mydate um Sekunden
DataClass mytime {mydate {int seconds}}

# Klasse, die nur gueltige Datumsangaben enthaelt
# WICHTIG: bereits Initialwerte muessen gueltig sein
# daher: Definition einer neuen Basisklasse mit entsprechendem
#        Initialwert
# Anmerkung:
# Initialwert von Int ist 1.
DataClass Int int {} 1
DataClass date {{Int day} {Int month} {Int year}} {complextcl {
   if {$day<1 || $day>31} {return 0}
   if {$month<1 || $month>12} {return 0}
   if {$day<28} {return 1}
   if {$month==2} {
      if {$day>29} {return 0}
      if {$day==29} {
         if {!($year % 100)} {
            if {!($year % 400)} {return 0}
         } else {
            if {$year % 4} {return 0}
         }
         return 1
      }
      return 1
   }
   if {[lsearch -exact {1 3 5 7 8 10 12} $month]!=-1} {
      return 1
   }
   if {$day<31} {return 1}
   return 0
}}
```

Neben neuen Basisdatenklassen können auch strukturierte Datenklassen, sogenannte komplexe Datenklassen, erstellt werden. Komplexe Datenklassen ermöglichen es, verschiedenste Komponenten bestehender Datenklassen (komplexe oder Basisdatenklassen) in einer neuen Datenklasse zu kombinieren. Das erweiterte Tcl bietet das gleiche Skript-Kommando zur Definition komplexer Datenklassen an (vgl. Abbildung 3.11).

Abbildung 3.11
Das Kommando DataClass zur Definition komplexer Datenklassen

```
DataClass classname classdefinition ?valuechecker?
```

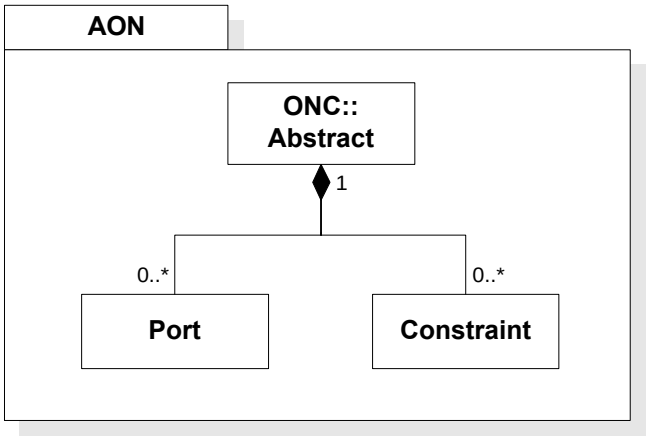
Das Argument *classdefinition* ist eine Liste. Die Listenelemente können einerseits die Namen der Datenklassen sein, deren Eigenschaften die neue Datenklasse erben soll. Anderseits können

die Listenelemente zweielementige Listen sein, die die Komponenten der neuen Klasse beschreiben. Das erste Element dieser zweielementigen Listen kennzeichnet die Datenklasse der Sub-Komponente, das zweite Element einen Bezeichner (Deskriptor) über den auf die Komponente zugegriffen werden kann. Abbildung 3.10 zeigt beispielhaft, wie mit dem erweiterten Tcl neue komplexe Datenklassen definiert werden können.

3.3.2 Abstrakte Objektnetz-Klassen

Abstrakte ON-Klassen besitzen kein konkretisiertes Verhalten. Sie können jedoch bereits über ein definiertes Klassen-Interface verfügen, welches aus einer Menge von sogenannten Ports besteht; über Ports werden an- und abgehende Nachrichten geleitet. Durch die zusätzliche Angabe von speziellen Constraints können der Klasse bereits bestimmte Einschränkungen und Zwänge für den Empfang und Versand von Nachrichten auferlegt werden. Diese Zwänge können bei der Konkretisierung dieser Klasse auf ihre Einhaltung hin überprüft werden.

Abbildung 3.12
Aufbau abstrakter Objektnetz-Klassen



Wie bereits aus Abbildung 3.6 ersichtlich ist, können abstrakte Klassen zu hierarchischen oder elementaren Klassen konkretisiert werden. Folglich besitzen abstrakte Klassen nur die Elemente, die sowohl elementare als auch hierarchische Klassen besitzen. Abbildung 3.12 zeigt als UML-Klassendiagramm den grundsätzlichen Aufbau abstrakter Objektnetz-Klassen mit Ports und Constraints als mögliche Elemente.

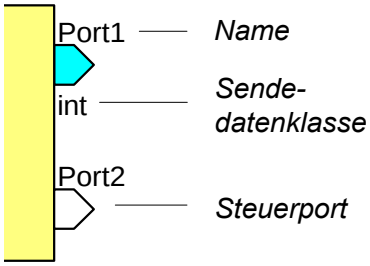
3.3.2.1 Ports als Interface-Elemente

Ports verkörpern die Interface-Elemente einer Objektnetz-Klasse. Eine Objektnetz-Instanz kommuniziert mit den Instanzen in ihrem Umfeld ausschließlich über diese Ports. Nachrichten werden über Empfangsports entgegengenommen. Über Sendeports werden Nachrichten an Instanzen im Umfeld abgegeben. Spezielle Parameterports ermöglichen die Parametrisierung klasseninterner Attribute.

Innerhalb einer Objektnetz-Klasse werden Ports eindeutig durch ihren Namen unterschieden. Im Rahmen der Konkretisierung von Objektnetz-Klassen durch Vererbung können Ports durch namensgleiche neue Ports überschrieben werden. Der neue Port muß sich jedoch in seinen grundlegenden Eigenschaften, wie z.B. seinem Grundtyp, an den überschriebenen halten. Die Objektnetz-Methodik definiert fünf verschiedene Portgrundtypen:

- **Asynchrone Sendeports - ASP:**

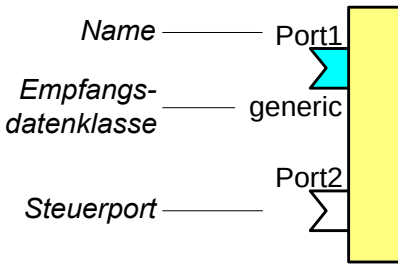
Abbildung 3.13
Asynchrone Sendeports



Die Abbildung 3.13 zeigt eine Objektnetz-Klasse mit zwei asynchronen Sendeports (*Asynchronous Send Port* – ASP). Über diese Sendeports können Nachrichten, ohne auf eine Rückmeldung von der Empfängerinstanz zu warten, abgesendet werden. Über *Port1* (Datenport) kann die Instanz Nachrichten der Datenklasse *int* versenden. Für *Port2* wurde keine Sendedatenklasse angegeben, es handelt sich somit um einen Steuerport. Über ihn werden Steuernachrichten versendet.

- **Asynchrone Empfangsports - ARP:**

Abbildung 3.14
Asynchrone Empfangsports



Asynchrone Empfangsports (*Asynchronous Receive Port* – ARP) bilden das komplementäre Gegenstück zu den asynchronen Sendeports. Abbildung 3.14 zeigt zwei unterschiedliche Empfangsports. Bei *Port1* handelt es sich um einen Datenport, für den die Empfangsdatenklasse *generic* angegeben wurde. *Port1* kann folglich Nachrichten aller Datenklassen verarbeiten. *Port2* ist ein Steuerport.

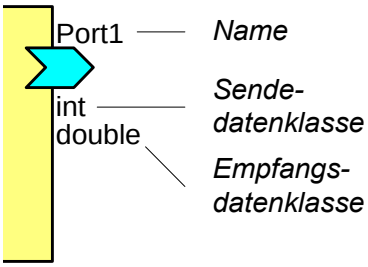
Jeder Empfangsport realisiert einen Zwischenspeicher nach dem FIFO-Prinzip. Zusätzlich zur Datenklasse muß jedem Empfangsport eine Pufferzeit (*timeout*) zugewiesen werden. Wird die Pufferzeit 0 angegeben, so ist die Nachricht für genau einen Bearbeitungszyklus der Instanz gültig.

Für dringende Nachrichten kann dem Port (nur ARP) eine erhöhte Priorität zugeordnet werden. Die Prioritätsstufe ist somit eine Eigenschaft des Ports und nicht der Nachricht. Die Objektnetz-Methodik unterscheidet eine Prioritätsstufe *standard* und mehrere Stufen *important*. Ein Mechanismus sorgt in Zusammenarbeit mit dem Knotenscheduler dafür, daß eine ON-Instanz mit einer *important* Nachricht in seiner Empfangswarteschlange zum nächstmöglichen Zeitpunkt (sortiert nach Priorität) den Steuerfokus erhält. Die Beschränkung auf asynchrone Ports mit erhöhter Priorität erfolgt aus Überlegungen der Anwendung wichtiger Nachrichten (Alarme, Fehlerbehandlung u.ä.) heraus und nicht aus Sicht der Implementierung [Schulze 97].

● Synchrone Sendeports - SSP:

Das Versenden einer Nachrichten über einen synchronen Sendeport (*Synchronous Send Port* – SSP) ist erst abgeschlossen, sobald eine Antwortnachricht der Gegenseite (ein synchroner Empfangsport) empfangen wurde. Für synchrone Empfangs- bzw. Sendeports können unterschiedliche Datenklassen für die Nachricht bzw. die Antwortnachricht angegeben werden. Ebenso ist es möglich, eine Nachricht z.B. der Datenklasse *int* zu senden, und als Antwort nur eine Steuernachricht zu erwarten.

Abbildung 3.15
Synchroner Sendeport

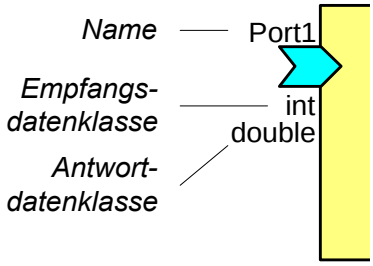


Die Zeit, die der Sendeport auf eine Antwort wartet, kann durch Angabe einer Wartezeit (*timeout*) begrenzt werden. Läuft diese Zeit ab, ohne daß eine Antwort empfangen wurde, wird eine Fehlerbehandlung ausgelöst (vgl. Kapitel 5.2).

● Synchrone Empfangsports - SRP:

Synchrone Empfangsports (*Synchronous Receive Port* – SRP) bilden das komplementäre Gegenstück zu den SSPs. Nach dem Empfang einer Nachricht wird zum Sendeport eine Antwortnachricht zurückgesendet. Analog zum ARP werden beim synchronen Empfangsport empfangene Nachrichten nach einer vorgegebenen Pufferzeit (*timeout*) ungültig. Das Ablaufen der Pufferzeit hat die Sendung einer Fehlerantwortnachricht zur Folge. Diese Nachricht löst beim Sender eine Fehlerbehandlung aus. Abbildung 3.16 zeigt einen SRP mit unterschiedlichen Empfangs- und Antwortdatenklassen.

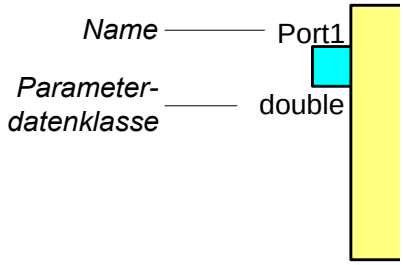
Abbildung 3.16
Synchroner Empfangsport



● Parameterports - PP:

Der Parameterport unterscheidet sich grundsätzlich von den bisher vorgestellten Porttypen. Er dient ausschließlich dazu, um einem internen Attribut (lokale Variable) einen konstanten Wert zuzuweisen. Auf Attribute, die auf diese Weise initialisiert wurden, kann nur lesend zugegriffen werden.

Abbildung 3.17
Parameterport



Vererbung: Im Rahmen der Konkretisierung durch Vererbung werden alle in einer Oberklasse definierten Ports an die Unterklassen vererbt. In den Unterklassen können diese Ports nach definierten Restriktionen überschrieben bzw.

verändert werden. Grundsätzlich gilt dabei die Regel, daß das Klassen-Interface nur erweitert, aber nicht eingeschränkt werden kann.

- Der Portgrundtyp (ASP, ARP, SSP, SRP oder PP) kann nicht verändert werden.
- Die Empfangsdatenklasse eines Ports kann nur durch bestimmte Datenklassen über-schrieben werden. Die Datenklassen müssen gewährleisten, daß die Nachrichten, die für den überschriebenen Port bestimmt waren, ohne Informationsverlust durch den neuen Port verarbeitet werden können. Folglich können Basisdatenklassen nur durch ihre allge-meineren Oberklassen überschrieben werden (z.B. *byte* durch *int*). Komplexe Datenklassen können nur durch ihre Unterklassen überschrieben werden (z.B. *mydate* durch *mytime*). Steuernachrichten (*control*) können durch jede Datenklasse überschrieben werden.
- Die Sende- bzw. Antwortdatenklasse eines Ports kann im Umkehrschluß nur durch eine Datenklasse überschrieben werden, die die Informationsmenge einschränkt. Damit wird gewährleitet, daß der neue Port keine Informationen abgibt, die das unveränderte Umfeld nicht mehr verarbeiten kann. Sende- und Antwortdatenklassen verhalten sich somit komplementär zu den Empfangsdatenklassen.

Diese Regeln garantieren die statische Polymorphie von Objektnetz-Klassen. Statische Polymorphie bedeutet innerhalb der Objektnetz-Methodik, daß Instanzen konkretisierter Objektnetz-Klassen immer im gleichen Umfeld (einer hierarchischen Klasse) wie eine Instanz ihrer allgemeineren Oberklasse eingesetzt werden können, ohne daß das Umfeld der Instanz verändert werden muß.

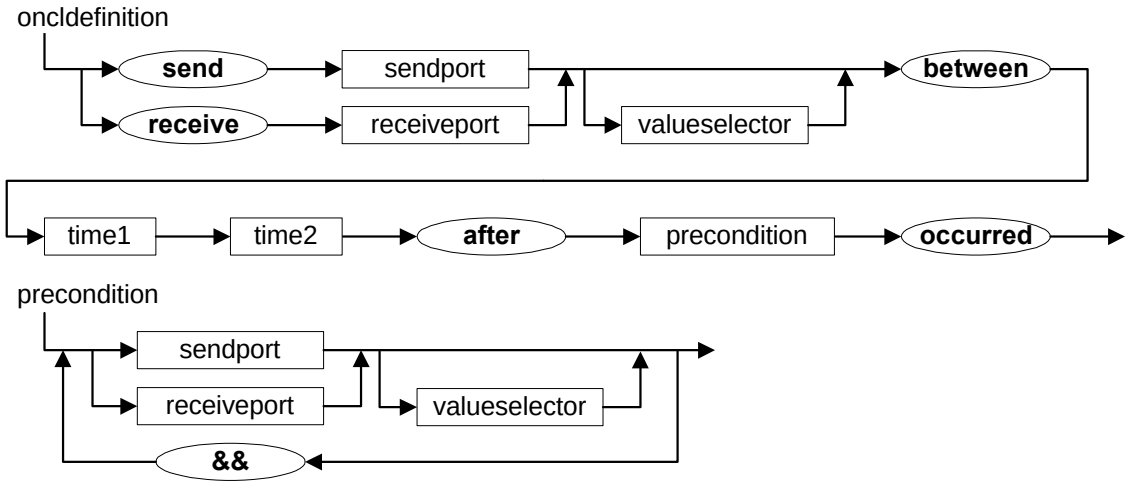
3.3.2.2 Constraints - Temporale und kausale Zwänge

Angelehnt an die Semantik der Sequenzdiagramme [UML 97], definiert die Objektnetz-Methodik eine formale Beschreibungssprache, die *Object Net Constraint Language* (ONCL). Sie ermöglicht es dem Designer, kausale und temporale Zwänge für die Abfolge von Nachrichten, die eine Objektnetz-Instanz erreichen bzw. verlassen, auf Klassenebene formal zu notieren. Die Verwendung einer textuellen Notation gewährleistet, im Gegensatz zu den Sequenzdiagrammen, eine wesentlich kompaktere Notationsform. Über Constraints können auf der Ebene der abstrakten Klassen besonders wichtige Eigenschaften, die in der Regel eine hohe Sicherheitsrelevanz besitzen, spezifiziert werden. Während der Validierung kann die Einhaltung dieser Constraints überprüft werden. Jedes Constraint schränkt den durch die Portdefinitionen festgelegten Spielraum einer Instanz, mit Instanzen des Umfelds zu kommunzieren, ein.

Das Syntaxdiagramm in Abbildung 3.18 definiert den Aufbau der unterstützten Basis-ONCL. Eine Erweiterung um komplexere Konstrukte ist in Anlehnung an dieses Diagramm möglich. Jeder durch eine ONCL-Definition formulierte Constraint legt fest, unter welchen zeitlichen (nicht früher als nach *time1* und nicht später als nach *time2*) und kausalen Bedingungen entweder eine Nachricht, mit spezifischem Inhalt (*valueselector*), an einem speziellen Empfangsport (*receiveport*) erwartet wird, bzw. an einem Sendeport (*sendport*) abgesendet werden muß. *Precondition* legt fest, an welchen Ports welche Nachrichten als auslösende Vorbedingung für den Constraint zuvor eintreffen bzw. abgesendet werden müssen. In Kapitel 5.3 wird das

Verhalten der Constraints mittels Petri-Netzen beschrieben. Die dargestellten Netze enthalten sogenannte verbotene Transitionen, vergleichbar den Fakten in [Starke 80]. Das Schalten der verbotenen Transitionen signalisiert eine Constraint- Verletzung.

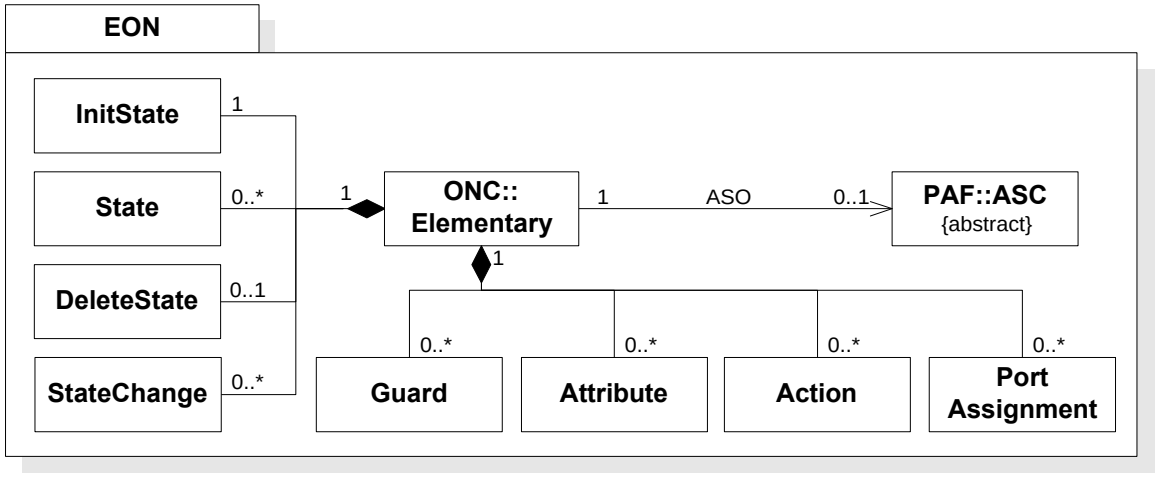
Abbildung 3.18
Syntaxdiagramm der ONCL-Definition



Vererbung: Im Rahmen der Konkretisierung durch Vererbung werden alle in einer typischerweise abstrakten Oberklasse definierten Contraints an die Unterklassen, die entweder abstrakt, elementar oder hierarchisch sein können, weiter vererbt. In den Unterklassen können den geerbten Constraints weitere hinzugefügt werden.

3.3.3 Elementare Objektnetz-Klassen

Abbildung 3.19
Aufbau elementarer Objektnetz-Klassen



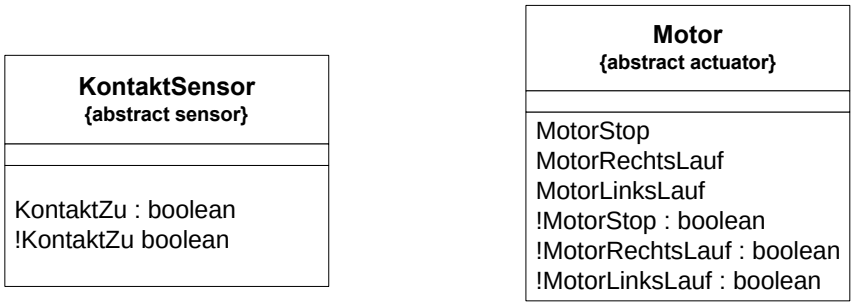
Das Verhalten von Objektnetz-Klassen auf der untersten logischen Entwurfsebene wird durch eine erweiterte endliche Zustandsmaschine modelliert. Diese Klassen werden daher als elementare ON-Klassen (*Elementary Object Net Classes* – EONC) bezeichnet.

Durch Vererbung können aus abstrakten ON-Klassen elementare abgeleitet werden. Diese erben automatisch Ports und Constraints der abstrakten Oberklasse. Im Rahmen der Konkretisierung können Instanzen dieser elementaren Unterklassen alle Instanzen der allgemeineren Oberklasse ersetzen. Die abgeleiteten elementaren Klassen können schließlich weiter konkretisiert werden. Alle abgeleiteten Unterklassen einer elementaren ON-Klasse sind automatisch wieder elementar. Abbildung 3.19 benennt die einzelnen Komponenten elementarer Klassen.

• Referenz auf Aktor/Sensor-Objekte - ASO:

Abstrakte Aktor/Sensor-Klassen (AS-Klassen) repräsentieren die Funktionalität von peripheren Aktoren bzw. Sensoren. Objekte dieser abstrakten Klassen, die AS-Objekte (ASO), realisieren die Kopplung zur Umgebung. Jede AS-Klasse definiert hierzu eine oder mehrere abstrakte Methoden, die den Zugriff auf die Umgebung über plattformspezifische Hardware-Komponenten (z.B. D/A- oder A/D-Wandler) kapseln. Die Implementierung der abstrakten Methoden bleibt auf dieser Entwurfsebene offen. Sie erfolgt erst durch konkrete AS-Unter-klassen, die eine spezifische Implementierung bereitstellen (vgl. Kapitel 3.4). Auch wenn für die abstrakten Methoden noch keine Implementierung angegeben werden muß, so müssen doch zumindest die Datenklassen für Übergabe- und Rückgabeparameter definiert werden. Sensoren besitzen Methoden (z.B. *KontaktZu* in Abbildung 3.20) mit einem Rückgabeparameter. Aktoren besitzen Methoden mit einem Übergabeparameter. Der Übergabeparameter kann optional entfallen. In diesem Fall wird ein Boole‘scher Parameter mit dem Wert *true* impliziert. Eine AS-Methode weist somit Analogien zu einem synchronen Empfangsport (SRP) auf. Im Gegensatz zu diesem verbirgt sich hinter einer AS-Methode jedoch kein Pufferungs-mechanismus. Die Methoden des AS-Objekts erweitern lediglich den Befehlsvorrat der Software-Prozeduren (Aktionen und Guards), die innerhalb einer elementaren ON-Klasse definiert werden können.

Abbildung 3.20
Zwei abstrakte AS-Klassen mit ihren Schnittstellendefinitionen



Da über die referenzierten Aktor/Sensor-Objekte die Kopplung zwischen Steuerungs- und Umgebungsspezifikation erfolgt, muß jedes in der Steuerung angesprochene AS-Objekt auch von

der Umgebung angesprochen werden. Für diese Zugriffe durch die Umgebung stellt eine abstrakte AS-Klasse für jede Methode automatisch ein komplementäres Gegenstück bereit. Diese komplementären Methoden, die durch ein vorangestelltes Ausrufezeichen (vgl. Abbildung 3.20) als solche gekennzeichnet sind, dienen z.B. dazu, einem Sensor seinen Kontaktzustand zuzuordnen. D. h. Sensoren der Steuerung werden zu Aktoren der Umgebung, und Aktoren der Steuerung werden zu Sensoren der Umgebung. Dabei werden Übergabeparameter zu Rückgabeparametern und umgekehrt.

Die Bedingung dafür, daß von einer Instanz innerhalb der Steuerungsspezifikation das gleiche AS-Objekt angesprochen wird wie von einer Instanz der Umgebungsspezifikation, ist zum einen die gleiche AS-Klasse der referenzierten AS-Objekte und zum anderen der gleiche Instanzname der referenzierenden ON-Instanzen. (z.B. *SensorLinks* in Abbildung 3.4).

Vererbung: Referenzen auf AS-Objekte, die in einer elementaren ON-Klasse definiert wurden, werden bei der Konkretisierung dieser Klasse unverändert vererbt. ASO-Referenzen können nicht überschrieben bzw. umdefiniert werden.

- **Attribute:**

Lokale Variablen – die Attribute – erweitern den Zustandsraum einer elementaren ON-Klasse. Attribute sind Objekte der in Kapitel 3.3.1 beschriebenen Datenklassen. Es wird zwischen parametrisierten und normalen Attributen unterschieden. Parametrisierte Attribute erhalten bei der Instanzierung einer ON-Klasse durch einen zugeordneten Parameterport ihre initiale Wertbelegung. Normale Attribute hingegen erhalten bereits auf Klassenebene ihre Startbelegung.

Tabelle 3.3 zeigt beispielhaft für das erweiterte Tcl als Modellierungssprache die Definition und Manipulation von Attributen (innerhalb einer Aktion oder eines Guards).

Tabelle 3.3
Beispielhafte Initialisierung und Manipulation von Attributen

Datenklasse	Name	Initialisierung/Port	Zugriff lesend	Zugriff schreibend
string	*str1*	*"Hallo World!"*	*Get str1*	*Set str1 "Hallo"*
double	*value1*	*3.56e-23*	*Get value1*	*Set value1 0.0*
date	*d1*	*{{day 31} {month 12} {year 2000}}*	*Get d1* *Get d1 month*	*Set d1 month 1* *Set d1 {{month 1} {day 1}}*
boolean	*result*	*false*	*Get result*	*Set result true*
double	*value2*	*Port2*	*Get value2*	-

Vererbung: Die Objektnetz-Methodik erlaubt es, Attribute zu denen von der Oberklasse geerbten hinzuzufügen. Die Datenklasse eines geerbten Attributes kann gemäß den Regeln, wie sie für die Empfangsdatenklasse eines Ports formuliert wurden, überschreiben bzw. erweitert werden (vgl. Seite 46ff). Die geerbte Initialisierung kann nur erweitert werden.

- **Aktionen:**

Aktionen sind parameterlose Softwareprozeduren, die einem oder mehreren Zuständen bzw. Zustandswechseln zugeordnet werden können. Sie haben Zugriff auf sämtliche Attribute und AS-Methoden ihrer Instanz. Ebenso können sie auf alle Nachrichtenpuffer der Ports (über Referenzen) zugreifen, die dem aufrufenden Zustandswechsel zugeordnet wurden. Aktionen können prinzipiell beliebigen Code enthalten, der Anwender sollte sich jedoch vor Augen führen, daß Aktionen (vgl. Kapitel 5 und 7) exklusiv und ununterbrechbar ausgeführt werden. Damit ist eine Aktion einem kritischen Bereich bei preemptiven Multitasking [Tanenbaum 95] gleichzusetzen. Sie sollten daher grundsätzlich von kurzer Ausführungsdauer sein.

Für die Validierung einer Spezifikation unter realen Zeitbedingungen ist es notwendig, möglichst genaue Schranken für die Ausführungsdauer einer Aktion zu kennen. Es wird hierzu eine maximale Abarbeitungszeit ermittelt. Diese Abarbeitungszeit wird auf einen Referenzrechner bezogen und als relative Zeiteinheiten zusammen mit der Aktion gespeichert (*maxduration*). Zur Bestimmung dieses Wertes wird die Aktion mit ihren typischen Parameter auf einem beliebigen Rechner (z.B. die Entwicklungsplattform) abgearbeitet. Die dabei ermittelte Maximalzeit wird durch die Abarbeitungszeit einer definierten Benchmark-Aktion, welche auf dem gleichen Rechner ermittelt wurde, dividiert und mit der Abarbeitungszeit der gleichen Benchmark-Aktion, die auf dem Referenzrechner ermittelt wurde, multipliziert. Wird nun eine Aktion (zusammen mit ihrer elementaren ON-Instanz) schließlich für die Implementierung auf einem realen Rechnerknoten ausgewählt (*mapping*), so werden aus diesen normierten Zeiteinheiten reale Zeiten [HenPat 93] (vgl. Kapitel 5.2).

Die Definition des Aktionscodes erfolgt in der Modellierungssprache der Objektnetz-Methodik, dem erweiterten Tcl. Dieser Code kann ohne Compilierung direkt im Objektnetz-Simulator zur Ausführung gebracht werden. Soll nun die Steuerungsspezifikation auf einer verteilten Zielplattform implementiert werden, so müssen die Aktionen in der Sprache C bereitgestellt werden; C-Compiler sind für nahezu alle Plattformen verfügbar. Da eine Transformation von Tcl nach C oft unmöglich ist, besteht die Option die Aktionen direkt in C zu notieren. Nach der Compilierung können diese ebenfalls im Simulator ausgeführt werden.

Vererbung: Eine elementare ON-Klasse erbt alle Aktionen ihrer Oberklasse. Prinzipiell ist es möglich, die geerbten Aktionen mit geändertem Code zu überschreiben.

- **Guards:**

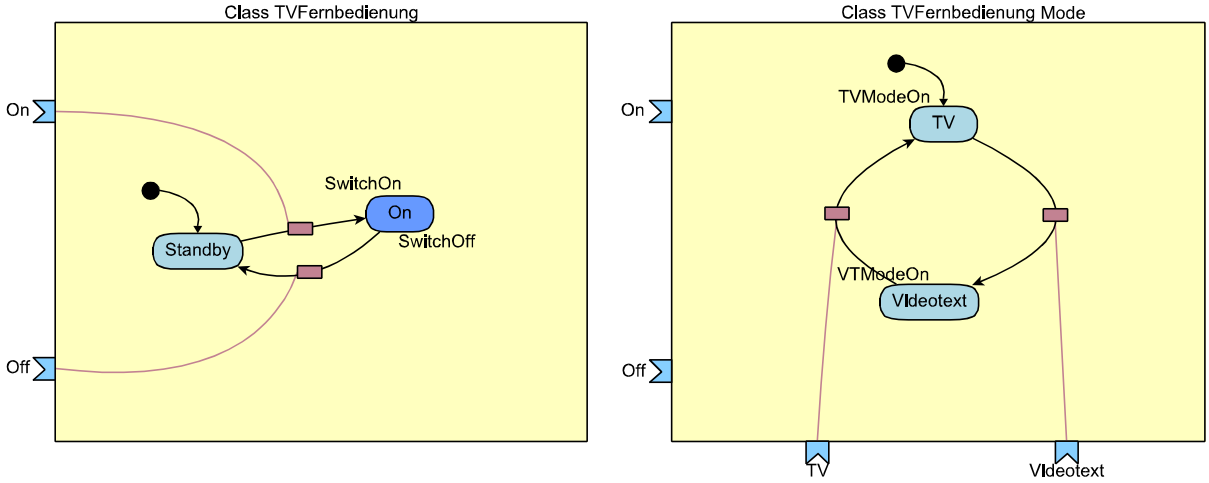
Guards sind parameterlose Softwarefunktionen mit Boole‘schem Rückgabewert. Sie werden im Gegensatz zu Aktionen nur von Zustandswechseln referenziert. Sie dienen der Ermöglichung bedingter Zustandswechsel. Guards haben lesenden Zugriff auf alle Attribute und Empfangsportpuffer ihrer Instanz. Guards können analog den Aktionen AS-Methoden enthalten, jedoch nur solche, die lesenden Charakter (Sensor-Methoden) besitzen. Die Ausführung eines Guards darf keinesfalls den Systemzustand verändern. Alle weiteren Eigenschaften der Guards decken sich mit den Aktionen.

- **Die Zustandsmaschine mit Zuständen und Zustandswechseln:**

Eine hierarchische Zustandsmaschine bestehend aus Zuständen (*States*) und Zustandswechsel (*State Changes*) bildet den Kern einer elementaren ON-Klasse. Sie modelliert den Lebenszyklus der elementaren ON-Instanzen. Die Zustände bilden die Spanne, in denen eine Instanz auf ein Ereignis (die Ankunft einer Nachricht oder den Ablauf einer Wartezeit) wartet. Zustandswechsel definieren den Übergang zwischen zwei Zuständen. Ein Zustand stellt über seine Vorzustände eine Aussage über schon empfangene Nachrichten dar [Schulze 97]. Vergleichbar mit den Statecharts [Harel 87] können Zustände hierarchisch untergliedert werden. Parallele Zustände werden analog zu den ROOMcharts [SeGuWa 94] ausgeschlossen (vgl. Kapitel 2).

Die hierarchische Zustandsmaschine einer elementaren ON-Klasse enthält typischerweise mehrere Zustände (z.B. *Standby* und *On*, siehe linke Seite von Abbildung 3.21). Durch die Angabe des **Startzustandes** (*init state*) ist jeder Zustand durch seine möglichen Vorzustände eindeutig charakterisiert. Optinal kann jede Zustandsmaschine einen **Endzustand** (*delete state*) besitzen. Das Erreichen dieses Zustandes beendet die Ausführung der jeweiligen Instanz. Jeder Zustand kann mit einer Aktion für den Eintritt (z.B. *SwitchOn*) in den Zustand (*entry action*) und mit einer Aktion für das Verlassen (z.B. *SwitchOff*) des Zustandes (*exit action*) bewertet werden.

Abbildung 3.21
Elementare ON-Klasse mit hierarchisch konkretisierten Oberzustand *On* (linke Seite) [Langbein 98]



Jeder Zustand (Oberzustand, hier *On*) kann durch eine Unterzustandmaschine (*sub state diagram*) hierarchisch verfeinert werden. Diese Unterzustandsmaschine (vgl. rechte Seite von Abbildung 3.21) besitzt wiederum einen Startzustand, der eingenommen wird, nachdem der zugehörige Oberzustand erreicht wurde. Die Unterzustandsmaschine kann aus jedem ihrer Zustände heraus wieder verlassen werden. Vor dem Eintritt in die Unterzustandsmaschine wird die Eintrittsaktion des Oberzustands (hier *SwitchOn*) ausgeführt. Nach dem Verlassen wird deren Austrittsaktion (*SwitchOff*) aktiviert. Jede Unterzustandsmaschine kann von verschiedenen Oberzuständen aus referenziert werden. Die Zustände der Unterzustands-maschine können ihrerseits wieder hierarchisch untergliedert werden.

Zustandswechsel innerhalb einer Zustandsmaschine definieren den Übergang zwischen zwei Zuständen (den Vorzustand mit dem Folgezustand). Ein Zustandswechsel kann einen Guard und eine Aktion referenzieren. Eine zusätzlich angegebene Wartezeit (*wait time*) kann den Zustandswechsel verzögern. Für die Dauer dieser Wartezeit muß sich die Instanz im Vorzustand befunden haben, bevor der Zustandswechsel erfolgen kann. Wurden dem Zustandswechsel synchrone Ports zugeordnet, so können weitere optionale Referenzen definiert werden: für den Fehlerfall einen alternativen Folgezustand (*error post state*); die Aktion (*error action*), die dabei ausgeführt wird; und eine weitere Aktion (*reply action*), für die Vorbereitung von Antwortnachrichten. Die vollständige Semantik eines Zustandswechsels mit den verschieden Referenzen und unterschiedlichen zugeordneten Ports wird detailliert in Kapitel 5.2 beschrieben.

Besitzen mehrere Zustandswechsel den gleichen Vorzustand, so muß ein möglicher Konflikt für den Fall, daß die elementare ON-Instanz Bestandteil einer Steuerungsspezifikation ist, ausgeschlossen werden. Dies kann durch Guards bzw. Portzuordnungen, oder durch eine unterschiedliche Priorisierung der betroffenen Zustandswechsel geschehen. Für den Fall, daß die Instanz zur Umgebungsspezifikation gehört, kann dagegen ein Konflikt sein. Während einer simulativen Validierung wird er Konflikt zufällig entschieden. Ebenso ist der Fall zu vermeiden, daß mehrere ARP mit der gleichen *important* Priorität innerhalb einer Instanz konkurrieren.

Vererbung: Das Konzept der Hierarchisierung der Zustandsmaschine kann sowohl als Mittel der Strukturierung als auch bei der Konkretisierung durch Vererbung eingesetzt werden. Hierbei wird ein nichthierarchischer Zustand der Oberklasse durch einen hierarchischen in der Unterklasse konkretisiert. Zustandswechsel können nicht hierarchisch verfeinert werden. Es ist jedoch gestattet, zusätzliche Ports ihnen zuzuordnen. Des weiteren können Referenzen auf Guards und Aktionen hinzugefügt werden, falls diese in der Oberklasse nicht bereits belegt waren. Belegte Referenzen können nicht geändert bzw. überschrieben werden.

● Portzuordnungen - PA:

Eine Portzuordnung (*Port Assignment* - PA) verbindet einen Port mit einem Zustandswechsel. Diese Verbindung wird durch eine hellere Linie (vgl. Abbildung 3.21) dargegestellt. Eine Portzuordnung verknüpft die für das Umfeld sichtbare (öffentliche) Schnittstelle einer elementaren ON-Klasse mit ihrer internen (privaten) Zustandsmaschine. Die Ankunft einer Nachricht an einem zugeordneten Empfangsport stellt eine zusätzliche Vorbedingung für den Zustandswechsel dar. Wurde ein Sendeport einem Zustandswechsel zugeordnet, so wird über diesen eine Nachricht abgesendet, sobald der Zustandswechsel erfolgte.

Ist der zugeordnete Port ein Datenport, so werden automatisch Referenzen auf die Puffer des Ports gebildet. Über diese Referenzen erfolgt der Zugriff auf die Daten der Puffer. Bei einem ARP erfolgt der Zugriff nur lesend, bei einem ASP nur schreibend. Handelt es sich um einen synchronen Port, so kann innerhalb einer Aktion über die Referenz schreibend auf die abgesendete Nachricht und innerhalb einer Aktion, einem Guard oder einem Selektor lesend auf die empfangene Nachricht zugegriffen werden. Zur Unterscheidung von lesbaren und schreibbaren Puffern wird bei der Bildung der Referenz ein Präfix (*send*, *reply* bzw. *receive*) dem Portnamen vorangestellt. Bei Datenempfangsports kann optional ein sog. Selektor (vgl. Tabelle

3.4) an die PA geschrieben werden. Selektoren sind spezielle Boole'sche Ausdrücke, die in Abhängigkeit vom Dateninhalt der empfangenen Nachricht abweisen oder passieren lassen.

Tabelle 3.4
Auswahl einiger Portzuordnungen mit Pufferreferenzen und Nachrichtenselektor

Porttyp	Name (Beispiel)	mögl. Referenzen	Selektor (Beispiel)
ARP	*P1*	*receive.P1*	*[Get receive.P1] > 34*
ASP	*P2*	*send.P2*	-
SRP	*P3*	*receive.P3 reply.P3*	*[Get receive.P3] == 100*
SSP	*P4*	*send.P4 receive.P4*	-

Einem Zustandswechsel darf maximal ein SRP und ein SSP zugeordnet werden. Für die Zuordnung von asynchronen Ports existieren keine Einschränkungen. Jeder Port kann grundsätzlich mehreren Zustandswechseln zugeordnet werden. Haben diese Zustandswechsel jedoch einen gemeinsamen Vorzustand, so müssen die Selektoren der PAs bzw. die Guards der Zustandswechsel (in der Steuerungsspezifikation) disjunkt sein, da sonst die Nachrichtenannahme konfliktbehaftet wäre. Für den Sendefall entsteht durch den sequentiellen Charakter der Zustandsmaschine kein Konflikt.

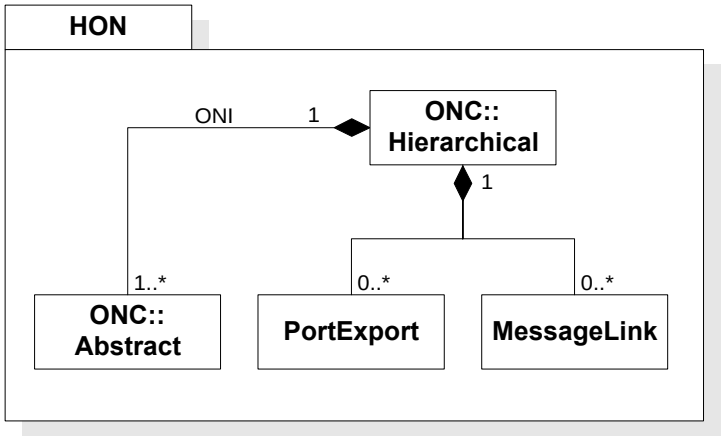
Vererbung: Portzuordnungen werden automatisch an die Unterklassen vererbt. Sie können dabei nicht verändert oder überschrieben werden.

3.3.4 Hierarchische Objektnetz-Klassen

Durch hierarchische Objektnetz-Klassen (*Hierachical Object Net Classes* – HONC) werden zwei verschiedene Entwurfsprinzipien unterstützt. Zum einen *Bottom-Up*, bei dem Instanzen bestehender (abstrakter, hierarchischer oder elementarer) Klassen zu einer neuen hierarchischen Klasse kombiniert werden (➪ Objekt-Komposition). Und zum anderen *Top-Down*, bei dem abstrakte Kassen durch schrittweise konkretisierte hierarchische Klassen – wie im Entwicklungsprozeß für Objektnetze definiert – ersetzt werden (➪ statische Polymorphie).

Durch die Verwendung einer hierarchischen ON-Klasse kann kein anderes Verhalten modelliert werden, welches nicht auch durch die entsprechende Menge von elementaren ON-Instanzen modelliert werden könnte. Hierarchische ON-Klassen dienen lediglich der Unterstützung des Designers bei einem strukturierten Vorgehen. Die gegenüber einer abstrakten ON-Klasse zusätzlichen Elemente (vgl. Abbildung 3.22) sind die Objektnetz-Instanzen (*ONI*), die Nachrichtenverbindungen (*MessageLink*) und die Portexportierungen (*PortExport*).

Abbildung 3.22
Der Aufbau hierarchischer Objektnetz-Klassen



Durch die Komposition mehrerer ON-Instanzen innerhalb einer hierarchischen ON-Klasse (die Container-Klasse der ON-Instanzen) kann im Unterschied zu einer elementaren ON-Klasse auch nebenläufiges Verhalten modelliert werden.

● Objektnetz-Instanzen - ONI:

Eine ON-Instanz wird innerhalb ihrer Container-Klasse (eine hierarchische ON-Klasse) eindeutig durch ihren Instanznamen (z.B. *BahnSchranke* in Abbildung 3.4) identifiziert. Zusätzlich zum Instanznamen müssen vorhandene Parameterports mit einer Initialisierung versehen werden. Wichtig ist, daß keine Instanzen der Container-Klasse bzw. von Unterklassen dieser eingefügt werden. Dies würde sonst die Einebnung der hierarchischen ON-Klasse (siehe Kapitel 4.2) aufgrund einer endlosen Rekursion unmöglich machen [Schmidt 98].

Vererbung: ON-Instanzen werden an die Unterklassen vererbt. In den Unterklassen können die geerbten ON-Instanzen zum Zwecke der Konkretisierung der hierarchischen ON-Klasse überschrieben werden. Die Klasse der überschreibenden ON-Instanz muß eine Unterklasse der zu überschreibenden Instanz sein.

● Nachrichtenverbindungen - ML:

Nachrichtenverbindungen (*Message Link*s - ML) stellen die Kommunikationsbeziehung zwischen zwei komplementären, wertverträglichen Ports dar. Sie sind Ausdruck des logischen Kanals, über den Nachrichten (Steuer- oder Dateninformationen) von einer ON-Instanz zu einer anderen übertragen werden. Ein ML wird durch eine durchgezogene Linie, die die kommunizierenden Ports verbindet, dargestellt (vgl. Abbildung 3.4). Für die Erstellung von Nachrichtenverbindungen gelten die folgenden Regeln:

- Es dürfen nur Sende- mit Empfangsport verbunden werden (ASP nur mit ARP und SSP nur mit SRP).

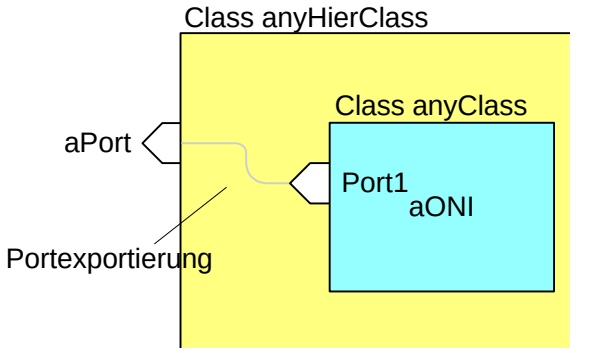
- Asynchrone Ports können beliebig viele Nachrichtenverbindungen besitzen. Synchrone Ports dürfen nur genau eine besitzen.
- Nachrichtenverbindungen müssen die Wertverträglichkeit der Ports berücksichtigen, so daß alle übertragenen Nachrichten auf der Empfängerseite ohne Informationsverlust verarbeitet werden können. Ein ARP der Datenklassse *int* darf z.B. auch mit ASP der Datenklasse *byte* verbunden werden (vgl. hierzu die Vererbungsregeln für Ports auf Seite 46ff).

Vererbung: Nachrichtenverbindungen werden unverändert an die jeweiligen Unterklassen vererbt. Sie können dort nicht geändert bzw. entfernt werden.

● Portexportierungen - PE:

Die internen Ports der ON-Instanzen (z.B. *aONI* in Abbildung 3.23) können an die externen Ports (z.B. *aPort*) der Container-Klasse „exportiert" werden. Durch diese Portexportierung (*Port Export* - PE) wird eine vormals gekapselte Schnittstelle (*Port1*) über das Interface der Container-Klasse für das Umfeld sichtbar. Jeder Port kann nur auf genau einen Container-Port exportiert werden. Eine PE wird ähnlich einer Portzuordnung dargestellt. Wenn interner und externer Port bezüglicher ihrer Eigenschaften nicht identisch sind, so gelten für die Erstellung einer PE die Vererbungsregeln der Ports (vgl. Seite 46ff). Die Anwendung dieser Regeln erfolgt in dem Sinne, daß der interne Port gegenüber dem externen Port die Rolle des konkretisierten Ports spielt. Dies garantiert die Wertverträglichkeit der beteiligten Ports, auch wenn eine spätere Konkretisierung der Container-Klasse durch vereinzeltes Überschreiben der ON-Instanzen erfolgt.

Abbildung 3.23
Exportierung von *Port1* auf *aPort*



Vererbung: Analog zu den Nachrichtenverbindungen werden Portexportierungn unveränderbar an die jeweiligen Unterklassen vererbt. Eine PE kann sich ebenso wie eine ML auf von der Oberklasse geerbte Ports beziehen.

3.4 Plattformabstraktion für Objektnetze

In Kapitel 3.1 erfolgte die Einordnung der Objektnetz-Methodik. Es wurden dort bereits die Gründe für eine abstrakte Darstellung der Funktionalität der Zielplattform genannt; der Entwurf von Objektnetz-Steuerungsspezifikationen sollte grundsätzlich unabhängig von schaltungstechnischen Details erfolgen. Die eigentliche Umsetzung der abstrahierten Plattformspezifika erfolgt auf einer getrennten Entwurfsebene. Zur Unterstützung dieser hardwarenahen Ebene definiert die Objektnetz-Methodik ein spezielles Framework (*Platform Abstraction Framework*

– PAF) [Richter 97], [RiNüFe 97]. Es unterstützt den Designer sowohl bei der Kapselung von verteilten Hardware-Plattformen als auch bei der automatischen Implementierung von Objektnetzen auf diesen so gekapselten Plattformen. Es wurde ein objektbasierter Ansatz gewählt, welcher sich am Konzept von [Zöller 91] orientiert.

Abbildung 3.24
AS-Klassen als Verbindung von Objektnetz-Spezifikation und verteilter Zielplattform

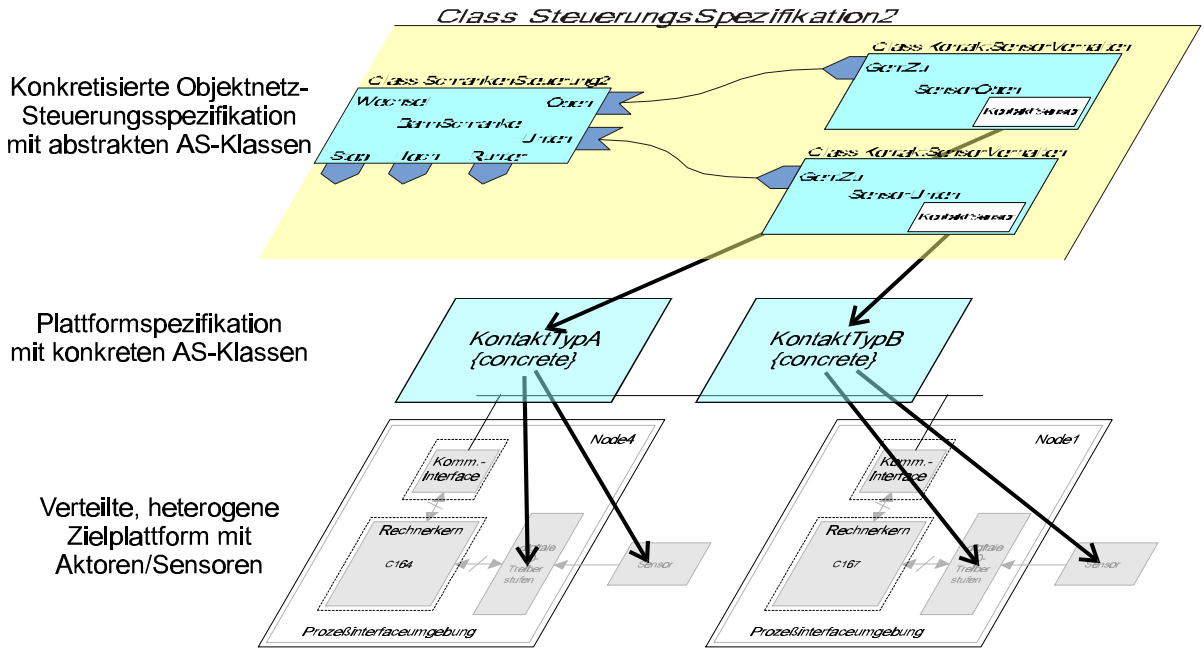
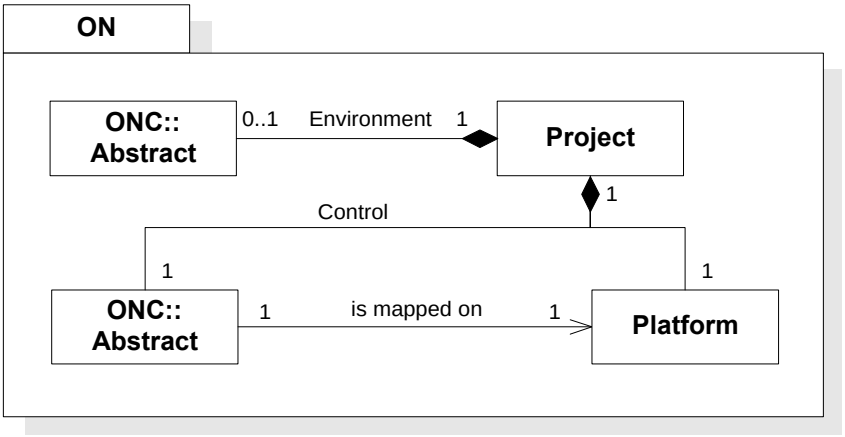


Abbildung 3.24 zeigt das Prinzip, wie eine konkretisierte Objektnetz-Spezifikation, die abstrakte Aktor/Sensor-Klassen referenziert, mit einer verteilten Plattform verbunden wird. Die Verbindung erfolgt hierbei über die abstrakten AS-Klassen bzw. deren konkrete Unterklassen.

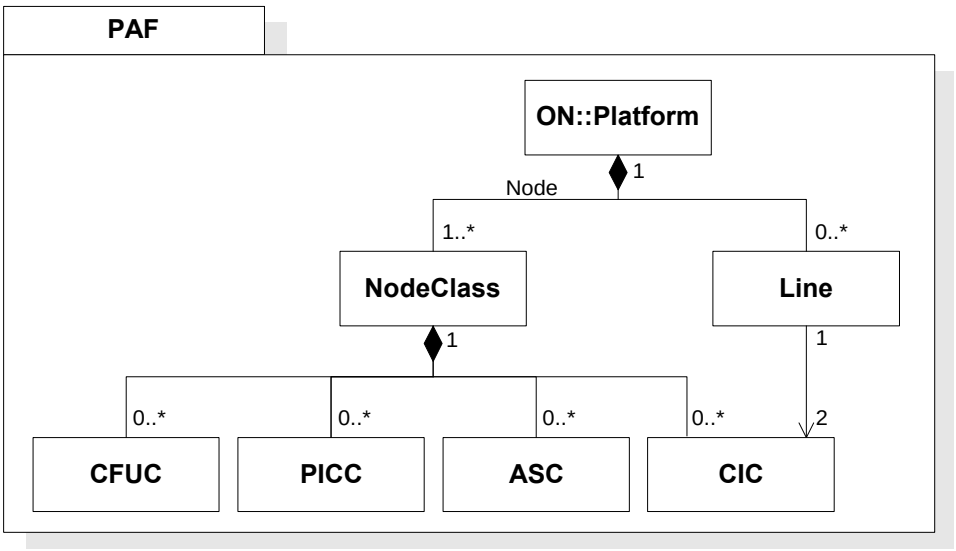
Abbildung 3.25
Einbindung der Steuerungs- und Umgebungsspezikation in eine Projektspezifikation



In Kapitel 3.3.3 wurden bereits die abstrakten AS-Klassen eingeführt. Ihre konkreten Unterklassen (d.h. die Objekte davon) können einer realen Plattform zugeordnet werden. Die gemeinsame Methodenschnittstelle von abstrakten und konkreten AS-Klassen bildet das Bindeglied zwischen hardwareunabhängigen Objektnetz und realer Zielplattform.

Die Spezifikation der Funktionalität einer verteilten Plattform ist neben den Objektnetz-Spezifikationen für Steuerung (*Control*) und Umgebung (*Environment*) fester Bestandteil eines Entwurfsprojektes (vgl. Abbildung 3.25). Neben den Informationen zur automatischen Er-zeugung der Aktor-/Sensor-Ansteuersoftware enthält die Plattformspezifikation Informationen über das Kommunikationsnetzwerk, die unter anderem die Grundlage für die Zuordnung (*mapping*) der Steuerung auf die verteilte Zielplattform bildet.

Abbildung 3.26
Die Metaklassen des Frameworks zur Plattformabstraktion (PAF)



Um die Komplexität unterschiedlichster verteilter Rechnerplattformen zu beherrschen, erfolgt der Spezifikationsvorgang getrennt nach Rechnerknoten- und Rechnernetzebene. Rechnerknoten werden durch Objekte (*Node*) der Knotenklasse (*NodeClass*) beschrieben. Eine Knotenklasse setzt sich aus Objekten zur Klassifikation der Rechnerkernfunktionalitäten (*Core Function Unit Class* – CFUC), der Prozeßinterfaceumgebung (*Process Interface Coupler Class* – PICC), der Kommunikationsschnittstellen (*Communication Interface Class* – CIC), sowie den Aktoren und Sensoren (*Actuator/Sensor Class* – ASC) zusammen (vgl Abbildung 3.26). Mit *Line*-Objekten, die Punkt-zu-Punkt-Verbindungen repräsentieren, können beliebige logische Netztopologien (z.B. Bus oder Ring) eines eingebetteten Systems modelliert werden.

3.4.1 Spezifikation der Rechnerknoten

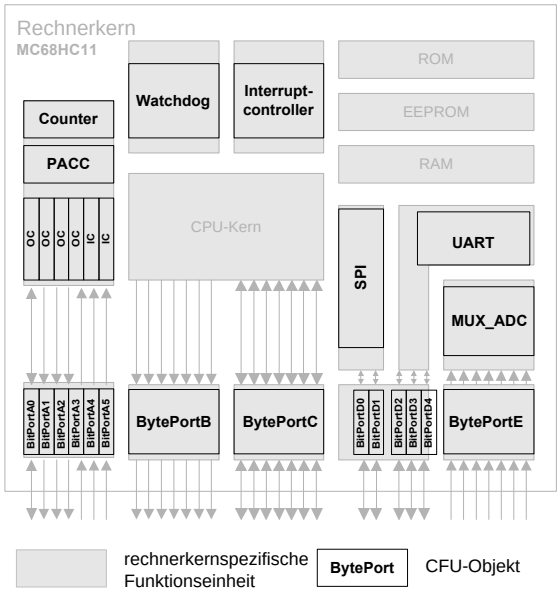
Die Klassifikation der Rechnerknoten aus Kapitel 2.2.1 bildet die Grundlage für die Struktur der Plattformspezifikationen. Sie erfolgt dabei im Rahmen eines *Bottom-Up*-Vorgehens. Von einer

konkreten Rechnerknoten-Hardware wird schrittweise abstrahiert, indem zuerst die Rechnerkernfunktionseinheiten, danach die Prozeßinterfaceumgebung und darauf aufbauend die Aktoren, Sensoren und Kommunikationsschnittstellen klassifiziert werden (vgl. Kapitel 2.2).

3.4.1.1 Kapselung der Rechnerkernfunktionseinheiten

Ziel der CFU-Klassen (*Core Function Unit Class* – CFUC) ist es, die verschiedenen Funktionseinheiten eines Rechnerkerns zu abstrahieren, ihre rechnerkernspezifische Ansteuerung zu kapseln und einen konsistenten Zugriff auf möglicherweise überlagerte Funktionseinheiten zu gewährleisten. Abbildung 3.27 zeigt exemplarisch, wie CFU-Objekte die Funktionseinheiten des Motorola 8-Bit-Controllers MC68HC11 [MOTOROLA 98] spezifizieren.

Abbildung 3.27
Klassifikation der Funktionseinheiten des Controllers MC68HC11 durch CFU-Objekte



Gegenstand der Spezifikation sind nur die Funktionseinheiten zur Zeit-, Interrupt, Kommunikations- und E/A-Steuerung. Die zentrale Verarbeitungseinheit, Speicher sowie mathematische Coprozessoren werden nicht betrachtet, da sie nicht explizit für die Modellierung der Aktor-/Sensoransteuerung bzw. der Kommunikationsschnittstellen verwendet werden. Die Verwaltung der Hardware-Details dieser Module (z.B. die Speicherverwaltung) erfolgt bereits durch den rechnerkernspezifischen Hochsprachencompiler, der für die Implementierung der Objektnetz-Spezifikation eingesetzt wird.

Die folgenden Definitionen bilden den Rahmen für die Erstellung von CFU-Klassen:

- Ein **Hardware-Modul** (z.B. Watchdog, Timer, Ports) ist eine Schaltungsstruktur eines Controllers, die eine oder mehrere Hardware-Funktionen (z.B. Zählen oder binäre Ein-/Ausgabe) realisiert. Typisch für Controller-Hardware-Module ist ihre Programmierung, über in den Speicher- oder E/A-Adreßraum eingeblendete Registerzellen. Schaltungskomponenten eines Rechnerknotens, die nicht zum Controller gehören, aber mit diesem busgekoppelt sind und auf gleiche Art und Weise programmiert werden, werden ebenfalls dem Rechnerkern als Hardware-Modul zugeordnet.

- Ein Hardware-Modul kann ein oder mehrere **Hardware-Funktionen** bereitstellen, die sich durch eine unterschiedliche Programmierung des Moduls aktivieren lassen. Das Hardware-Modul Timer kann z.B. zur Impulszählung oder Taktteilung herangezogen werden.

- Eine **elementare Funktionseinheit** abstrahiert die zur Realisierung einer Hardware-Funktion konfigurierte programmierbare Hardware-Struktur. Sie ist gekennzeichnet durch diese Hardware-Funktion. Mehrere elementare Funktionseinheiten können zu einer **komplexen Funktionseinheit** zusammengefaßt werden, deren Hardware-Funktion sich dann aus den Hardware-Funktionen der elementaren Funktionseinheiten zusammensetzt.

- Eine **CFU-Klasse** schließlich kapselt die Ansteuerungsdetails für eine elementare oder komplexe Funktionseinheit des Rechnerkerns. Sie besitzt ein controllerunabhängiges Interface. Über ihre Methoden kann der konsistente Zugriff auf eine oder mehrere elementare Funktionseinheiten erfolgen.

Abbildung 3.28
Ausschnitt aus einer exemplarischen CFU-Klassenhierarchie

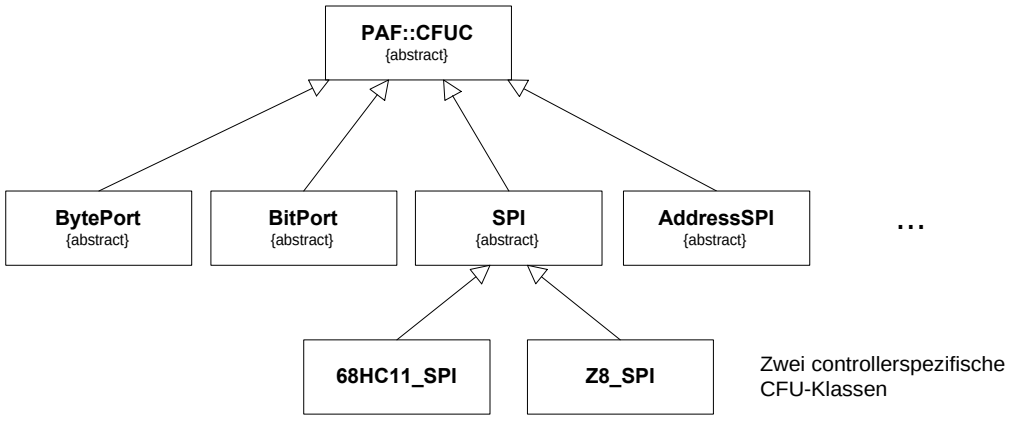


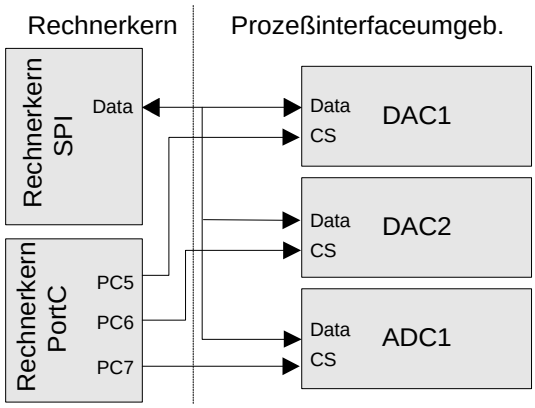
Abbildung 3.28 zeigt einen Ausschnitt aus einer CFU-Klassenhierarchie. Von einer abstrakten Oberklasse (*PAF::CFUC*) sind spezialisierte abstrakte Unterklassen abgeleitet. Durch die Analyse einer Vielzahl von Mikrocontrollern [Schossig 93] wurden diese abstrakten Unterklassen ermittelt. Sie bilden die Basis für das PAF. Der rechnerkernunabhängige Entwurf der PIC-Klassen baut auf diesen Klassen auf. Mehrfachvererbung wird in diesem Ansatz aufgrund der notwendigen komplexen Mechanismen zur Konsistenzsicherung nicht zugelassen. Bei der hier betrachteten Einfachvererbung übernimmt die erbende Klasse das Interface der abstrakten Oberklasse. In den konkreten Unterklassen erfolgt die spezifische Methoden-implementierung. Die beiden controllertypspezifischen Konkretisierungen eines Seriell-Parallel-Wandlers (*68HC11_SPI* und *Z8_SPI*) zeigen beispielhaft, wie eine abstrakte Klasse (*SPI*) je nach Controllertyp unterschiedliche Unterklassen erhalten muß.

Die konkreten CFU-Klassen sind speziell einer oder mehreren physikalisch vorhandenen Funktionseinheiten eines Rechnerkerns zugeordnet, ihre Methodenimplementierungen sind **rechnerkernabhängig**. Diese Klassen können nur einmal instantiiert werden. Die entsprechenden CFU-Objekte sind immer den gleichen Funktionseinheiten zugeordnet. Jeder Rechnerkern besitzt seine spezifischen CFU-Klassen.

3.4.1.2 Definition von CFU-Klassen

Moderne Controller sind durch ein optimiertes Platz-Leistungsverhältnis ausgezeichnet. Dies hat zur Folge, daß sich z. T. mehrere elementare Funktionseinheiten ein gemeinsames Hardware-Modul teilen. Dadurch kommt es zu Überlagerungen. Der Hardware-Zugriff muß im gegenseitigen Ausschluß durch diese Funktionseinheiten erfolgen. Das Beispiel in Abbildung 3.29, welches eine DAC-/ADC-Steuerung über ein SPI (*Serial Parallel Interface*) zeigt, soll dies verdeutlichen. Wegen der gemeinsamen SPI-Nutzung, muß die Ansteuerung der D/A-Wandler (DAC) und des A/D-Wandlers (ADC) im gegenseitigen Ausschluß erfolgen. Aufgrund dieser Verflechtung wird die Anordnung aus SPI und PortC als komplexe Funktionseinheit durch eine eigene CFU-Klasse (*AddressSPI*) gekapselt. Die IO-Pins PC5-PC7 fungieren dabei als Adreßeinheit.

Abbildung 3.29
DAC-/ADC-Steuerung über ein SPI



Eine genaue Beschreibung der konsistenten Verwaltung von Verflechtungen mehrerer Funktionseinheiten und der sich dabei gegenseitig ausschließenden Zugriffe auf ein CFU-Objekt (z.B. durch mehrerer PIC-Objekte) ist in [Richter 97] auf den Seiten 42-45 zu finden. Die Notation des Methodencodes erfolgt in der Regel mit dem rechnerkernspezifischen Assembler. CFU-Klassen ermöglichen somit alleinig innerhalb des PAF die direkte Ansteuerung der Hardware.

3.4.1.3 Kapselung der Prozeßschnittstellenumgebung

Ziel der PIC-Klassen (*Process Interface Coupler Class* – PICC) ist es, die Prozeßschnittstellen eines Rechnerknotens zu abstrahieren und die schaltkreisspezifische Ansteuerung der der dort enthaltenen Prozeßschnittstellenkoppelelemente (*Process Interface Coupler* - PIC) zu kapseln. Folgende Definitionen verdeutlichen das Konzept der PIC-Klassen:

- Die **Prozeßschnittstellenumgebung** eines Rechnerknotens besteht aus einer oder mehreren Prozeßschnittstellen.
- Eine **Prozeßschnittstelle** umfaßt alle rechnerkernexternen Schaltungskomponenten zur Abbildung der vom Rechnerkern ausgehenden Signale auf die Eingangsgrößen eines Aktors bzw. der Ausgangssignale einer Sensoreinheit auf die Eingänge des Rechnerkerns. Über eine Prozeßschnittstelle können ein oder mehrere Aktoren bzw. Sensoren angesteuert werden.
- Die Schaltungskomponenten zur Signalwandlung, -weiterleitung oder -anpassung werden als **Prozeßschnittstellenelemente** bezeichnet. Prozeßschnittstellenelemente, die direkt an die Rechnerkern-E/A-Einheiten gekoppelt sind, werden als **Prozeßschnittstellen-koppelelemente** (PIC) bezeichnet.

- Gleichartige PIC werden entsprechend ihrer Funktionalität zu **PIC-Klassen** zusammengefaßt. Die PIC-spezifische Ansteuerung ist in der Methodenimplementierung gekapselt.

Abbildung 3.30
Schema einer komplexen Prozeßschnittstelle

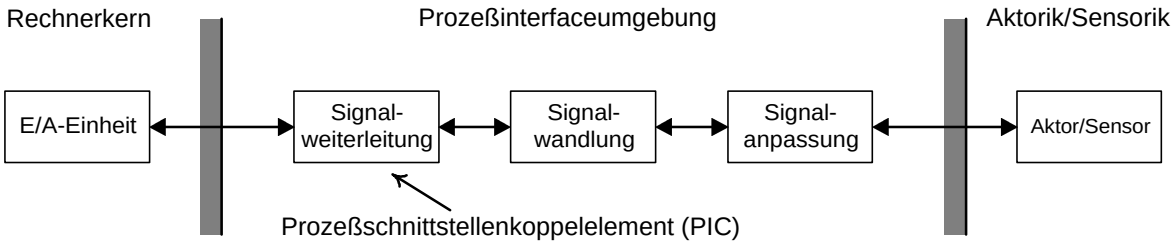
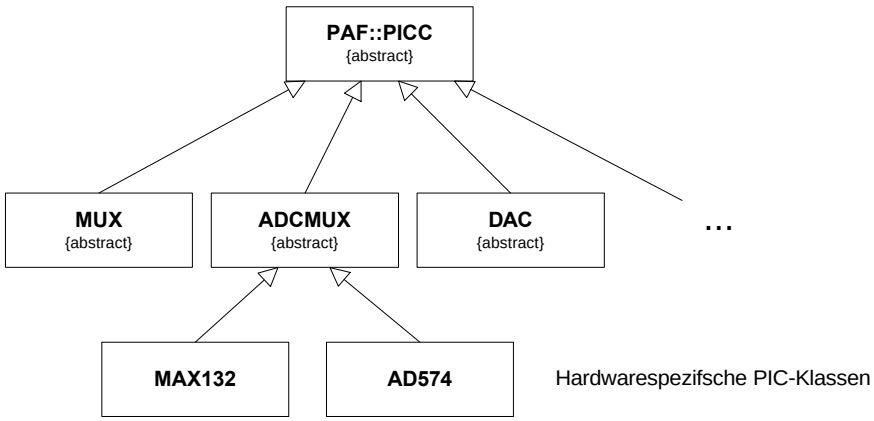


Abbildung 3.30 zeigt schematisch die Struktur einer komplexen Prozeßschnittstelle. Die Existenz und die Anordnung der Prozeßschnittstellenelemente in der Prozeßschnittstellenumgebung ist abhängig vom eingesetzten Rechnerkern und der verwendeten Aktorik bzw. Sensorik. So sind Rechnerkerne mit integrierten Signalwandlungs- und Signalweiterleitungselemente denkbar, oder Sensoren bzw. Aktoren mit integrierten Anpassungselementen. Für den Entwurf der Ansteuerungssoftware einer Prozeßschnittstelle sind nur die PIC interessant. Sie können in programmierbare und nichtprogrammierbare PIC unterteilt werden. Programmierbare besitzen Registerspeicherzellen bzw. Adreßeinheiten. Signalwandler oder Signalweiterleitungselemente, wie z. B. DAC, ADC oder MUX sind typisch für diese Kategorie. Typisch für nichtprogrammierbare PIC sind Signalanpaßelemente, wie z. B. Verstärkerstufen. Für den Entwurf der Ansteuerungssoftware sind nur die programmierbaren PIC relevant, da nur über sie die Manipulation des Signalflusses in der Prozeßinterfaceumgebung erfolgen kann.

Abbildung 3.31
Ausschnitt aus einer exemplarischen PIC-Klassenhierarchie



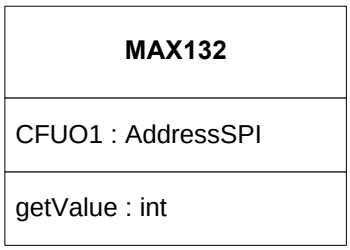
Wie in [Zöller91] ausgeführt, können die PIC entsprechend ihrer Funktionalität in fundamentale Kategorien eingeordnet werden. Das PAF stellt eine Auswahl abstrakter PIC-Klassen zur Verfügung; sie repräsentieren diese fundamentalen Kategorien.

Analog zu den CFU-Klassen bilden die PIC-Klassen eine zweistufige Klassenhierarchie (vgl. Abbildung 3.31). Von einer abstrakten Oberklasse (*PAF::PICC*) leiten sich die PIC-Klassen ab, die die fundamentalen Kategorien des PAF repräsentieren. Sie sind ebenfalls abstrakt, d. h. sie besitzen keine Methodenimplementierung. Um eine Aktor/Sensor-Klasse bzw. eine CI-Klasse unabhängig von einer konkreten PIC-Hardware-Struktur definieren zu können, werden diese abstrakten PIC-Klassen verwendet. Klassen, wie z. B. *MAX132* [MAXIM 98], sind konkret und schaltkreisbezogen. Sie besitzen die Methodenschnittstelle der abstrakten Oberklasse mit einer schaltkreisspezifischen aber rechnerkernunabhängigen Methoden-implementierung. Die Mehrfachvererbung ist analog zu den CFU-Klassen ausgeschlossen.

3.4.1.4 Definition von PIC-Klassen

Abbildung 3.29 zeigt die bereits erläuterte Ansteuerung zweier D/A-Wandler und eines A/D-Wandlers über ein SPI. Für die beiden D/A-Wandler (*DAC1* und *DAC2*) und den A/D-Wandler (*ADC1*) wird im Beispiel jeweils ein PIC-Objekt eingesetzt. Für das PIC-Objekt *ADC1* wird die konkrete schaltkreisbezogene PIC-Klasse *MAX132* verwendet. Am Analogeingang des 18-Bit-A/D-Wandlers MAX132 des Halbleiter-herstellers Maxim [MAXIM 98] ist ein Temperatursensor angeschlossen (vgl. Abbildung 2.4). Die PIC-Klasse ermöglicht es, die konkrete Aktor/Sensor-Klasse für diesen Temperatursensor rechnerkernunabhängig zu entwerfen.

Abbildung 3.32
Definition einer konkreten PIC-Klasse



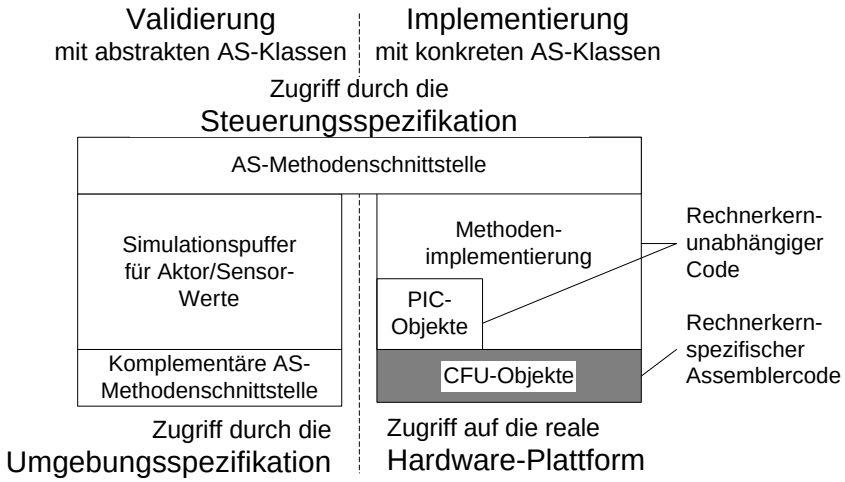
Die Abbildung 3.32 zeigt die Klasse *MAX132*. In der Klassenspezifikation ist die Referenz (*CFUO1*) auf das genutzte CFU-Objekt enthalten. Die referenzierten CFU-Klassen sind immer abstrakt. Erst bei der Generierung eines PIC-Objektes im Rahmen der Definition einer Rechnerknotenklasse (*NodeClass*) wird der CFU-Objektreferenz ein konkretes CFU-Objekt zugeordnet. Die Notation des Methodencodes (hier für *getValue*) erfolgt im Unterschied zu den CFU-Klassen rechnerkernunabhängig, z.B. in einer Hochsprache, unter Benutzung der Methoden des CFU-Objekts.

Die Behandlung von Mehrfachzugriffen durch AS- oder CI-Objekte wird analog zu den CFU-Objekten realisiert. Eine genaue Beschreibung dieser Behandlung ist in [Richter 97] auf den Seiten 50-53 zu finden.

3.4.1.5 Kapselung von Aktoren und Sensoren

Aktoren und Sensoren lassen sich nach ihrer Funktion in Kategorien, den abstrakten AS-Klassen (ASC), einteilen. Aktoren bzw. Sensoren einer Kategorie unterscheiden sich in den spezifischen Schaltungsinterfaces und der Charakteristik der elektrischen Ein- bzw. Ausgabegröße. Letztere müssen für die Kopplung an den Rechnerkern unterschiedlich stark aufbereitet werden. Dementsprechend unterschiedlich mächtig sind die Prozeßschnittstellen ausgelegt. Im Sonderfall kann ein Sensor bzw. Aktor (z.B. ein binärer Kontaktsensor) direkt an den Rechnerkern gekoppelt werden, wenn in ihm die Transformations- und Anpaßstufen integriert sind.

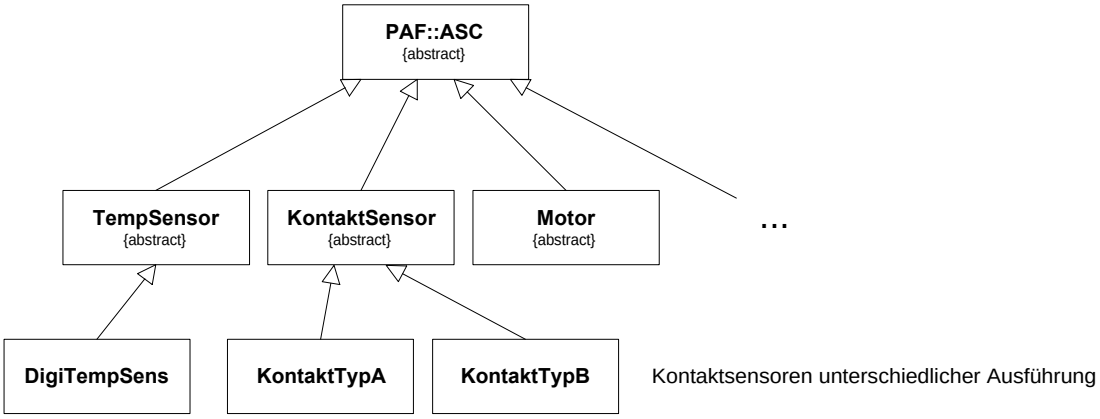
Abbildung 3.33
Abstrakte AS-Klassen für die Validierung und konkrete AS-Klassen für die Implementierung



Ziel der bereits vorgestellten abstrakten AS-Klassen ist es, die durch die Steuerung angesprochenen Sensoren und Aktoren zu abstrahieren. Durch die vollständige Abstraktion von plattformspezifischen Realisierungsdetails ist es möglich, im Rahmen der Validierung Steuerungs- und Umgebungsspezifikation ohne Kenntnis der Zielplattform zu koppeln. Für die Implementierung der Steuerung werden diese abstrakten Klassen schließlich durch konkrete Klassen ersetzt. Konkrete AS-Klassen werden durch Vererbung aus den abstrakten abgeleitet. Abbildung 3.33 zeigt diese unterschiedlichen Interpretationen von abstrakten und konkreten AS-Klassen.

Das PAF ermöglicht nun den controller- (Rechnerkern) und schaltkreisunabhängigen (Prozeßinterfaceumgebung) Entwurf konkreter AS-Klassen. Die auf der abstrakten Ebene noch leeren Methoden werden dabei durch Code in einer plattformunabhängigen Hochsprache gefüllt. AS-Klassen stellen die oberste Ebene des PAF dar. Die Methoden der konkreten AS-Klassen greifen auf die Funktionalität der abstrakten CFU- und PIC-Klassen zurück.

Abbildung 3.34
Ausschnitt aus einer exemplarischen AS-Klassenhierarchie

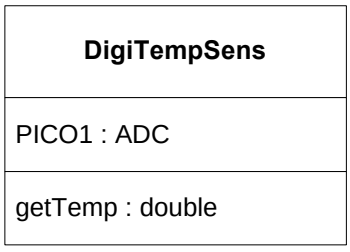


Der AS-Klassenbaum (Abbildung 3.34 zeigt einen exemplarischen Ausschnitt) besitzt die bekannte zweistufige Grundstruktur. Konkrete AS-Klassen (z.B. *KontaktTypA* und *KontaktTypB*) beschreiben Aktor/Sensoreinheiten mit spezifischer Ansteuerungsstruktur. Sie unterscheiden sich in der Regel durch ihre typ- bzw. herstellerabhängige Ausführung.

3.4.1.6 Definition konkreter AS-Klassen

Die Definition von konkreten AS-Klassen erfolgt weitgehend analog zu den PIC-Klassen. Als Unterschied ist die Möglichkeit der zusätzlichen Verwendung von PIC-Objekten zu nennen. Hierbei sind die referenzierten PIC-Klassen ebenfalls immer abstrakt. Erst bei der Erstellung eines AS-Objekts wird der PICO-Referenz ein konkretes PIC-Objekt (z.B. *MAX132* für *ADC*) zugeordnet. Abbildung 3.35 zeigt als Beispiel den Temperatursensor *DigiTempSens* aus Abbildung 3.34. Die Methode *getTemp*,die einen gültigen Temperaturwert liefert, nutzt die PIC-Methode *getValue*, um an den 18-Bit-Digitalwert des A/D-Wandlers zu gelangen.

Abbildung 3.35
Definition einer konkreten AS-Klassen



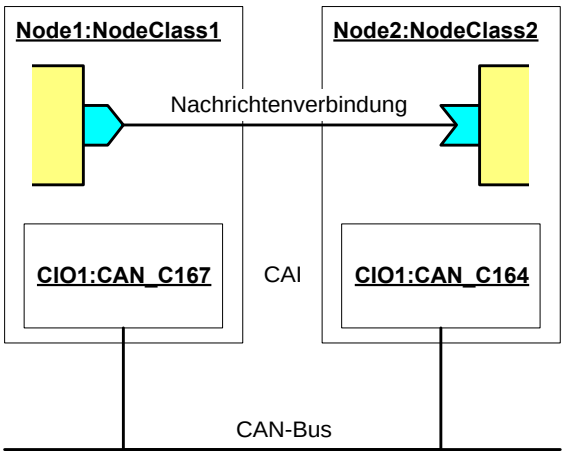
Durch die exklusive Verwendung eines AS-Objekts durch eine elementare ON-Instanz sind Zugriffskonflikte ausgeschlossen.

3.4.1.7 Kommunikationsschnittstellen

Rechnerknoten innerhalb eines verteilten Systems besitzen mindestens eine Kommunikationsschnittstelle. Durch spezielle CI-Klassen (*Communication Interface* - CI) wird die Funktionalität dieser Schnittstellen gekapselt. Das einheitliche Interface dieser Klassen

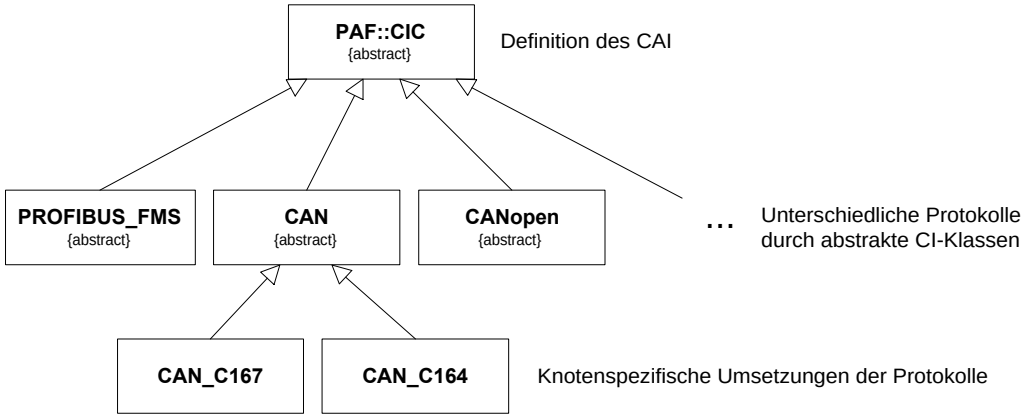
abstrahiert die unterschiedlichen Spezifika verschiedenster konkreter Schnittstellen bzw. Protokolle, und realisiert somit eine abstrakte Kommunikationsebene *(Communication Abstraction Interface* – CAI) [NütFen 95b]. Dieses Interface ermöglicht eine plattformunabhängige Umsetzung auch von Nachrichtenverbindungen, die zwei ON-Instanzen auf zwei getrennten Rechnerknoten verbinden (vgl. Abbildung 3.36). Bei der Modellierung einer Objektnetz-Spezifikation treten die CI-Klassen allerdings nicht in Erscheinung. Sie dienen primär der Implementierung.

Abbildung 3.36
Umsetzung von Nachrichtenverbindungen



Konkrete CI-Klassen sind vom Aufbau (Zugriff auf CFU- und PIC-Objekte) mit den konkreten AS-Klassen vergleichbar. Die zweistufige Klassenhierarchie der CI-Klassen ist allerdings im Vergleich zu den AS-Klassen geringfügig unterschiedlich aufgebaut. Bereits die abstrakte CI-Metaklasse (*PAF::CIC*) definiert Methodenschnittstellen, die die Charakteristik des CAI bestimmen. Die Zusammenstellung der Methoden entspricht der OSEK-COM-Spezifikation v2.0a [OSEK 97]. Abbildung 3.37 zeigt, wie sich von dieser Oberklasse weitere abstrakte CI-Klassen ableiten. Diese abstrakten Unterklassen verkörpern Kategorien von Schnittstellen, die das gleiche Protokoll benutzen. Zwei konkrete Schnittstellen können folglich nur dann miteinander gekoppelt werden, wenn sie eine gemeinsame Oberklasse besitzen. Die konkreten Unterklassen, die diese konkreten Schnittstellen realisieren, enthalten schließlich die spezifische Methodenimplementierung.

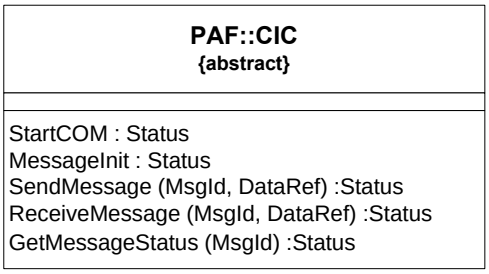
Abbildung 3.37
Ausschnitt aus einer exemplarischen CI-Klassenhierarchie



Die parameterlose *StartCOM* Methode wird zur Initialisierung des CI-Objekts und zum Starten der Kommunikation gerufen. *StartCOM* muß die zweite parameterlose Methode *MessageInit*

rufen, in der die Empfangspuffer (*message objects*), die das CI-Objekt verwaltet, eingerichtet werden. Die Nachrichtenobjekte werden durch einen systemweit eindeutigen Bezeichner (*MsgId*) unterschieden; er wird von den Methoden *SendMessage*, *ReceiveMessage* und *GetMessageStatus* zur Adressierung benutzt (vgl. Abbildung 3.38). *SendMessage* sendet an ein entferntes Nachrichtenobjekt, das im ersten Parameter spezifiziert ist, eine Nachricht, die durch den zweiten Parameter referenziert wird. *ReceiveMessage* kopiert eine empfangene Nachricht eines im ersten Parameter spezifizierten Nachrichtenobjekts in einen durch den zweiten Parameter spezifizierten Puffer. Mit *GetMessageStatus* wird der Zustand eines Nachrichtenobjekts, welches durch den einzigen Parameter spezifiziert wird, abgefragt. Das Ergebnis dieser Abfrage informiert die Applikation darüber, ob eine empfangene Nachricht vorliegt. Weitere Vereinbarungen und Vorgaben werden, um den Freiheitsgrad nicht zu weit einzuengen, durch das PAF nicht vorgenommen.

Abbildung 3.38
Vollständige Definition der Meta-CI-Klasse



3.4.2 Spezifikation des Kommunikationsnetzwerkes

Nachdem alle Rechnerknotentypen durch zugehörige Rechnerknotenklassen (*NodeClass*) spezifiziert wurden, kann das verteilte System modelliert werden. Dabei wird für jeden Rechnerknoten (*Node*) ein Instanz seiner zugehörigen Rechnerknotenklasse gebildet. Bei diesem Vorgehen ist die Verwendung von abstrakten Rechnerknotenklassen für eine Validierung der ON-Spezifikation unter realen Zeitbedingungen jedoch bereits ausreichend. Abstrakte Knoten-klassen referenzieren nur abstrakte AS- und CI-Klassen. Konkrete Rechnerknotenklassen, die sich durch Vererbung aus abstrakten ableiten lassen, werden für die automatische Im-plementierung eingesetzt; sie referenzieren Objekte konkreter AS-, CI-, CFU- und PIC-Klassen.

Jeder Rechnerknoteninstanz werden zwei weitere Parameter zugeordnet; sie spezifizieren das zeitliche Verhalten des Knotens bei der Abarbeitung von ON-Instanzen:

- *Rechnerknotengeschwindigkeit (speedscale)*: Diese Maßzahl steht für die normierte Abarbeitungszeit eines Benchmark-Programms auf dem jeweiligen Rechnerknoten. Die Normierung erfolgt gegenüber einem virtuellen Rechnerknoten, auf dem das gleiche Programm eine Abarbeitungszeit von einer Sekunde erzielen würde. Die Güte dieser Maßzahl steht und fällt mit dem Aufbau des Benchmark-Programms, welches eine repräsenta-tive Auswahl von Operationen enthalten sollte [HenPat 93]. Für den Validierungsvorgang werden die normierten Bearbeitungszeiten der Aktionen, die einem speziellen Rechner-knoten zugeordnet wurden, mit dieser Maßzahl des Rechnerknotens multipliziert.

- *Knotengrundtaktzeit (ticktime)*: Wartezeiten werden in der Regel durch Timer realisiert. Diese Timer besitzen eine kleinste Zeiteinheit, die Knotengrundtaktzeit. Folglich können Warte-, Kommunikationsbearbeitungs- und Taskwechselzeiten in der Objektnetz-Methodik nur Vielfache der Grundtaktzeit betragen.

Abbildung 3.39
Verteilte Plattform mit CAN-Bus-Netzwerk und vier Rechnerknoten

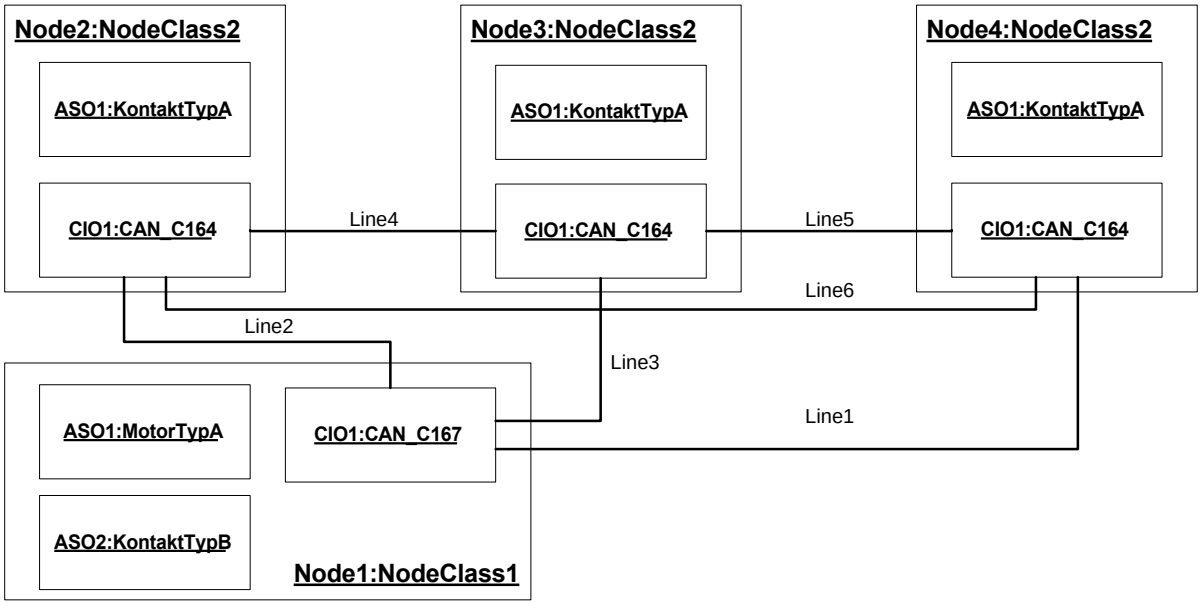


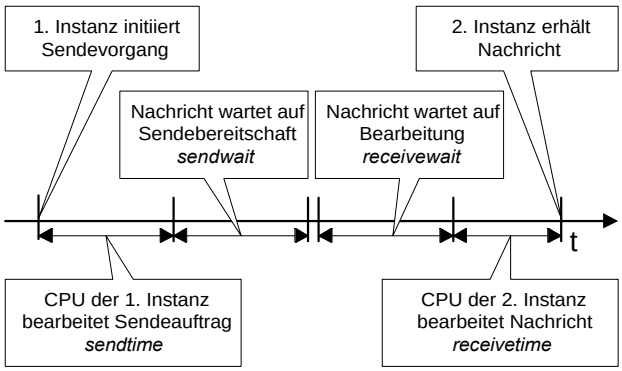
Abbildung 3.39 zeigt beispielhaft die Spezifikation einer Plattform (mit den konkreten Knotenklassen *NodeClass1* und *NodeClass2*), die für die Umsetzung der Steuerungs-spezifikation von Abbildung 3.4 eingesetzt werden kann. Die verteilten Rechnerknoten werden im allgemeinen Fall durch ein Kommunikationsnetzwerk verbunden. Die Topologie dieses Netzwerkes wird durch das PAF stark abstrahiert modelliert. Für jede Kommunikations-schnittstelle eines Rechnerknotens kann zu jeder anderen Schnittstelle der gleichen Kategorie (= gleiches Protokoll) eine Punkt-zu-Punkt-Kommunikationsverbindung (*Line*) definiert werden. Durch diese Punkt-zu-Punkt-Verbindung wird festgelegt, daß eine unmittelbare (kein Vermittlungsrechner dazwischen) physikalische Kommunikationsverbindung zwischen zwei Kommunikationsschnittstellen existiert. Sie ist die Voraussetzung für die Realisierung einer Objektnetz-Nachrichtenverbindung. Die Umsetzung von Nachrichtenverbindungen, die über mehrere in Reihe zu schaltende Kommunikationsverbindungen geführt werden müßten, wird aufgrund der Beschränkung auf Feldbusprotokolle, die keine Vermittlungsschicht besitzen, ausgeschlossen. Komplexe Bus- bzw. Ringstrukturen werden aus mehreren *Line*-Objekten zusammengesetzt. Für eine allgemeine Busstruktur mit n Teilnehmern (bzw. n Kommunikationsschnittstellen) müssen $(n^2-n)/2$ Verbindungen (sechs in Abbildung 3.39) definiert werden. Diese Zahl verringert sich, wenn einige Knoten keine Master-Funktionalität besitzen. Slave-Knoten, die oft mit verminderten Ressourcen auskommen müssen, können untereinander nicht direkt kommunizieren.

Rechnerknoten, die eine Kommunikationsschnittstelle besitzen, erhalten für jede Schnittstelle weitere Parameter zugeordnet; sie definieren den Zeitbedarf für das Senden bzw. Empfangen einer Nachricht in Abhängigkeit von deren Datenklasse der zu übertragenden Nachricht. Durch die Angabe dieser Maximalwerte kann die Validierung unter Berücksichtigung des realen Zeitverhaltens der Kommunikation erfolgen:

- Sendezeit (*sendtime*): Die Sendezeit einer Kommunikationschnittstelle ist die Maximalzeit, die für das Senden einer Nachricht von der CPU des Rechnerknotens aufzubringen ist. Die Wartezeit in einem separaten Kommunikationscontroller zählt nicht zur Sendezeit. Die Sendezeit kann im allgemeinen von der Gesamttopologie des Netzwerkes abhängig sein. Für jede Datenklasse existiert ein eigener Parameter.
- Sendewartezeit (*sendwait*): Die Sendewartezeit faßt die Zeiten für das Senden einer Nachricht zusammen, die nicht von der CPU-Zeit abgehen. Das Warten einer Nachricht auf den Buszugang im Schnittstellenbaustein kann z.B. dazu gerechnet werden.
- Empfangszeit (*receivetime*): Die Empfangszeit bildet das Gegenstück zur Sendezeit.
- Empfangswartezeit (*receivewait*): Die Empfangswartezeit bildet das Gegenstück zur Sendewartezeit.

Die Abbildung 3.40 verdeutlicht den Zusammenhang zwischen den einzelnen Zeitparametern. Die physikalische Übertragungszeit auf der Leitung wird dabei als vernachlässigbar angesetzt. Die Summe der vier dargestellten Parameter bildet die Zeit, um die eine asynchrone Nachrichtenübertragung über ein Kommunikationsmedium gegenüber einer lokalen Kommunikation auf einem Rechnerknoten länger dauert.

Abbildung 3.40
Zeitparameter der Kommunikation



Für die Ermittlung dieser Zeitparameter wird innerhalb der Objektnetz-Methodik kein eigenes Verfahren bereitgestellt. Für den PROFIBUS und ähnliche Protokolle kann auf [Li 93] verwiesen werden. In [Li 93] wird für die Spezifikation der Kommunikationsparameter alternativ die Matrixdarstellung verwendet. Für den komplizierteren Fall des CAN-Busses, bei dem höherpriore Nachrichten niederpriore Nachrichten verdrängen können, lassen sich aus [WaLuSt 92] und [Staub 95] ergänzende Informationen entnehmen.

3.5 Objektnetz-Plattform-Mapping

Vervollständigt wird eine Objektnetz-Projektspezifikation durch die Angabe einer Zuordnung (*mapping*) der elementaren Objektnetz-Instanzen und Aktor/Sensor-Objekte zu den Rechnerknoten (*Nodes*) der Zielplattform. Erst nach dieser Zuordnung kann eine automatische Implementierung bzw. eine Validierung mit realen Zeitparametern erfolgen. Die Objektnetz-Methodik definiert hierzu kein automatisches Verfahren, da das Finden des optimalen Mappings in der Regel nicht ohne das aufwendige Testen aller Varianten erfolgen kann. Durch heuristische Verfahren können allerdings brauchbare Näherungslösungen ermittelt werden. Ohne diese Verfahren erfolgt die Zuordnung manuell; unter Berücksichtigung einiger einfacher Regeln bzw. Einschränkungen:

- Elementare ON-Instanzen mit einer ASO-Referenz müssen auf dem Knoten zugeordnet werden, auf dem sich ein Objekt der referenzierten AS-Klasse befindet.
- Elementare ON-Instanzen ohne ASO-Referenz können beliebigen Knoten zugeordnet werden, solange gewährleistet ist, daß für alle Nachrichtenverbindungen zwischen ON-Instanzen, die auf getrennten Knoten lokalisiert wurden, eine Kommunikationsverbindung (*Line*) zwischen diesen Knoten existiert (kein Routing möglich).

Für die Plattformspezifikation von Abbildung 3.39 und die Steuerungsspezifikation von Abbildung 3.4 zeigt Tabelle 3.5 eine mögliche Mapping-Variante.

Tabelle 3.5
Beispiel für ein Objektnetz-Plattform-Mapping

	Node1		*Node2*	*Node3*	*Node4*
Instanz mit ASO:	*ASO1*	*ASO2*	*ASO1*	*ASO1*	*ASO1*
SensorLlinks			✗		
SensorRechts				✗	
SensorOben					✗
SensorUnten		✗			
SchrankenMotor	✗				
Instanz ohne ASO:	*Node1*		*Node2*	*Node3*	*Node4*
BahnSchranke. Abschnitt		✗			
BahnSchranke. Schranke		✗			

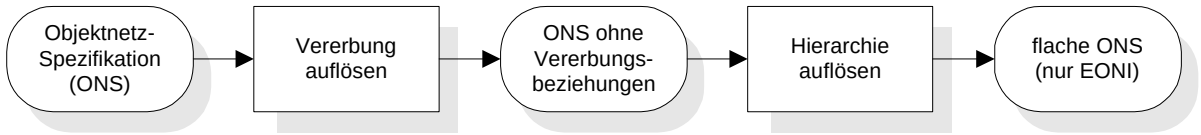
Die Zuordnung der Instanzen mit ASO-Referenzen unterliegt den Zwängen, die sich durch die physikalische Anordnung der Sensoren und Aktoren ergeben. Die beiden elementaren Instanzen ohne ASO-Referenz (durch die hierarchische Klasse *SchrankenSteuerung2* gekapselt) können aufgrund der allgemeinen Busstruktur beliebig zugeordnet werden.

4. Kapitel

Auflösen der Hierarchie- und Vererbungsbeziehungen

Bevor in Kapitel 5 die formale Verhaltensbeschreibung von Objektnetz-Spezifikationen mit höheren Petri-Netzen erfolgt, werden die in Kapitel 3 eingeführten Objektnetze von bestimmten Entwurfsstrukturen befreit. Die dabei aufgelösten Vererbungs- und Hierarchiekonstrukte dienen lediglich der Unterstützung des Designers. Ihre Entfernung verändert das Verhalten der Spezifikation nicht. Die Auflösung der Hierarchie- und Vererbungsbeziehungen erfolgt in zwei Schritten (vgl. Abbildung 4.1).

Abbildung 4.1
Auflösen von Vererbungs- und Hierarchiebeziehungen bei Objektnetz-Spezifikationen

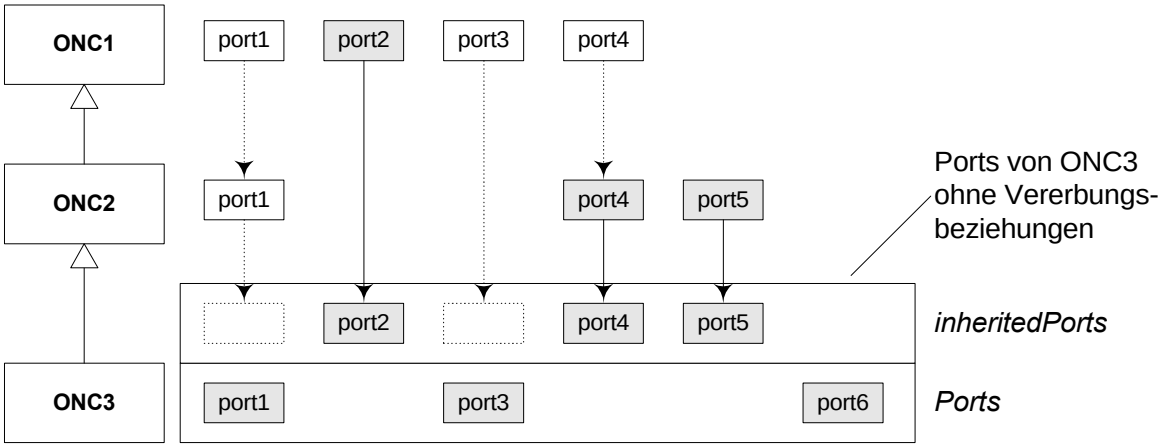


Im ersten Schritt werden sämtliche Vererbungsbeziehungen aufgelöst. Im zweiten Schritt erfolgt die Einebnung der hierarchischen Strukturen. Das Ergebnisse beider Schritte sind jeweils wieder gültige Objektnetz-Spezifikationen, die der Anwender innerhalb der Objektnetz-Methodik auch direkt notiert kann.

4.1 Auflösen der Vererbungsbeziehungen

Die Vererbungsbeziehungen zwischen Daten- Aktor/Sensor- und Objektnetzklassen dienen innerhalb der Objektnetz-Methodik der Entwurfsunterstützung. Eine vollständig konkretisierte Klasse ist in der Regel die Unterklasse einer allgemeineren Klasse. Am Beispiel der Ports zeigt Abbildung 4.2 das Prinzip der Vererbung und ihrer Auflösung, wie es von [Kahnert 98] bei der Implementierung der Klassenpersistenz umgesetzt wurde.

Abbildung 4.2
Auflösung der Vererbungsbeziehunng von *ONC3* am Beispiel der Ports [Kahnert 98]



In der Klasse *ONC1* werden *port1* bis *port4* definiert. Aus *ONC1* wird durch Vererbung die konkretere Klasse *ONC2*. Diese definiert die Ports *port1* und *port4* neu und überschreibt somit die beiden gleichnamigen Ports aus *ONC1*. Zusätzlich wird *port5* definiert. *ONC3* definiert *port1*, *port3* und *port6*, wobei *port1* und *port3* jeweils einen geerbten Port überschreiben.

Aus der Semantik der Vererbung, wie sie am Beispiel erläutert wurde, leitet sich das Verfahren zu ihrer Auflösung ab. Für eine spezielle Klasse werden ihre Vererbungsbeziehungen dadurch entfernt, daß von der allgemeinsten Klasse (der *root*-Klasse) bis hinab zur eigentlichen Klasse die vererbten Informationen akkumuliert werden. Aus der ersten Unterklasse (*ONC1*) von *root* werden die jeweiligen Elemente (z.B. Ports oder Zustände) in eine eigene Liste übernommen. Bei jeder weiteren Unterklasse (*ONC2* und *ONC3*) werden entweder neue Elemente in diese Liste eingefügt oder bereits vorhandene, gleichnamige Elemente durch neue Elemente überschrieben. Das Ergebnis dieser Operationen sind Klassendefinitionen, die keine Vererbungsbeziehungen mehr enthalten.

Die im Anhang beschriebene Objektnetz-Entwurfsunterstützung verwaltet alle vom Anwender erstellten ON-Spezifikationen in einer Entwurfsdatenbank, dem sogenannten *Inventory*. Der Anwender hat somit direkten Zugriff auf alle Klassendefinitionen. Die von [Kahnert 98] realisierte Schnittstelle zwischen dieser SQL-Datenbank und dem objektorientiert aufgebauten Objektnetz-Tool liefert zu jeder Klassendefinition nicht nur die Listen (*Ports*, *States* ...) der in der Klasse definierten Elemente, sondern noch weitere dynamisch ermittelte Listen (*inheritedPorts*, *inheritedStates* ...), die die aus den Oberklassen geerbten Elemente enthalten.

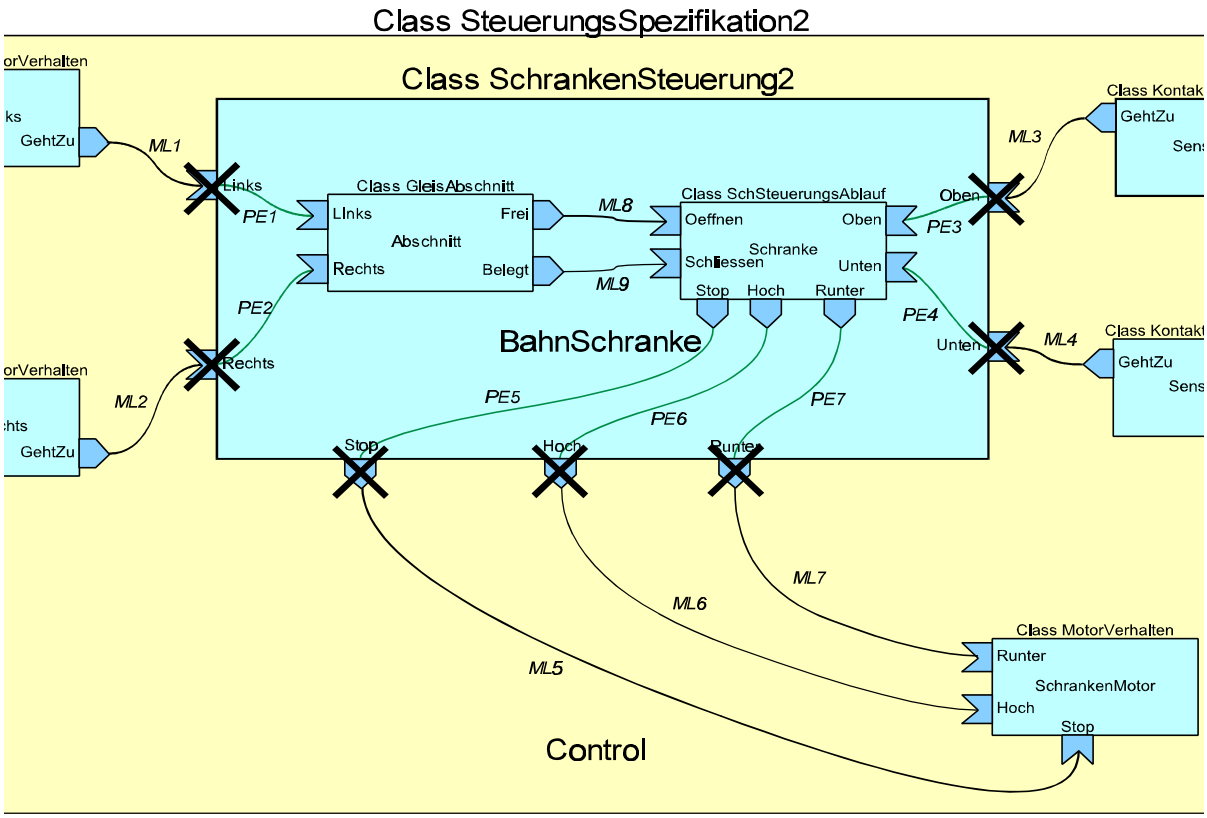
Die Vereinigung beider Listentypen (siehe Abbildung 4.2) liefert eine Klassen-definition, die nicht mehr auf bestehende Vererbungsbeziehungen angewiesen ist.

4.2 Einebnen hierarchischer Objektnetze

Ebenso wie das Ergebnis der ersten ist auch das Ergebnis der zweiten Transformationsstufe eine gültige ON-Spezifikation. Flache Objektnetze, ohne Vererbungsbeziehungen, lassen sich im Rahmen der Objektnetz-Methodik auch direkt erstellen. Dabei bedeutet allerdings der Verzicht auf Hierarchie und Vererbung eine starke Einschränkung an Modellierungsmächtigkeit.

4.2.1 Hierarchische ON-Instanzen

Abbildung 4.3
Die hierarchische Struktur der ON-Steuerungsspezifikation des Bahnschrankenprojektes

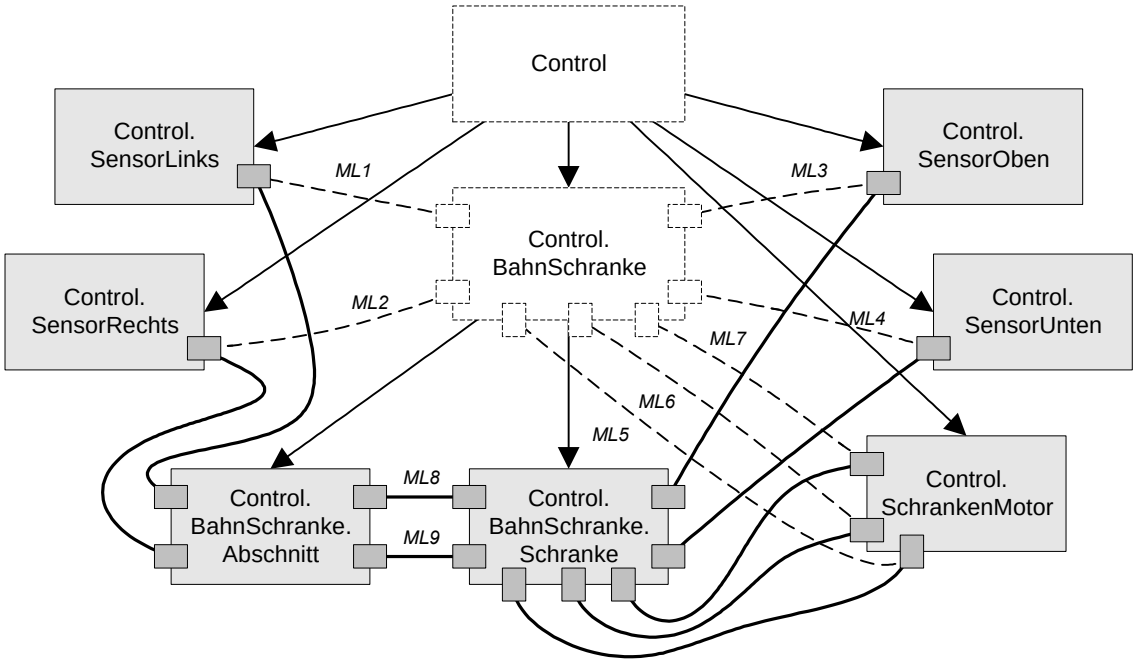


Während des Objektnetz-Entwurfs entstehen durch mehrstufige Komposition geschachtelte Klassendefinitionen. Dies geschieht z.B. beim *Bottom-Up*-Entwurf dadurch, daß elementare ON-Instanzen während des Entwurfs innerhalb eines Containers zusammengefaßt werden (Komposition). Von diesen so entstandenen neuen hierarchischen Klassendefinitionen (HONC), können ebenfalls ON-Instanzen gebildet werden, die wiederum in neue Container-Klassen eingebettet werden können.

Abbildung 4.3 erläutert am Beispiel der konkretisierten Bahnschrankensteuerung (vgl. Abbildung 3.4) die Umsetzung des in [Schmidt 98] realisierten Algorithmus‘ zur Dehierarchisierung. Abbildung 4.3 beschränkt sich dabei auf den Steuerungsteil der Spezifikation. Das Bahnschrankenprojekt enthält als ON-Steuerungsspezifikation eine Instanz (*Control*) der hierarchischen Klasse *SteuerungsSpezifikation2*. Ihre Oberklasse (*SteuerungsSpezifikation1* in Abbildung 3.4) enthält bereits die sechs Instanzen: *SensorLinks*, *SensorRechts*, *SensorOben*, *SensorUnten*, *SchrankenMotor* und *BahnSchranke*. In der Klasse *SteuerungsSpezifikation2* wurde die Instanz *BahnSchranke* durch die hierarchische Klasse *SchrankenSteuerung2* konkretisiert, die die beiden elementaren ON-Instanzen *Abschnitt* und *Schranke* enthält.

Für die Einebnung (Auflösung der Hierarchie) können die hierarchischen Beziehungen auch als baumartige Struktur mit den Instanzen als Knoten (sog. Instanzenbaum) dargestellt werden (vgl. Abbildung 4.4). Die Instanz *Control.BahnSchranke* ist ein Knoten, der die Wurzel eines Unterbaumes darstellt. Die grau unterlegten Instanzen der elementaren Klassen sind die Blätter des Baums; sie besitzen keine weiteren Söhne [Engesser 88]. Ziel der Einebnung ist es nun, die Blätter dieses Baums zu extrahieren und alle dabei nicht mehr benötigten Hierarchiekonstrukte, wie Portexportierungen (*PE1* bis *PE7* in Abbildung 4.3), aufzulösen.

Abbildung 4.4
Instanzenbaum mit alten (gestrichelt) und transformierten Nachrichtenverbindungen



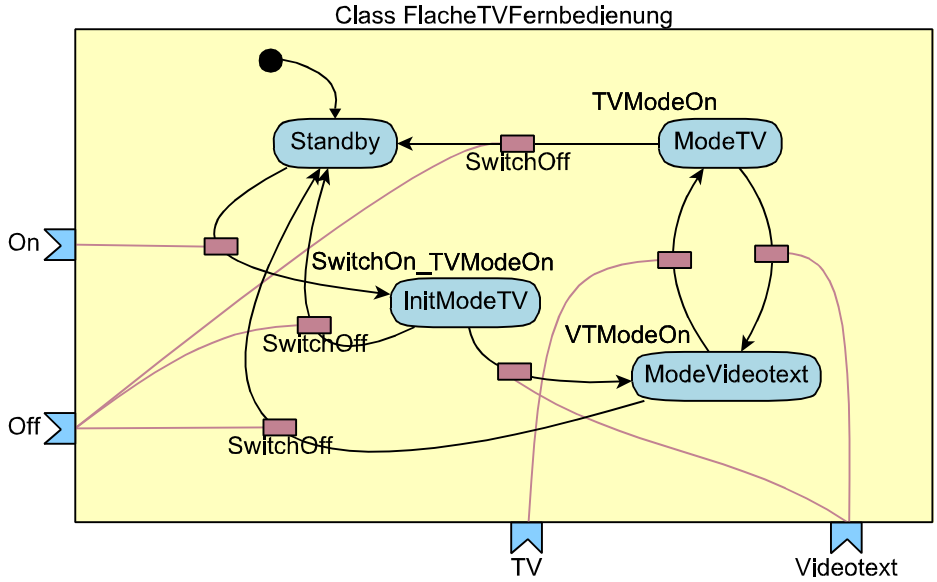
Die Extraktion der elementaren ON-Instanzen erfolgt rekursiv, bei der Wurzel des Baums beginnend. Alle Instanzen, die auf einer Hierarchieebene gefunden werden, werden daraufhin getestet, ob sie Instanzen einer hierarchischen Klasse sind. Ist dies der Fall, so geht die Rekursion eine Ebene tiefer. Falls nicht, so wurde ein Blatt des Baums erreicht. Die gefundene Instanz wird in das flache Objektnetz übernommen. Dabei wird der Instanznamen um den gesamten Hierarchiepfad erweitert, um die Instanz im flachen Netz eindeutig zu identifizieren. Aus

Abschnitt wird z.B. *Control.BahnSchranke.Abschnitt*. Während des Abstiegs im Instanzen-baum werden bei allen gefundenen Instanzen einer hierarchischen Klasse sämtliche Port-exportierungen (PE) aufgelöst. Dabei werden alle klassenexternen Nachrichtenverbindungen (*ML1* bis *ML7* in Abbildung 4.3) auf den Port (z.B. *ML1* auf *Links* von *Abschnitt*) geschoben, der durch die PE exportiert wurde. Neben der Auflösung der PE wird dabei ebenso der zugehörige Port (hier *Links* von *BahnSchranke*) des Containers gelöscht (durch ein Kreuz in Abbildung 4.3 angedeutet). Nach der Auflösung der Hierarchiebeziehungen finden sich nur noch elementare Instanzen, die durch Nachrichtenverbindungen verbunden sind. Im Gegensatz zur Entfernung der Vererbung reduziert die Einebnung die Informationsmenge, da Portexportierungen und zugehörige Containerports entfallen.

4.2.2 Hierarchische Zustände

Nach der Transformation hierarchischer ON-Instanzen enthält ein Objektnetz-Projekt ausschließlich elementare Instanzen, die über Nachrichtenverbindungen bzw. Aktor/Sensor-Objekte miteinander kommunizieren. Bevor die Verhaltensbeschreibung mit höheren Petri-Netzen erfolgt, werden die Zustandsmaschinen, die den Kern elementarer Klassen bilden, ebenfalls „verflacht". Abbildung 4.5 zeigt beispielhaft das Ergebnis für die Klasse *TVFernbedienung*.

Abbildung 4.5
Die verflachte elementare ON-Klasse *TVFernbedienung* aus Abbildung 3.21



Wurde ein Zustand der ON-Zustandsmaschine durch eine Referenz auf eine Unterzustandsmaschine hierarchisch konkretisiert, so wird dieser Oberzustand und die ihn referenzierenden Zustandswechsel transformiert. Die Transformation erfolgt in drei Schritten:

- Zu Beginn wird der hierarchisch konkretisierte Oberzustand durch seine Unterzustandsmaschine, die eigene Zustände, Zustandswechsel und Portzuordnungen enthält, ersetzt.

- Im zweiten Schritt wird der Startzustand der eingefügten Unterzustandsmaschine durch einen Nichtstartzustand ersetzt. Dieser erhält zusätzlich ein Duplikat (*InitModeTV* in Abbildung 4.5), welcher als Übergangszustand zur eingesetzten Unterzustandsmaschine dient. Im Unterschied zu *ModeTV* wurde der Eintrittsaktion von *InitModeTV* die Eintrittsaktion des ersetzten Oberzustand vorangestellt. Der Übergangszustand wird als Startzustand generiert, falls der ersetzte Oberzustand Startzustand ist. Falls der ersetzte Oberzustand Folgezustand eines Zustandswechsels war, so wird der neu generierte Zustand der neue Folgezustand dieser Zustandswechsel. Zustandswechsel in der Unterzustands-maschine, mit dem Startzustand der Unterzustandsmaschine als Vorzustand, werden verdoppelt. Beide Zustandswechsel unterschieden sich nur in ihrem Vorzustand. Der erste Zustandswechsel erhält den transformierten Startzustand der Unterzustandsmaschine als Vorzustand. Der zweite erhält den neu generierten Übergangszustand als Vorzustand.
- Der dritte Schritt transformiert die Zustandswechsel, die den ersetzten Oberzustand zum Vorzustand hatten. Jeder dieser Zustandswechsel wird durch ein Anzahl gleichartiger Zustandswechsel ersetzt. Für jeden Zustand, der eingesetzten Unterzustandsmaschine zuzüg-lich dem in Schritt zwei eingefügten Übergangszustand, wird ein solcher Ersatzzustands-wechsel generiert. Die Vorzustände dieser neuen Zustandswechsel sind die jeweiligen Zustände der Unterzustandsmaschine. Seine restlichen Definitionen übernimmt er von dem ersetzten Zustandswechsel. Zusätzlich werden den Aktionen der Ersatzzustandswechsel die Austrittsaktion des ersetzten Oberzustands vorangestellt.
- Die drei Schritte werden solange wiederholt, bis kein hierarchischer Zustand mehr existiert.

4.2.3 Constraints

Die auf der Ebene von abstrakten, hierarchischen oder elementaren ON-Klassen definierten Constraints müssen an die neuentstandenen elementaren ON-Instanzen angepaßt werden.

Abbildung 4.6
Die beiden transformierten Constraints aus Abbildung 3.5

```
Constraint Sicherheit1 {
 receive {Unten Control.Bahnschranke.Schranke} between 0 t1 after
 {Links Control.Bahnschranke.Abschnitt} &&
 {Oben Control.Bahnschranke.Schranke} occurred }
Constraint Sicherheit2 {
 receive {Unten Control.Bahnschranke.Schranke} between 0 t1 after
 {Rechts Control.Bahnschranke.Abschnitt} &&
 {Oben Control.Bahnschranke.Schranke} occurred }
```

Die in den Constraints vorkommenden Ports, werden durch zweielementige Listen ersetzt. Das erste Listenelement bezeichnet den Port, auf den sich der Constraint nach der Dehier-archisierung bezieht. Das zweite Element benennt die Instanz mit ihrem Hierarchiepfad an dem sich der Port befindet. Abbildung 4.6 zeigt die transformierten Constraints von Abbildung 3.5.

5. Kapitel

Verhaltensbeschreibung mit höheren Petri-Netzen

Für die formale Beschreibung des Verhaltens von Objektnetz-Spezifikationen werden höhere Petri-Netze eingesetzt. Die dabei verwendete Netzklasse ermöglicht es, alle Aspekte einer Objektnetz-Spezifikation, inklusive plattformspezifischer Zeiten und komplexer Verarbeitungsaktionen, zu beschreiben.

Die in Kapitel 3 erfolgte informelle Darstellung hatte das Ziel, den Leser mit den Entwurfsprinzipien und Entwurfselementen der Objektnetz-Methodik vertraut zu machen. In Kapitel 4 wurde die Auflösung von Hierarchie- und Vererbungsbeziehungen bei Objektnetz-Spezifikation beschrieben. Für die so umgeformten Objektnetze erfolgt schließlich die formale Beschreibung des Verhaltens; sie bildet die Basis für die Validierung.

Die Petri-Netz-Beschreibung der „flachen" Objektnetze erhält zusätzliche Informationen, die das spezifische Zeitverhalten auf einer konkreten Zielplattform widerspiegeln. Die Petri-Netz-Darstellung der Objektnetz-Constraints bildet den Abschluß von Kapitel 5.

Da Petri-Netze sich sehr gut dazu eignen, das Verhalten verteilter Systeme zu formalisieren, wurden sie für die Verhaltensbeschreibung und Validierung von Objektnetzen ausgewählt. Sicherlich hätten auch andere Formalismen das Verhalten von Objektnetz-Spezifikationen äquivalent abbilden können, aber nur höhere Petri-Netze waren in der Lage, dem Autor und somit auch dem Leser ein ausreichend anschauliches und dennoch mächtiges Beschreibungsmittel hierfür bereitzustellen. Höherer Petri-Netze ermöglichten die graphische Darstellung und Beschreibung aller Aspekte einer Objektnetz-Spezifikation.

Die Definition der eingesetzten höheren Netzklasse in Verbindung mit der Darstellung der Objektnetze als korrespondierendes Petri-Netz ermöglicht die formale Beschreibung des Verhaltens beliebiger Objektnetz-Spezifikationen.

5.1 Höhere Petri-Netze

Im vorliegenden Abschnitt wird eine spezielle höhere Petri-Netz-Klasse definiert. Dies erfolgt mit dem Ziel das Verhalten, der in Kapitel 3 eingeführten Objektnetz-Spezifikationen, zum Zwecke ihrer Validierung (vgl. Kapitel 5) formal zu beschreiben. Beim Entwurf der eingesetzten Objekt-Petri-Netze (der Begriff *Objekt* soll dabei lediglich die Nähe zu den Objektnetzen betonen) wurde besonders Wert auf eine praktikable Realisierung in einem Simulator [OPNTCL 98], [NütFen 98] gelegt. Ähnlich den *Coloured Petri Nets* von Jensen [Jensen 92] werden Netzanschriften in einer Modellierungssprache angegeben. Ohne, daß diese Sprache für die Definition der OPN fixiert werden soll, wird vorausgesetzt, daß die folgenden Begriffe eine definierte Bedeutung haben:

- *Datenklasse*. Der Begriff der Datenklasse wurde bereits in Kapitel 3.3.1 eingeführt. Die Menge aller Datenklassen wird mit *DC* bezeichnet. Ist $dc \in DC$, so wird *dc* als Menge aller Elemente (Wertbelegungen) dieser Datenklasse aufgefaßt. Für die Beschreibung der OPN muß mindesten $boolean \in DC$ sein. Dabei besitzen bestimmte Datenklassen die Eigenschaft, daß alle ihre Elemente ohne Informationsverlust in einer anderen Datenklasse gespeichert werden können. Dadurch ist auf der Menge der Datenklassen eine Halbordnung gegeben (die Datenklassenkompatibilität). Kann jedes Element einer Datenklasse dc_1 in ein Element einer Datenklasse dc_2 ungewandelt werden, so wird $dc_2 \succeq dc_1$ geschrieben.

- *Variable*. Die Datenklasse einer Variablen *v* wird durch *DC*(*v*) bezeichnet.

- *Ausdruck*. Ausdrücke enthalten Konstanten, Operatoren, Funktionen und Variablen. Jedem Ausdruck *E* wird eine Datenklasse *DC*(*E*) und eine Menge von Variablen *Var*(*E*) zugeordnet [Jensen 92].

- *Belegung*. Ist *V* eine Menge von Variablen, so wird eine Zuordnung *b*, welche jeder Variablen in *V* einen Wert ihrer Datenklasse zuordnet als Belegung von *V* bezeichnet. Ist *E* ein Ausdruck mit $Var(E) \subseteq V$ und *b* eine Belegung für *V*, so bezeichnet $E<b>$ den Wert, welchen *E* bei Ersetzung seiner Variablen durch deren Belegungen annimmt. Es gilt dabei $E<b> \in DC(E)$.

- *Folgen*. Für eine nicht leere Menge *M* bezeichnet *M** die Menge aller endlichen Folgen mit Elementen aus *M*, wobei *M** auch die leere Folge ε enthält. Es wird vereinbart, daß für eine endliche Folge $(a_1, \ldots, a_n)$, $last(a_1, \ldots, a_n) = a_n$ und $last(\varepsilon) = \varepsilon$ gilt.

In der konkreten Realisierung im Simulator ist die zugrunde gelegte Modellierungssprache das um die Datenklassen [Holbe 97] erweiterte Tcl. Im Unterschied zu den *Coloured Petri Nets* unterscheiden die Objekt-Petri-Netze zwei unterschiedliche Arten von Plätzen. Zum einen Zählplätze, die wie Plätze in klassischen Petri-Netzen [Petri 62], [Reisig 82], [KönQuä 88] uniforme Marken tragen und zum anderen Queue-Plätze, die unterscheidbare, strukturierte Marken aufnehmen können. Die Verwaltung dieser Marken erfolgt nach dem FIFO-Prinzip (*First-In-First-Out*). Eine ähnliche Vorgehensweise wird in den sogenannten Thor-Netzen [ScSoWi 95] verfolgt.

Die Definition der Objekt-Petri-Netze (OPN) erfolgt in zwei Schritten. Zuerst werden OPN ohne Zeitbewertung eingeführt. Darauf aufbauend erfolgt die Definition zeitbewerteter OPN.

5.1.1 Objekt-Petri-Netze ohne Zeitbewertung

Abbildung 5.1
Ein einfaches Objekt-Petri-Netz

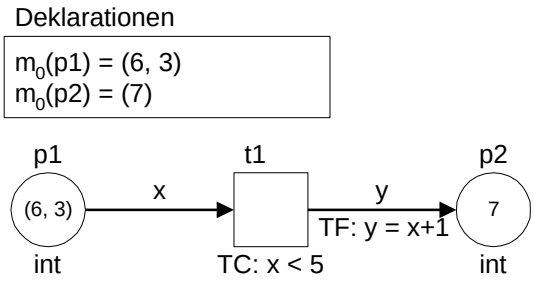


Abbildung 5.1 zeigt ein einfaches OPN mit zwei Queue-Plätzen und einer Transition. Dem Platz *p1*, im Vorbereich von *t1,* und dem Platz *p2,* im Nachbereich von *t1*, wurde die Datenklasse *int* zugeordnet. Das bedeutet, daß *p1* und *p2* mit Folgen, deren Elemente – die Marken – der Datenklasse *int* angehören, markiert werden können. Die Transition besitzt als zusätzliche Anschrift einen Boole'schen Ausdruck, die Transitionsbedingung *TC*, die der Variablen *x* (der Vorkanten) einen Boole'schen Wert zuordnet, der *true* sein muß, damit die Transition Konzession erhält. Neben der Transitionsbedingung besitzt die Transition eine weitere Funktion, die Transitionsfunktion *TF*, die der Variablen *y* (an der Nachkante) in Abhängigkeit von der Belegung der Transitionsvariablen (an der Vorkante) *x* einen Wert zuordnet. Die Transition *t1* kann nur ein einziges Mal schalten. Hierbei erhält die Variable *x* den Wert 3. Somit ergibt sich nach dem Schalten die neue Markierung $m(p1) = (6)$ und $m(p2) = (4, 7)$. Die Transitionsbedingung verhindert ein weiteres Schalten von *t1*.

● Definition der OPN ohne Zeitbewertung:

Im folgenden wird eine formale Definition der OPN gegeben. Wie in der Netztheorie üblich, werden OPN als Tupel dargestellt. Diese Notation dient nur der formalen Darstellung der Netze zum Zwecke ihrer Erklärung. Alle weiteren in der Arbeit auftretenden OPN werden graphisch dargestellt (vgl. Abbildung 5.1).

Ein OPN ist ein 12-Tupel $OPN = (P_1, P_{QUEUE}, T, F, F_{Test}, V, \Sigma, C, DC, TF, TC, m_0)$, welches den folgenden Beschreibungen genügt:

- P_1 ist die Menge der Zählplätze, P_{QUEUE} die Menge der Queue-Plätze. Die Menge aller Plätze wird mit P bezeichnet, d. h. $P = P_1 \cup P_{QUEUE}$.
- T bezeichnet die Menge der Transitionen.
- F ist die Menge der Flußkanten und F_{Test} die Menge der Testkanten. Es gilt $F \subseteq P \times T \cup T \times P$, $F_{Test} \subseteq P \times T$ und $F \cap F_{Test} = \varnothing$. Eine Fluß- oder Testkante f heißt Vorkante falls $f \in P \times T$ andernfalls heißt sie Nachkante. Testkanten sind also stehts Vorkanten.
- Für jeden beliebigen Netzknoten $k \in P \cup T$ ist die Menge aller Elemente im Vorbereich von k über $pre(k) = \{y \in P \cup T \mid (y, k) \in F\}$ bestimmt. Die Menge aller Elemente im Nachbereich von k ist über $post(k) = \{y \in P \cup T \mid (k, y) \in F\}$ bestimmt. Schleifen, bei dem ein Knoten ein

Element gleichzeitig im Vor- und im Nachbereich hat, sind dabei nicht ausgeschlossen. Die Menge der Plätze, die mit t über eine Testkante verbunden sind, ist über $test(t) = \{p \in P \mid (p, t) \in F_{Test}\}$ bestimmt. Die Menge der Transitionen, die mit p über eine Testkante verbunden sind, ist über $posttest(p) = \{t \in T \mid (p, t) \in F_{Test}\}$ bestimmt.

- Σ ist eine Menge von Variablen und DC die Menge der Datenklassen.
- Die Funktion C ordnet jedem Platz $p \in P_{QUEUE}$ eine Datenklasse $C(p) \in DC$ zu.
- Jeder Kante in $F \cup F_{Test}$ wird durch die Funktion V entweder eine natürliche Zahl oder eine Variable aus Σ zugeordnet. Dabei gilt: Ist $f = (p, t)$ oder $f = (t, p)$ mit $t \in T$ und $p \in P_1$, so ist $V(f)$ eine natürliche Zahl, die Vielfachheit der Kante f. $V(f)$ ist eine Variable aus Σ mit $DC(V(f)) \succeq C(p)$ für $f = (p, t)$ oder $DC(V(f)) \preceq C(p)$ für $f = (t, p)$, wenn $p \in P_{QUEUE}$.
- Jeder Transition $t \in T$ wird durch die Funktion TC ein Boole'scher Ausdruck $TC(t)$ mit $Var(TC(t)) \subseteq \{V(p, t) \mid (p, t) \in F \cup F_{Test}\}$ und $DC(TC(t)) = boolean$, die Transitionsbedingung zugeordnet.
- Die Funktion TF ordnet jeder Nachkante $(t, p) \in F$ mit $t \in T$ und $p \in P_{QUEUE}$ einen Ausdruck $TF(t, p)$ mit $Var(TF(t, p)) \subseteq \{V(p, t) \mid (p, t) \in F \cup F_{Test}\}$ und $DC(TF(t, p)) \preceq C(p)$ zu.
- Eine Markierung eines OPN ist eine Abbildung m, welche jedem Platz $p \in P_1$ eine nicht negative Zahl $m(p)$ und jedem Platz $p \in P_{QUEUE}$ eine endliche Folge $m(p) = (a_1, \ldots, a_n) \in C(p)^*$ zuordnet. Die Initialmarkierung eines OPN wird mit m_0 bezeichnet.
- Ist m eine Markierung, so bezeichnet $b(m)$ die Belegung der Vorkantenvariablen, welche jeder Vorkantenvariablen $V(p, t)$ den Wert $last(m(p))$ zuordnet.

● Vereinfachung der Notation:

Falls eine Transitionsbedingung in mehrere konjunktiv verknüpfte Teilbedingungen aufgeteilt werden kann, so daß jede dieser Teilbedingungen nur eine Variable enthält, können diese Teilbedingungen direkt an die entsprechenden Vorkanten geschrieben werden. Ebenso können die an den Nachkanten stehenden Funktionen $TF(t, p_1)$, ..., $TF(t, p_n)$ alternativ zu einer Transitionsfunktion $TF(t)$, die der Transition t direkt zugeordnet wird, zusammengefaßt werden. Eine weitere Vereinfachung der Notation ergibt sich für Transitionsfunktionen, die einer Variablen an einer Nachkante den unveränderten Wert einer Vorkantenvariablen zuweisen. In diesem Fall kann die Angabe der Transitionsfunktion entfallen, wenn die Variable an der Nachkante den gleichen Namen wie die Variable an der Vorkanten erhält.

● Konzession einer Transition:

Eine Transitionen $t \in T$ hat Konzession bei einer Markierung m, wenn:

- für alle Plätze $p \in P_1 \cap (pre(t) \cup test(t))$, $m(p) \succeq V(p, t)$, und für alle Plätze $p \in P_{QUEUE} \cap (pre(t) \cup test(t))$, $m(p) \neq \varepsilon$ und $TC(t)\langle b(m)\rangle = true$ gilt.

- **Konflikte und Nebenläufigkeit:**

Eine Teilmenge $T' \subseteq T$ ist konfliktbehaftet bei einer Markierung m, wenn:

- alle Transitionen $t \in T'$ bei m Konzession haben, und

eine der folgenden Bedingungen erfüllt ist:

- Es gibt einen Platz $p \in P_1$ so, daß mit $T'' = post(p) \cap T'$ und $T_{Test}'' = posttest(p) \cap T'$

$$m(p) < \sum_{t \in T''} V(p,t) + \max_{t \in T_{Test}''} V(p,t).$$

- Es gibt zwei verschiedene Transitionen t_1, $t_2 \in T'$ so, daß entweder $pre(t_1) \cap pre(t_2) \cap P_{QUEUE} \neq \varnothing$ oder $post(t_1) \cap post(t_2) \cap P_{QUEUE} \neq \varnothing$.

Eine Teilmenge $T' \subseteq T$ ist nebenläufig bei einer Markierung m, wenn alle Transitionen $t \in T'$ bei m Konzession haben und T' bei m nicht konfliktbehaftet ist.

- **Schalten konzessionierter Transitionen:**

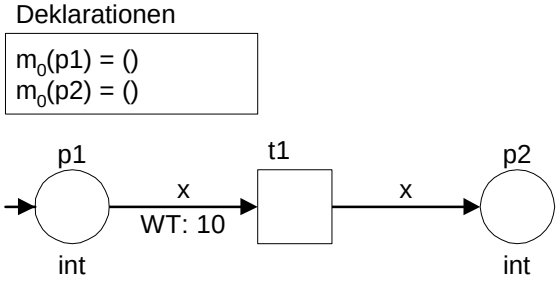
Durch Schalten einer bei einer Markierung m konzessionierten Menge von Transitionen $T' \subseteq T$ entsteht eine neue Markierung m'. Für diese gilt:

- $m'(p) = m(p) - \sum_{t \in post(p)} V(p,t) + \sum_{t \in pre(p)} V(t,p)$, wenn $p \in P_1$.

- Ist $p \in P_{QUEUE}$ und $m(p) = (a_1, \ldots, a_n)$, so wird, für alle $p \in pre(t)$ mit $t \in T'$, zuerst $last(m(p)) = a_n$ entfernt. Danach werden für alle $t \in pre(p)$ mit $t \in T'$ die Werte der Transitionsfunktionen $TF(t, p)\langle b(m)\rangle$ als neue Marken an den Anfang der Folge von p hinzugefügt.

5.1.2 Zeitbewertete Objekt-Petri-Netze

Abbildung 5.2
Ausschnitt aus einem zeitbewerteten OPN



Ähnlich dem Verfahren in [Jensen 92, S. 189] liegt den zeitbewerteten OPN eine globale Zeitskala zugrunde. Es wird dabei davon ausgegangen, daß die Zeit in diskreten Schritten voranschreitet. Alle im Modell vorkommenden Zeitbewertungen sind daher natürliche Zahlen. Die struk-turierten Marken und die Zählplätze eines zeitbewerteten OPN tragen Zeitstempel. Diese geben an, zu welchem Zeitschritt die Marke erzeugt wurde bzw. die Markierung im Zählplatz sich letztmalig geändert hat. Allen Vor-kanten wird eine Wartezeit zugeordnet. Abbildung 5.2 zeigt

einen Ausschnitt eines zeitbewerte-ten OPN. Die Transition *t1* kann frühestens zehn Takte, nachdem *p1* markiert wurde, schalten.

● Definition zeitbewerteter OPN:

Ein zeitbewertetes OPN ist ein Tripel (*OPN*, *PR*, *WT*). Dabei ist *OPN* ein Objekt-Petri-Netz, *PR* eine Abbildung von T in die Menge der natürlichen Zahlen und *WT* eine Abbildung, welche jeder Vorkante eine natürliche Zahl zuordnet. $PR(t)$ ist die Priorität der Transition t und $WT(f)$ ist die Wartezeit der Vorkante f.

● Situation:

Eine Situation ist ein Paar (i, s). Dabei ist i eine natürliche Zahl, der aktuelle Zeitschritt auf der globalen Zeitskala und s eine Abbildung, welche jedem Platz $p \in P_1$ ein Paar $(u(p), m(p))$ von natürlichen Zahlen und jedem Platz $p \in P_{\text{QUEUE}}$ eine endliche Folge $s(p) \in (\mathbb{N}_0 \times C(p))^* = ((u_1, a_1), \ldots, (u_n, a_n))$ von Paaren (u_j, a_j) bestehend aus einem Zeitstempel $u_j \in \mathbb{N}_0$ und einer Marke $a_j \in C(p)$ zuordnet.

Jeder Situation (i, s) entspricht eine Markierung $m_{(i,s)}$ des OPN. Dabei gilt für $p \in P_1$ mit $s(p) = (u(p), m(p))$, $m_{(i,s)}(p) = m(p)$ und für $p \in P_{\text{QUEUE}}$ mit $s(p) = ((u_1, a_1), \ldots, (u_n, a_n))$, $m_{(i,s)}(p) = (a_1, \ldots, a_n)$. Dabei wird $m_{(i,s)}$ als die zur Situation (i, s) korrespondierende Markierung bezeichnet.

Die Initialsituation $(0, s_0)$ ist durch $s_0(p) = (0, m_0(p))$ für $p \in P_1$ und $s_0(p) = ((0, a_1), \ldots, (0, a_n))$ für $p \in P_{\text{QUEUE}}$ definiert, dabei ist $(a_1, \ldots, a_n) = m_0(p)$ für $p \in P_{\text{QUEUE}}$.

● Konzession:

Eine Menge Transitionen $T' \subseteq T$ hat Konzession in einer Situation (i, s), wenn:

- T' bei der zu (i, s) korrespondierenden Markierung nebenläufig ist.
- $i - u(p) \geq WT(p, t)$ für alle $p \in P_1 \cap \left(\bigcup_{t \in T'} pre(t) \cup test(t) \right)$
- $i - u_n \geq WT(p, t)$ für alle $p \in P_{\text{QUEUE}} \cap \left(\bigcup_{t \in T'} pre(t) \cup test(t) \right)$

● Änderung der Situation:

Die Situationen ändern sich schrittweise. Die Folgesituation $(i+1, s')$ ergibt sich aus der vorhergehenden Situation (i, s) gemäß folgender Regeln:

- Es sei $T' \subseteq T$ eine maximale in (i, s) konzessionierte Menge, die der folgenden Bedingung genügt: Hat $t_0 \in T \setminus T'$ bei (i, s) Konzession, so gilt $PR(t_0) \leq \min_{t \in T'} PR(t)$. Ferner bezeichne m_1 die Markierung des OPN, welche durch Schalten aller Transitionen $t \in T'$ aus der zu (i, s) korrespondierende Markierung hervorgeht. Falls keine Transition in (i, s) Konzession hat, so ist $T' = \varnothing$ und, m_1 ist die zu (i, s) korrespondierende Markierung des OPN.

- Ist $p \in P_1$, so gilt $s'(p) = (u'(p), m'(p))$ mit $m'(p) = m_1(p)$ und

$$u'(p) = \begin{cases} i+1, & falls\ m(p) \neq m'(p) \\ u(p), & falls\ m(p) = m'(p) \end{cases}$$

- Ist $p \in P_{QUEUE}$, und $m_1(p) = (a'_1, \ldots, a'_n)$ so ist $s'(p) = ((u'_1, a'_1), \ldots, (u'_n, a'_n))$ und

$$u'_j = \begin{cases} i+1, & falls\ p \in post(T')\ und\ j = 1 \\ u_j, & sonst. \end{cases}$$

● Vergleichbarkeit zweier Situationen:

Zwei unterschiedliche Situationen (i, s) und (i', s') werden als vergleichbar bezeichnet, wenn $i - u(p) = i' - u'(p)$ und $m(p) = m'(p)$ für alle $p \in P_1$ und gleichzeitig $((i - u_1, a_1), \ldots, (i - u_n, a_n)) = ((i' - u'_1, a'_1), \ldots, (i' - u'_n, a'_n))$ für alle $p \in P_{QUEUE}$ gilt. Ebenso werden zwei Situationen (i, s) und $(i+k, s')$ mit $k \geq 1$ als vergleichbar bezeichnet, wenn $s = s'$ gilt. Vergleichbarkeit ist eine Äquivalenzrelation.

● Situationsgraph:

Die Knotenpunktmenge des Situationsgraphen SG eines zeitbewerteten OPN wird durch die Äquivalenzklassen der vergleichbaren Situationen gebildet. Zwei solcher Äquivalenzklassen S_1 und S_2 werden durch eine Kante von S_1 nach S_2 verbunden, wenn S_2 eine Situation $(i + 1, s')$ enthält, welche Folgesituation einer Situation (i, s) aus S_1 ist. Die Kante ist zum einen mit den Transitionen beschriftet, deren Schalten die Situation verändert und zum anderen mit dem Zeitschritt i beschriftet, bei dem erstmalig der Übergang S_1 nach S_2 erfolgte.

● Zustandsmaschinennetz (ZM-Netz):

In den folgenden Verhaltenbeschreibungen werden sogenannte Zustandsmaschinennetze (ZM-Netz) eingesetzt. Diese speziellen OPN enthalten nur Zählplätze, Flußkanten und Transitionen. Die Zählplätze tragen maximal eine Marke. Die Transitionen eines ZM-Netzes besitzen maximal eine Nachkante und eine Vorkante zu einem Zählplatz des ZM-Netzes. ZM-Netze sind zusammenhängend [Starke 90].

5.2 Verhaltensbeschreibung

Der von [Schmidt 98] realisierte Algorithmus zur Auflösung der Hierarchie generiert für ein ausgewähltes Projekt ein flaches Objektnetz. Es enthält alle Informationen, die für seine Implementierung bzw. Validierung benötigt werden. Für die Validierung erfolgt seine formale Verhaltensbeschreibung auf Basis von zeitbewerteten OPN.

Für eine „flache" Objektnetz-Spezifikation, deren elementare ON-Instanzen den Rechnerknoten einer verteilten Zielplattform zugeordnet wurden (vgl. Kapitel 3.5), erfolgt die Verhaltensbeschreibung. Sie beginnt bei den elementaren ON-Instanzen, die als ZM-Netze dargestellt werden. Die ZM-Netze werden schrittweise ergänzt, bis schließlich alle Netzkonstrukte enthalten sind, die für eine vollständige Verhaltensbeschreibung benötigt werden.

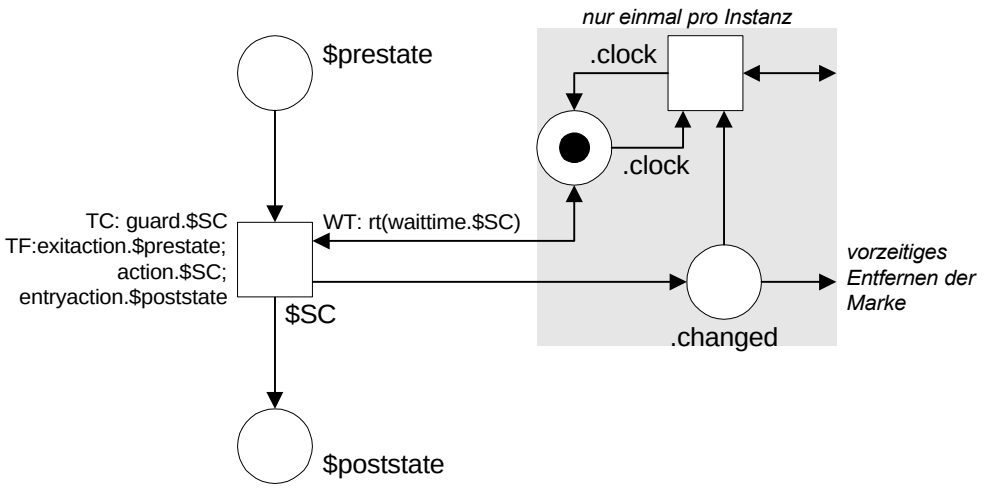
5.2.1 Elementare ON-Instanzen

Das Verhalten einer elementaren ON-Instanz wird im Kern durch ein ZM-Netz beschrieben. Die Zählplätze des ZM-Netzes repräsentieren die Zustände der ON-Instanz. Der Startzustand wird durch einen Platz mit Initialmarke dargestellt. Jedes ZM-Netz besitzt genau einen so markierten Zählplatz. Zustandswechsel werden durch die Transitionen des ZM-Netzes repräsentiert. Das Verhalten von Ports und Nachrichtenverbindungen, die die Kopplung der ON-Instanzen realisieren werden durch zusätzliche Netzkonstrukte dargestellt.

5.2.1.1 Zustandswechsel mit Wartezeit

Die Transitionen des ZM-Netzes, die das Verhalten der Zustandswechsels beschreiben, sind mit Transitionsfunktionen (*TF*), Transitionsbedingungen (*TC*) und Wartezeiten (*WT*) beschriftet. *TF* leitet sich aus der Verkettung der Austrittsaktion des Vorzustands (*$prestate*), der Aktion des Zustandswechsels und der Eintrittsaktion des Folgezustands (*$poststate*) ab. Unbelegte Referenzen werden durch leere Aktionen ersetzt. *TC* wird durch den Guard des Zustandswechsels bestimmt. Referenziert der Zustandswechsel keinen Guard, so wird für *TC* der

Abbildung 5.3
Eine Transition eines ZM-Netzes mit Wartezeitmechanismus



Boole'sche Wert *true* eingesetzt. Die Wartezeit des Zustandswechsels wird durch den Parameter *waittime.$SC* definiert. Dieser Parameter bestimmt nicht direkt *WT*, es handelt sich vielmehr um eine ganzzahlige, normierte Zeitangabe, die erst mit dem auf Klassenebene definierten Skalierungsfaktor (*timescale*) multipliziert werden muß. Diese Berechnung übernimmt die Funktion *rt* (vgl. Abbildung 5.3).

Ein zusätzliches Netzkonstrukt, welches einmal für jede elementare ON-Instanz existiert (in Abbildung 5.3 grau unterlegt dargestellt), beschreibt den speziellen Wartezeitmechanismus der Objektnetze; alle Wartezeiten werden auf den Anfang des nächsten Bearbeitungszyklus´ (Scheduler-Umlauf) verschoben. Nach dem Schalten einer Transition im ZM-Netz wird zusätzlich der Platz *.changed* markiert. Dies bewirkt zusammen mit einer weiteren Bedingung (Testkante an der Transition *.clock*), die das Ende eines Scheduler-Umlaufs anzeigt, das Rücksetzen des Zeitstempels in *.clock*. Falls der Scheduler-Umlauf nach dem Schalten von *$SC* noch nicht abgeschlossen wurde, wird die Marke in *.changed* wieder entfernt. Somit kann erst nach einem Zustandswechsel am Ende des Scheduler-Umlaufs die Transition *.clock* schalten.

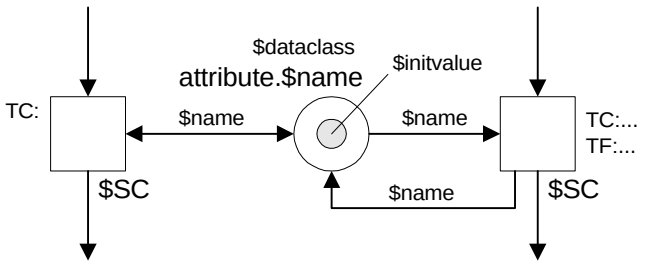
• Aktionen und Guards:

Aktionen und Guards sind Software-Prozeduren bzw. Software-Funktionen, die von Zustandswechseln oder Zuständen aus aktiviert werden. Wobei Zustände nur Aktionen referenzieren. Der Aktions- bzw. Guardcode bleibt für die Petri-Netz-Darstellung unverändert. Er bildet direkt die Transitionsfunktion (*TF*) bzw. die Transitionsbedingung (*TC*). Über spezielle Variablen an den Kanten haben diese Zugriff auf die Puffer der Ports, auf die Attribute und auf die Aktor/Sensor-Objekte.

• Attribute:

Attribute sind Variablen; sie sind einer elementaren ON-Instanz zugeordnet. Sie können von Aktionen und Guards geschrieben bzw. gelesen werden. Attribute besitzen eine definierte Startbelegung (*initvalue*), die in der Regel auf Klassenebene definiert wird. Attribute, deren Initialwert durch einen Parameterport festgelegt wird, erhalten ihren Startbelegung auf Instanzebene.

Abbildung 5.4
Der Zugriff auf Attribute



Attribute werden durch einen Queue-Platz mit einer Marke repräsentiert (vgl. Abbildung 5.4). Die Marke trägt die Wertbelegung des Attributs. Jede Transition des ZM-Netz, die über *TC* nur lesenden Zugriff auf dieses Attribut hat, wird über eine Testkante mit diesem Attributplatz verbunden. Transitionen, die über *TF* zusätzlich schreibenden Zugriff haben, werden statt dessen über eine Vor- und Nachkante mit dem Attributplatz verbunden. Die Kantenvariablen (siehe Abbildung 5.4) erhalten ihren Namen von dem Namen (*$name*) des Attributes. Über sie erfolgt in *TF* und *TC* der

eigentliche Zugriff auf das Attribut. Der sequentielle Charakter elementarer ON-Instanzen garantiert, daß kein Konflikt um ein Attribut bzw. um die Marke in einem Attributplatz entsteht.

- **Kopplung mit der Umgebung durch Aktor/Sensor-Objekte:**

Die Kopplung der Steuerungs- und Umgebungsspezifikation erfolgt über Objekte der abstrakten Aktor/Sensor-Klassen. Die Beschreibung des Verhaltens dieser Kopplung erfolgt über globale Koppelattribute, welche durch einen Attributplatz repräsentiert werden (vgl. Abbildung 5.4). Für jedes referenzierte AS-Objekt steht ein Koppelattribut. Auf dieses Attribut kann sowohl von der referenzierenden Instanz der Steuerungsspezifikation, als auch von der referenzierenden Instanz der Umgebung aus zugegriffen werden. Der Name des Attributs wird durch das AS-Objekt bestimmt. Die Datenklasse des Attributs leitet sich in zwei Schritten aus der Klasse des AS-Objekts ab:

- Der erste Schritt liefert eine Datenklasse, die einen leeren String und alle Methodennamen der Klasse als Wertbelegungsmenge definiert. Über diese Datenklasse wird gespeichert, welche Methode zuletzt ausgeführt wurde. Für die AS-Klasse *Motor* (vgl. Abbildung 3.20) z.B. lautet die Datenklassendefinition: *DataClass ascmethods.Motor string {enum {} MotorStop MotorRechtsLauf MotorLinksLauf }*

- Aufbauend auf dieser ersten Datenklasse und den Datenklassen der Rück- bzw. Übergabeparametern der AS-Methoden erfolgt schließlich die Definition der komplexen Attribut-Datenklasse. Sie besitzt eine Komponente der zuvor definierten Datenklasse (*ascmethods.$ASC*). In dieser Komponente wird der Name der zuletzt ausgeführten Aktor-Methode gespeichert. Für jede AS-Methode, die entweder einen Rückgabe- oder einen Übergabeparameter besitzt, wird eine weitere Komponente aufgenommen. Diese Komponenten speichern die Daten, die zwischen Umgebung und Steuerung ausgetauscht werden. Die Datenklasse dieser Komponenten ergeben sich aus den jeweiligen Rückgabe- bzw. Übergabeparametern der AS-Methoden. Für die AS-Klasse *KontaktSensor* (vgl. Abbildung 3.20) lautet die komplette Klassendefinition: *DataClass asc.KontaktSensor {{ascmethods.KontaktSensor .method} {boolean KontaktZu}}*

Der Zugriff auf die Koppelattribute erfolgt von der Steuerungsspezifikation aus durch die Methoden der Aktor/Sensor-Objekte und von der Umgebungsspezifikation durch die dazu komplementären Methoden. Aktor-Methoden und komplementäre Sensor-Methoden greifen (von den Aktionen aus) schreibend auf die Koppelattribute zu. Sensor-Methoden und komplementäre Aktor-Methoden greifen (von den Guards aus) nur lesend zu.

5.2.1.2 Zugeordnete Ports

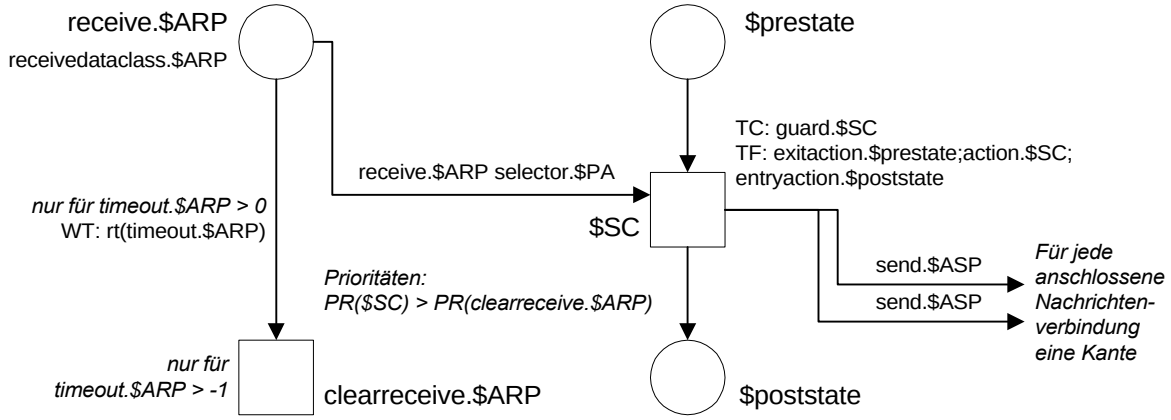
Eine Transition des ZM-Netzes erhält durch die Zuordnung eines synchronen oder mehrerer asynchroner Empfangsports zusätzliche Vorbedingungen. Erst wenn alle zugeordneten Empfangsports eine passende Nachricht haben, kann die Transition schalten. Infolge dieses Schaltens werden zugeordnete Sendeports aktiviert.

- **Zugeordnete asynchrone Ports:**

Abbildung 5.5 zeigt das korrespondierende Petri-Netz für einen Zustandswechsel (*$SC*) mit einem zugeordneten asynchronen Empfangsport (*$ARP*) und einem asynchronen Sendeport (*$ASP*). Die Beschreibung des Verhaltens für diesen Zustandswechsels bezieht sich auf Abbildung 5.3. Zusätzlich wurden in Abbildung 5.5 die Netzkonstrukte, die die zugeordneten asynchronen Ports repräsentieren, eingefügt. Der Queue-Platz *receive.$ARP* und die Transition *clearreceive.$ARP* repräsentieren den Empfangsport. Der Sendeport wird durch die beiden Kanten in der linken Bildhälfte wiedergegeben.

Die Definition eines ARP enthält neben dem Namen (*$ARP*) drei weitere Parameter (*receivedataclass*, *timeout* und *importance*). Der Parameter *timeout*, der das Pufferungsverhalten des Ports festlegt, verkörpert nicht direkt die Pufferzeit, es handelt sich vielmehr um eine ganzzahlige, normierte Zeitangabe. In Abbildung 5.5 wird diese Zeitangabe analog zu den Wartezeiten durch die Funktion *rt* entnormiert.

Abbildung 5.5
Transition des ZM-Netzes mit den Netz-Konstrukten für zugeordnete asynchrone Ports



Die zusätzlich zum ZM-Netz eingesetzte Transition *clearreceive.$ARP* beschreibt das Löschen „veralteter" Nachrichten. Bei *timeout.$ARP* = -1 (unendlich) entfällt diese Transition. Bei *timeout.$ARP* = 0 wird jedesmal, wenn die Instanz den Steuerfokus besitzt, eine Marke (Nachricht) aus dem Pufferplatz (*receive.$ARP*) entfernt. Konflikte zwischen Transitionen des ZM-Netzes und den Löschtransitionen werden über Prioritäten (*PR*) zugunsten des ZM-Netzes entschieden. Der Pufferplatz besitzt grundsätzlich keine begrenzende Kapazität. Der Parameter *importance* schlägt sich ausschließlich bei der Beschreibung der Nachrichtenverbindungen nieder. Die Anbindung an den Zustandswechsel erfolgt über eine Vorkante, die den optionalen Selektor (*selector.$PA*) als zusätzliche Bedingung (*TC* an der Vorkante) trägt.

Für jeden Zustandswechsel, dem ein asynchroner Sendeport zugeordnet wurde, werden die an diesem Sendeport angeschlossenen Nachrichtenverbindungen als einzelne Nachkanten (zwei in Abbildung 5.5) umgesetzt.

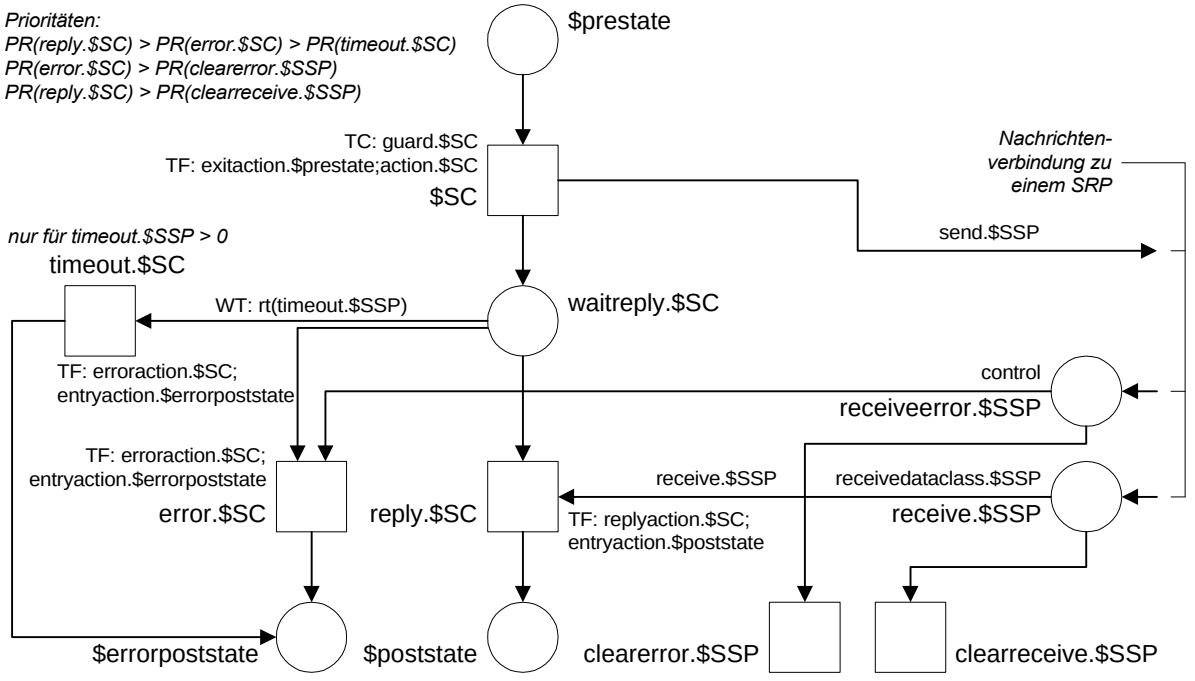
Über die Kantenvariablen *receive.$ARP* und *send.$ASP* an den Vor- bzw. Nachkanten erfolgt (analog den Attributen) der Zugriff von Guards und Aktionen auf die Puffer der Ports.

● Zugeordnete synchrone Sendeports:

Ein Zustandswechsel mit einem zugeordneten SSP besitzt die komplexeste Petri-Netz-Semantik (siehe Abbildung 5.6). Sie muß gewährleisten, daß die sendende Instanz sich nicht verklemmt. Dies kann dadurch geschehen, daß sie auf eine Antwort wartet, die in Folge eines Fehler nicht eintreffen kann.

Die Transition (*$SC)* des ZM-Netzes wird hierzu gemäß Abbildung 5.6 ergänzt. Es werden die Transitionen *timeout.$SC*, *error.$SC* und *reply.$SC* sowie der Zählplatz *waitreply.$SC* eingefügt. Im Unterschied zu Abbildung 5.3 entfällt bei *$SC* die Eintrittsaktion des Folgezustands (*entryaction.$poststate*). Analog zu Abbildung 5.5 wird an *$SC* eine Nachkante angeschlossen, die für die Versendung der Nachricht zuständig ist.

Abbildung 5.6
Transition des ZM-Netzes mit dem Netzkonstrukt für einen zugeordneten synchronen Sendeport



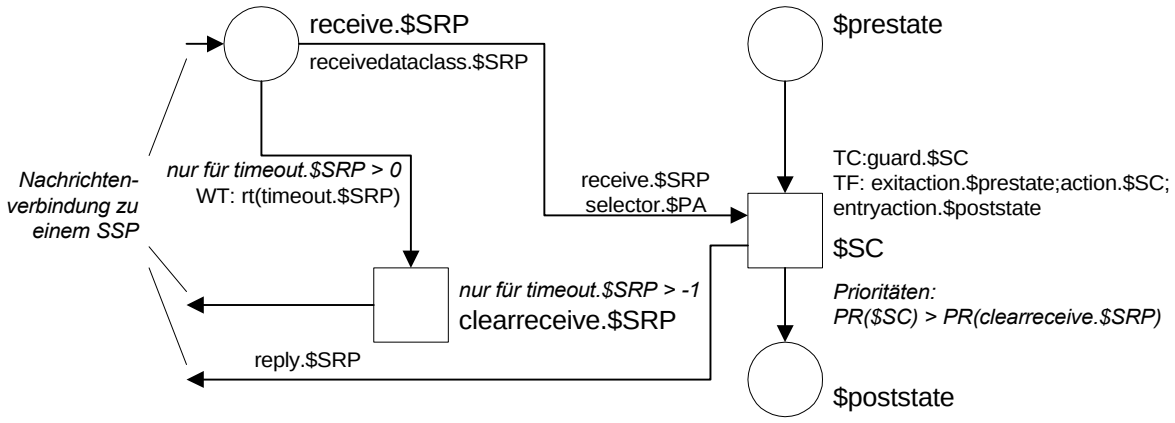
Ein Zustandswechsel, dem ein SSP zugeordnet wurde, durchläuft grundsätzlich immer zwei, durch einen Zwischenzustand (*waitreply.$SC*) getrennte, Transitionen. Dabei bestehen für die zweite Transition drei Alternativen. *Reply.$SC* schaltet, wenn vor Ablauf des spezifizierten Zeitlimits (*timeout.$SSP*) eine Antwortnachricht am SSP empfangen wurde. In diesem Fall wird die Antwortaktion (*replyaction*) und die Eintrittsaktion des Folgezustands ausgeführt. Die Transition *timeout.$SC* schaltet, wenn das Zeitlimit des SSP erreicht wurde. Das hat zur Folge, daß die spezielle Fehleraktion (*erroraction*) des Zustandswechsel und die Eintrittsaktion des Fehlerfolgezustands ausgeführt wird. Als Fehlerfolgezustand kann ein beliebiger Zustand der Zustandsmaschine definiert werden. Wird auf der Gegenseite (beim SRP) das Zeitlimit

überschritten (vgl. Abbildung 5.7), so wird der SSP durch eine Fehlerantwortnachricht (in *receiveerror.$SSP*) über diese Zeitüberschreitung informiert. Der Empfang einer solchen Fehlernachricht konzessioniert die dritte Transition *error.$SC* (siehe Abbildung 5.6). Sowohl Antwortnachricht, als auch Fehlerantwortnachricht, werden durch *clearreceive.$SSP* bzw. *clearerror.$SSP* bei der nächsten Aktivierung der Instanz (analog zu Abbildung 5.5) gelöscht.

- **Zugeordnete synchrone Empfangsports:**

Das Gegenstück zu einem Zustandswechsel mit SSP zeigt Abbildung 5.7. Das Verhalten solcher Zustandswechsel, die einen zugeordneten SRP besitzen, basiert auf der Beschreibung von Abbildung 5.3. Die Anbindung des Empfangs- und des Antwortnetzkonstrukts (*reply*) erfolgt analog zu einem ARP und ASP (vgl. Abbildung 5.5). Die Fehlerantwortnachricht wird aus der Aktivierung der Transition *clearreceive.$SRP* abgeleitet (siehe Abbildung 5.7).

Abbildung 5.7
Transition des ZM-Netzes mit dem Netzkonstrukt für einen zugeordneten synchronen Empfangsport



Ein Zustandswechsel mit SSP, der nicht beliebig lange auf eine Antwortnachricht wartet, wechselt nach Ablauf des Timeouts (*timeout.$SSP*) in seinen Fehlerfolgezustand. Wird nun derselbe Zustandswechsel wieder aktiviert, so kann es vorkommen, daß er nach dem Absenden einer neuen Nachricht, die nun veraltete, zuvor verpaßte Antwortnachricht, doch noch erhält. Somit bedeutet das erstmalige Wechseln in den Fehlerfolgezustand, daß der Zusammenhang von Nachricht zu Antwortnachricht für den Rest der Bearbeitung verloren gehen kann. Diese Möglichkeit muß durch zusätzliche Objektnetz-Konstrukte, die eine erneute Aktivierung dieses SSP ausschließen, berücksichtigt werden.

Einem Zustandswechsel kann maximal nur ein SSP und ein SRP zugeordnet werden. Für die Zuordnung von asynchronen Ports bestehen dagegen keine Beschränkungen.

5.2.2 Nachrichtenverbindungen

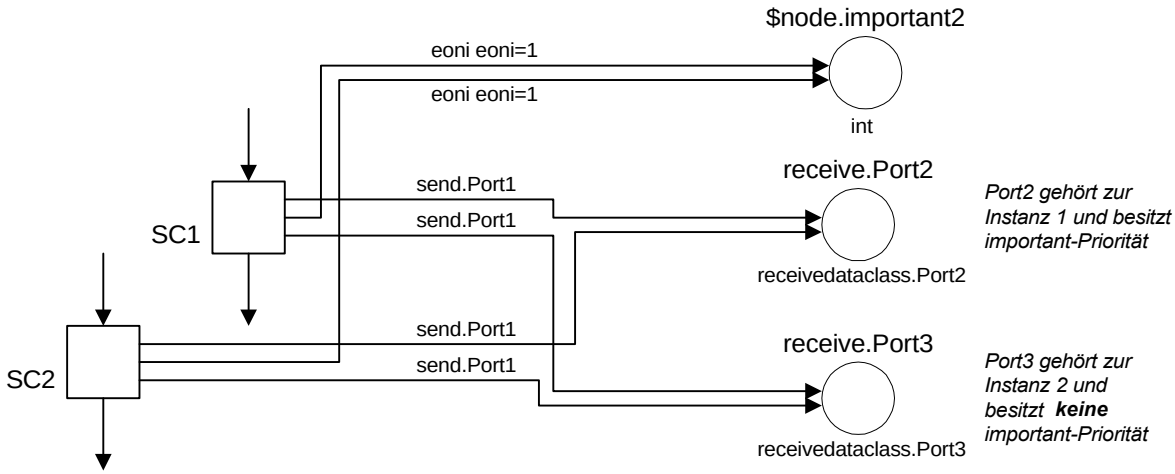
Eine Nachrichtenverbindung, die zwei komplementäre, wertverträgliche Ports verbindet, stellt einen logischen Kommunikationskanal zwischen zwei elementaren ON-Instanzen dar. Verbindet sie zwei asynchrone Ports, so verkörpert sie einen unidirektionellen Kanal. Werden dagegen zwei synchrone Ports verbunden, so verkörpert die Nachrichtenverbindung drei einzelne unidirektionelle Kanäle. Aufgrund des unterschiedlichen zeitlichen Verhaltens von Nachrichtenverbindungen, die unterschiedliche Rechnerknoten verbinden, und solchen, die lokale Instanzen verbinden, erfolgt deren Beschreibung in zwei getrennten Abschnitten.

5.2.2.1 Nachrichtenverbindungen zwischen lokalen Instanzen

- **Asynchrone Nachrichtenverbindungen:**

Das Verhalten von Nachrichtenverbindungen, die einen asynchronen Sendeport (*Port1* in Abbildung 5.8) mit einem oder mehreren asynchronen Empfangsports (*Port2* und *Port3*) verbinden, wird durch mehrere Nachkanten beschrieben. Die Transitionen des ZM-Netzes (*SC1* und *SC2*), denen *Port1* zugeordnet wurde, werden durch eine Nachkante mit den Pufferplätzen *receive.Port2* und *receive.Port3* verbunden (vgl. Abbildung 5.8). Auf die Variablen (Name: *send.Port1*, Datenklasse: *senddataclass.Port1*) an diesen Kanten hat *TF* schreibenden Zugriff. *TF* bestimmt somit, welche Informationen mit einer Nachrichten übertragen werden.

Abbildung 5.8
Port1 ist über zwei Nachrichtenverbindungen mit *Port2* und *Port3* verbunden



Handelt es sich um einen Port mit *important*-Priorität (*Port2*), so wird für jede Transition eine zusätzliche Kante eingesetzt. Diese Kanten erhalten als *TF* eine feste Nummer zugeordnet; sie bezeichnet die Instanz des *important*-Empfangsports. Schaltet die Transition, so wird eine Marke mit dieser Nummer in *$node.important2* (Bestandteil des Schedulers) abgelegt. Dies ist für den Scheduler der Hinweis, die Instanz mit dieser Nummer bevorzugt zu behandeln.

- **Synchrone Nachrichtenverbindungen:**

Da eine Mehrfachzuordnung synchroner Ports ausgeschlossen wurde, ist die Nachrichtenverbindung zwischen einem synchronen Sendeport (*$SSP*) und einem synchronen Empfangsport (*$SRP*) ist immer eine 1-zu-1-Verbindung. Im Unterschied zu asynchronen Verbindungen beschreiben bei synchronen *drei* Kanten die Koppelstruktur. Diese sind sowohl in Abbildung 5.6 als auch in 5.7 bereits zu sehen. Die erste Kante (von oben gezählt) verkörpert den Sende-kanal. Sie verbindet die Transition *$SC* in Abbildung 5.6 mit dem Pufferplatz *receive.$SRP* in Abbildung 5.7. Die zweite Kante verkörpert den Fehlerantwortkanal. Sie verbindet die Transition *clearreceive.$SRP* in Abbildung 5.7 mit dem Pufferplatz *receiveerror.$SSP* in Abbildung 5.6. Die dritte Kante stellt den Antwortkanal dar und verbindet die Transition *$SC* in Abbildung 5.7 mit dem Pufferplatz *receive.$SSP* in Abbildung 5.6.

5.2.2.2 Nachrichtenverbindungen zwischen verteilten Rechnerknoten

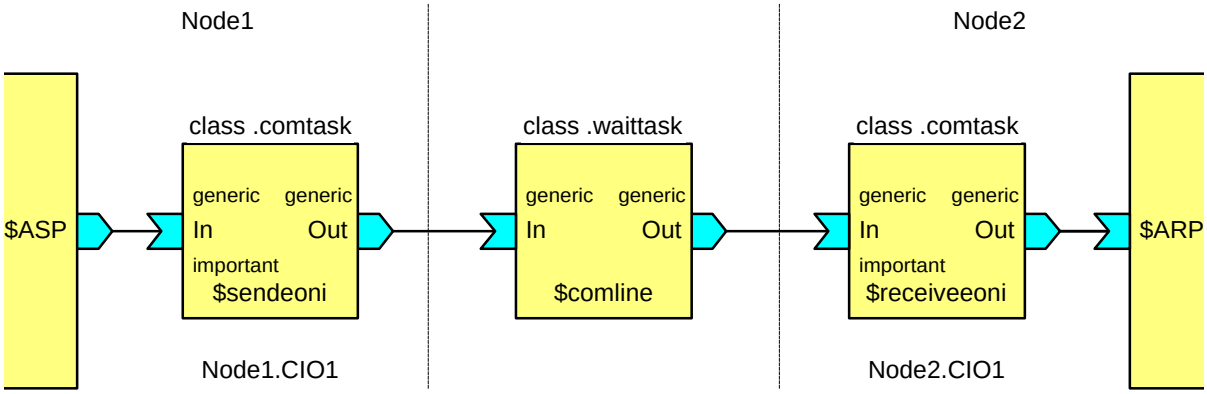
Eine zentrale Eigenschaft der Objektnetz-Methodik besteht darin, daß elementare ON-Instanzen, die über Nachrichtenverbindungen miteinander kommunizieren, über mehrere Rechnerknoten verteilt werden können. Diese Verteilung ist allerdings nur dann möglich, wenn auch eine unmittelbare physikalische Kommunikationsverbindungen zwischen den Rechnerknoten existiert (z.B. zwischen *Node1.CIO1* und *Node2.CIO1*, in Abbildung 3.39). Diese Zuordnung (*mapping*) der Instanzen auf verschiedene Knoten ist vollständig vom eigentlichen Entwurfsprozeß entkoppelt. Für den Entwurf einer Objektnetz-Spezifikation ist es ohne Belang, ob eine Nachrichtenverbindung nach dem Mapping in eine lokale Kommunikationsverbindung oder in eine Kommunikationsverbindung zwischen zwei Rechnerknoten überführt wird.

- **Asynchrone Nachrichtenverbindungen:**

Der Versand einer Nachricht über eine Kommunkationsschnittstelle (*Node1.CIO1*) des Rechnerknotens (vgl. Abbildung 3.36) ist ebenso wie der Empfang an einer Schnittstelle (*Node2.CIO1*) mit einer zusätzlichen Zeitverzögerung verbunden. Sie unterteilt sich in vier Abschnitte (vgl. Abbildung 3.40): Sendezeit (*sendtime*), Sendewartezeit (*sendwait*), Empfangswartezeit (*receivewait*) und Empfangszeit (*receivetime*). Alle vier Zeitparameter hängen von der Datenklasse der übertragenen Nachricht und der beteiligten Kommunikationsschnittstelle (*Node1.CIO1* bzw. *Node2.CIO1*) ab. Sende- und Empfangswartezeit belasten die CPU nicht. Die Sendezeit muß durch die CPU des sendenden Rechnerknotens (*Node1*), die Empfangszeit durch den empfangenden Rechnerknoten (*Node2*) aufgebracht werden.

Das Verhalten einer Nachrichtenverbindung zwischen den Kommunikationsschnittstellen (CI-Objekte) zweier Rechnerknoten kann für den asynchronen Fall mit Objektnetz-Elementen dargestellt werden (siehe Abbildung 5.9). Für die beteiligten CI-Objekte werden zusätzliche elementare ON-Instanzen (*$sendeoni* und *$receiveeoni*) eingesetzt, die über lokale asynchrone Nachrichtenverbindungen mit der eigentlich sendenden bzw. empfangenden Instanz verbunden werden.

Abbildung 5.9
Darstellung einer asynchronen Nachrichtenverbindung zwischen zwei Rechnerknoten



Beide Instanzen besitzen die gleiche Struktur (Klasse: *.comtask*): Einem einzelnen Zustandswechsel werden beide Ports (*In* und *Out*) zugeordnet. Sein Vor- und Folgezustand ist der Startzustand; der einzige Zustand der Klasse. Eine Aktion, die von diesem referenziert wird, kopiert die Nachrichten, die an dem *important*-Port *In* anstehen, auf den Sendeport *Out*. In Bezug auf die Bearbeitungsdauer für diese Kopieraktion unterscheiden sich die Instanzen allerdings. Die maximale Dauer der Kopieraktion in der Instanz *$sendeoni* wird durch die Funktion *sendtime*(*Node1.CIO1*,*senddataclass.$ASP*) bestimmt. Die Bearbeitungsdauer der Aktion in *$receiveeoni* wird durch die Funktion *receivetime*(*Node2.CIO1*, *receivedata-class.$ARP*) definiert (vgl. Kapitel 3.4).

Für die eigentliche Kommunikationsverbindung der CI-Objekte wird die Pseudo-Instanz *$comline* eingefügt. Im Unterschied zu den beiden Instanzen zuvor, ist sie keine vollwertige ON-Instanz; sie wird nicht durch eine Rechnerknoten-CPU abgearbeitet. Sie dient lediglich dazu, den zeitlichen Versatz zwischen dem Senden durch *$sendeoni* und dem Empfangen durch *$receiveeoni* darzustellen. Dieser Versatz beträgt für Abbildung 5.9: *sendwait*(*Node1.CIO1*, *senddataclass.$ASP*) + *receivewait*(*Node2.CIO1*, *receivedataclass.$ARP*).

- **Synchrone Nachrichtenverbindungen:**

Der Transformationsprozeß für synchrone Nachrichtenverbindung bedient sich der in Abbildung 5.9 dargestellten asynchronen Verbindungen. Eine synchrone Verbindung wird aus drei solchen Verbindungen zusammengesetzt.

5.2.3 Scheduling

Um eine möglichst implementierungsnahe Beschreibung des zeitlichen Verhaltens zu erhalten, muß die zeitliche Charakteristik der Zielplattform einbezogen werden. Neben der bereits berücksichtigten Verteilung und Kommunikation werden die realen Verarbeitungszeiten der

Aktionen und Guards sowie die Art und Weise, wie elementare ON-Instanzen auf einem Rechnerknoten koordiniert werden, in die Verhaltensbeschreibung integriert.

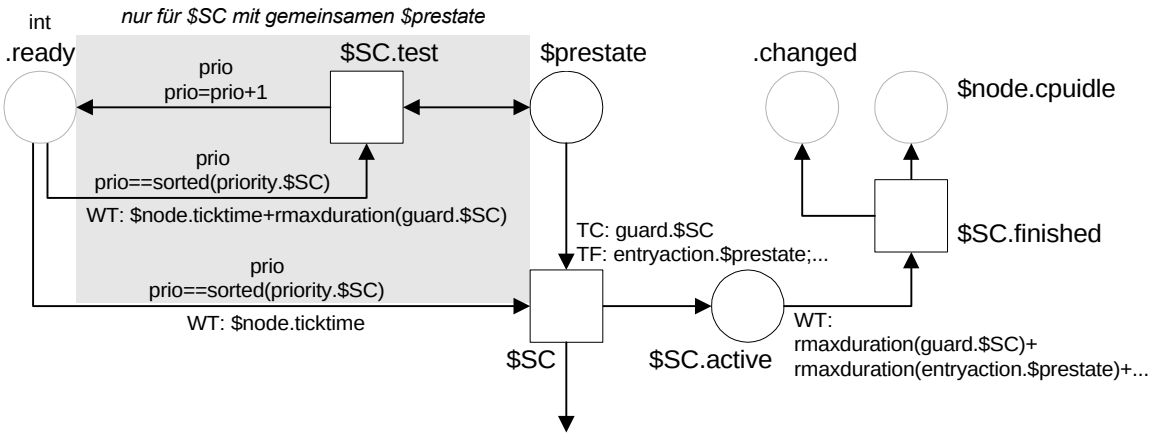
5.2.3.1 Abarbeitungsmodell der Objektnetz-Methodik

Die Objektnetz-Methodik legt ein spezielles Abarbeitungsmodell für alle Projekte fest. Bei diesem Abarbeitungsmodell werden sämtliche Aktivitäten, wie Nachrichtenversand und Nachrichtenverarbeitung, Aktionen- und Guards-Abarbeitung exklusiv und unteilbar von der CPU des jeweiligen Rechnerknotens ausgeführt. Alle Aktivitäten, die ein Rechnerknoten ausführt, werden folglich sequentiell abgearbeitet. Dieses kooperative Abarbeitungsmodell (kooperatives Multitasking) verlangt von den Aktionen (und den anderen Aktivitäten), daß sie eine möglichst kurze Ausführungsdauer besitzen, um während ihrer Bearbeitung das restliche System nicht unnötig lange zu blockieren. Eine Aktion bzw. ein Guard wird somit einem kritischen Bereich in einem preemptiven Multitasking System [Tanenbaum 95] gleichgesetzt.

Das im folgenden dargestellte Abarbeitungsmodell für Objektnetze realisiert sowohl die Verwaltung von *important*-Ports als auch die Kommunikation mit anderen Rechnerknoten. Das zeitliche Verhalten einer Objektnetz-Steuerungsspezifikation unter diesem speziellen kooperativen Abarbeitungsmodell kann vollständig beschrieben werden. Die Beschreibung für eine Umsetzung mit preemptiven Echtzeitbetriebsystem (RTOS), bei dem eine Aktivität an beliebiger Stelle durch eine andere Task unterbrochen werden kann, wäre um ein Vielfaches komplexer. Diese Komplexität würde die Validierung zeitlicher Anforderung unnötig erschweren. Ebenso würde die Implementierung gegenüber der kooperativen Variante mehr Ressourcen in Anspruch nehmen. Für die Realisierung des Abarbeitungsmodell der Objektnetz-Methodik genügt ein einfaches kooperativ arbeitendes Echtzeitbetriebssysteme, wie z.B. OSEK [OSEK 97]. Alternativ kann die Realisierung auch gänzlich ohne RTOS durch einen einfachen timergesteuerten Scheduler erfolgen. In [Cooling 91] werden neben dem kooperativen und preemptiven weitere alternative Scheduling-Algorithmen beschrieben.

5.2.3.2 Die Abarbeitung von Aktionen und Guards einer Instanz

Abbildung 5.10
Ergänzungen, die das zeitliche Verhalten von Aktionen und Guards beschreiben



Um das zeitliche Verhalten von Guards und Aktionen zu beschreiben, werden die Transitionen, die eine *TC* bzw. *TF* tragen, um zusätzliche Netzelemente ergänzt. Die Ergänzungen für Transitionen (*$SC*), die sowohl eine *TC* als auch eine *TF* tragen, werden durch Abbildung 5.10 beschrieben.

Jede Transition des ZM-Netzes (*$SC*) wird um die Elemente *$SC.active*, *$SC.finshed* ergänzt. Sobald der Scheduler der Instanz den Steuerfokus (über eine Marke in *.ready*) erteilt, kann *$SC*, sofern *TC* zusammen mit den anderen Vorbedingungen erfüllt ist, schalten. Der Scheduler kann allerdings erst dann einer weiteren Instanz den Fokus erteilen, wenn eine Wartezeit abgelaufen und der Platz *$node.cpuidle* wieder markiert ist. Zusätzlich wird der Platz *.changed* markiert. Er bestimmt, ob zu Beginn der ersten Scheduler-Phase der Zeitstempel der Marke in *.clock* neu gesetzt wird (vgl. Abbildung 5.3). Die Wartezeit (für *$SC.finshed*) beschreibt die Ausführungsdauer des Guards und der Aktionen. Die ermittelten normierten maximalen Bearbeitungszeiten (*maxduration*) werden zuvor in reale maximale Bearbeitungszeiten (*rmaxduration*) überführt. Hierzu werden als Parameter die Rechnerknotengeschwindigkeit (*speedscale*) und die Knotengrundtaktzeit (*ticktime*) des Zielrechnerknotens benötigt:

$$rmaxduration = round(maxduration \cdot speedscale) / ticktime - 0.5) \cdot ticktime$$

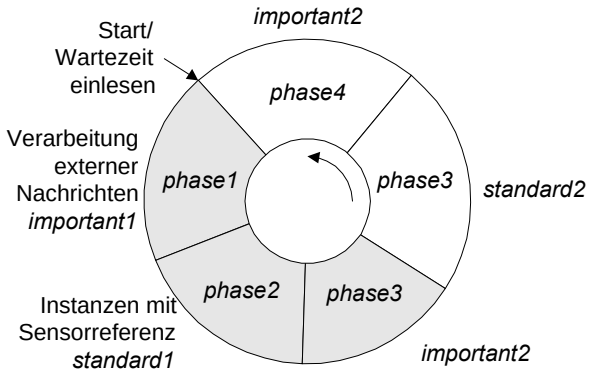
Für Transitionen des ZM-Netzes, die einen Zählplatz aus dem ZM-Netz als Vorplatz besitzen, der gleichzeitig Vorplatz einer weiteren Transition ist, muß deterministisch eine Transition ausgewählt werden. Falls diese Auswahl nicht bereits durch zugeordnete Empfangs-port erfolgt, müssen die beteiligten Guards hinzugezogen werden. Die zusätzlich in diesem Fall angefügten Netzelemente (in Abbildung 5.10 grau unterlegt) beschreiben den sequentielle Test dieser Guards. Die Reihenfolge des Tests wird durch den Prioritätsparameter (*priority.$SC*) des Zustandswechsel bestimmt. Die Funktion *sorted* überführt die beteiligten Prioritätsparameter in eine spezielle Zahlenfolge: der höchste Prioritätsparameter liefert eine eins der nächstniedere eine zwei, usw..

Das Entfernen einer veralteten Nachricht aus einem Empfangspuffer (durch die Transitionen: *clearreceive.$ARP*, *clearreceive.$SSP*, *clearerror.$SSP* und *clearreceive.$SRP*) erfolgt nur, wenn die jeweilige Instanz den Fokus besitzt. Daher werden diese Löschtransitionen ebenfalls ergänzt. Eine Testkante zwischen *.ready* und der jeweiligen Transition gewährleistet, daß aus mehreren Puffern innerhalb einer Abarbeitungsphase veraltete Nachrichten entfernt werden können.

5.2.3.3 Koordinierung mehrerer Instanzen auf einem Rechnerknoten

Die Reihenfolge, in der den Instanzen den Steuerfokus zugeteilt wird, wird durch vier Instanztabellen bestimmt. Die Einträge in den Tabellen *standard1* und *standard2* sind fest vorgegeben; ihre Reihenfolge ändert sich während der Bearbeitung nicht. Die dynamisch nach dem FIFO-Prinzip verwalteten Tabellen *important1* und *important2* sind zu Beginn der Ausführung leer; sie füllen sich zur Laufzeit. Beide Tabellen werden zyklisch geleert, in dem den eingetragenen Instanzen gemäß der dynamisch entstandenen Reihenfolge der Steuerfokus zugeteilt wird.

Abbildung 5.11
Ein Scheduler-Umlauf

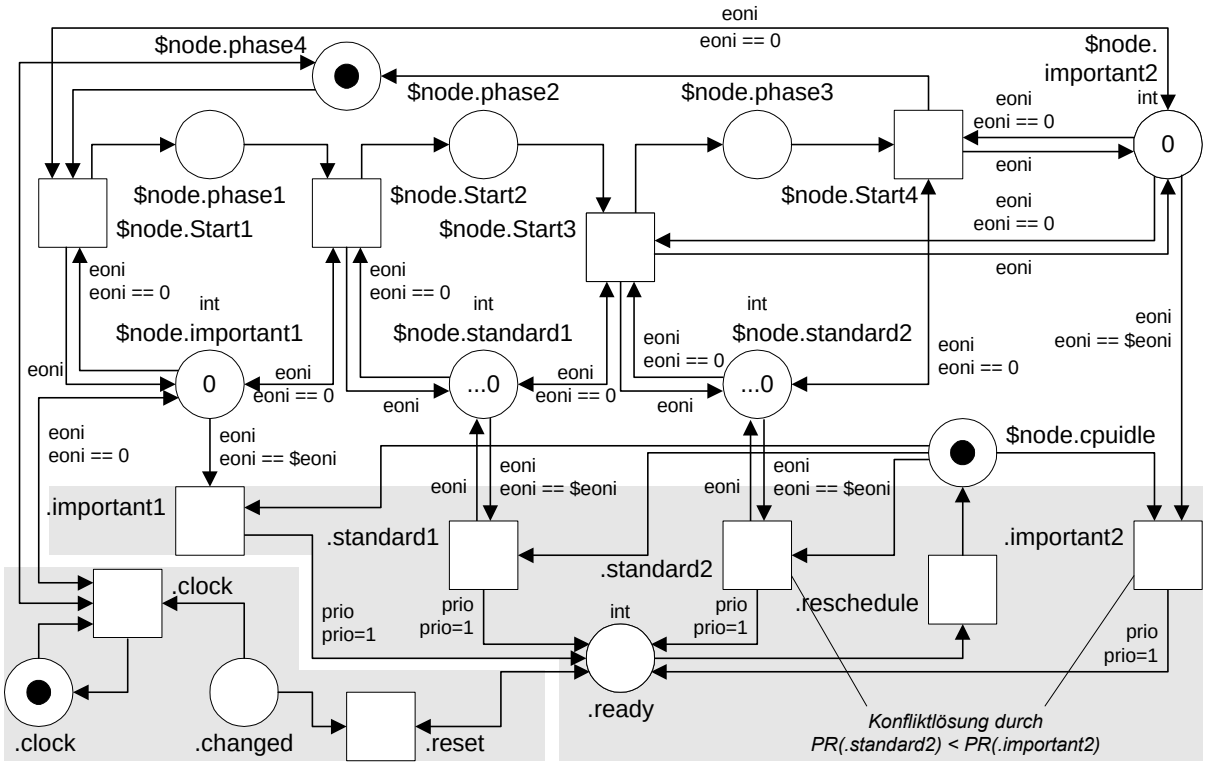


Für die Abarbeitung der Tabellen durchläuft der Scheduler zyklisch vier Phasen (vgl. Abbildung 5.11): In der ersten Phase werden die Empfangsinstanzen, die in der Tabelle *important1* bis zum Beginn der Phase eingetragen wurden, abgearbeitet. Eine Instanz wird in eine dynamisch verwaltete Tabelle eingetragen, wenn sie an einem ihrer *important*-ARPs eine Nachricht erhalten hat (vgl. Abbildung 5.8). In der zweiten Phase (*phase2*) des Scheduler-Umlaufs erhalten alle Instanzen, die in der Tabelle *standard1* stehen, einmal sequentiell den Steuerfokus zugewiesen. Analog wird mit den Instanzen der Tabelle *standard2* während der zweiten Hälfte der dritten Scheduler-Phase (*phase3*) verfahren. In der ersten Hälfte der dritten Phase und in der vierten Phase (*phase4*) werden die Instanzen abgearbeitet, die bis zum Beginn der jeweiligen Phase in der Tabelle *important2* dynamisch eingetragen wurden.

In Tabelle *standard1* stehen alle Instanzen mit einer Sensorreferenz. In Tabelle *standard2* stehen die Instanzen, die nicht *ausschließlich* dynamisch verwaltet werden können. Eine Instanz kann dann ausschließlich dynamisch verwaltet werden, wenn sich alleinig durch den Empfang einer Nachricht an einem ihrer *important*-ARP ein Zustandswechsel aktivieren läßt. Sende- und Empfangsinstanzen sind grundsätzlich den dynamischen Instanzen zuzurechnen; sie lassen sich ausschließlich durch eine Nachricht an ihrem *important*-Port *In* aktivieren. Empfangsinstanzen werden über *important1* und Sendeinstanzen über *important2* verwaltet. Neben den für die Verhaltensbeschreibung hinzugefügten Sendeinstanzen werden über *important2* auch die Anwenderinstanzen dynamisch verwaltet. Wobei Anwenderinstanzen, die sich nicht ausschließlich dynamisch verwalten lassen, *zusätzlich* in der Tabelle *standard2* stehen.

Der detaillierte Ablauf eines Scheduler-Umlaufs wird durch Abbildung 5.12 beschrieben. Neben dem einmal pro Rechnerknoten vorhandenen Petri-Netz, das den Knoten-Scheduler beschreibt, sind die Netzelemente (grau unterlegt) dargestellt, die einmal für jede elementare ON-Instanz existieren. Von den vier dargestellten Transitionen *.important1*, *.standard1*, *.important2* und *.standard2* sind allerdings immer nur jeweils zwei (und zwar solche mit gleicher Endnummer) für eine Instanz vorhanden. Instanzen, die in der ersten und zweiten Schedulerphase abgearbeitet werden, erhalten die Transitionen *.important1* und *.standard1*, die übrigen Instanzen, die Transitionen *.important2* und *.standard2*. Die beiden statischen Instanztabellen werden, durch die beiden Queue-Plätze *$node.standard1* und *$node.standard2* repräsentiert. Als Initialmarkierung besitzen sie eine Folge von Nummern (der Datenklasse *int*), wobei für jede Nummer ($\neq 0$) eine spezielle Instanz steht. Am Ende der Folge steht eine Null; sie markiert das Ende der Tabelle. Die beiden Queue-Plätze *$node.important1* und *$node.important2*, die die beiden dynamischen Tabellen verkörpern, sind initial nur mit einer Null als Endmarke markiert. Alle in Abbildung 5.12 gezeigten Vorkanten erhalten als Wartezeit (*WT*) die Knotengrundtaktzeit *$node.ticktime* zugeordnet.

Abbildung 5.12
Die Petri-Netz-Umsetzung für den Scheduler auf dem Knoten *$node* mit der Instanz *$eoni*



Alle Aktivitäten innerhalb einer Instanz haben den Platz *.ready* als Vorbedingung. Ist er markiert, so besitzt die jeweilige Instanz den Steuerfokus. Der Scheduler markiert diesen Platz, sobald *$node.cpuidle*, der die Verfügbarkeit der Knoten-CPU anzeigt markiert ist und eine der Instanzentabellen eine Marke mit der Nummer der Instanz bereithält. Die in Abbildung 5.11 gezeigten unterschiedlichen Scheduler-Phasen werden dadurch gestartet, daß die Null-Marke, die die jeweilige Tabelle blockiert, vom Anfang der Folge abgezogen und an das Ende wieder eingefügt wird. In der dritten Phase (*$node.phase3*) können die beiden Transitionen *.standard2* und *.important2* im Konflikt stehen. Dieser Konflikt wird durch Priorisierung zugunsten von *.important2* entschieden. Kann keine Transition des ZM-Netzes schalten, so gewährleistet die Transition *.reschedule*, daß der Steuerfokus wieder an den Scheduler zurückgegeben wird. Die Transition *.reschedule* hat grundsätzlich die niedrigste Priorität.

5.2.3.4 Koordinierung über mehrere Rechnerknoten

Für die Abarbeitung auf mehreren Rechnerknoten werden die lokal auf einem Rechnerknoten kommunizierenden Instanzen nach dem im vorhergehenden Abschnitt erläuterten Verfahren koordiniert. Für Instanzen, die mit anderen Rechenknoten kommunizieren, werden zusätzliche Kommunikationsinstanzen, die bereits im Zusammenhang mit den Nachrichtenverbindungen beschrieben wurden (vgl. Abbildung 5.9), eingesetzt. Die dynamische Verwaltung dieser zusätzlichen Instanzen sorgt dafür, daß das zeitliche Verhalten des Rechnerknotens nur dann beeinflußt wird, wenn Nachrichten den Knoten erreichen, bzw. diesen verlassen.

5.2.3.5 Unterschiede bei der Abarbeitung der Umgebungsspezifikation

Die Umgebungsspezifikation ist integraler Bestandteil eines Objektnetz-Projektes. Sie dient der Steuerungsspezifikation als Testszenario. Das Verhalten der Steuerungs- in Verbindung mit der Umgebungsspezifikation wird durch ein gemeinsames Petri-Netz beschrieben. Beide Projektbestandteile werden nach den gleichen Regel der Objektnetz-Methodik entworfen. Ihr zeitliches Verhalten ist allerdings unterschiedlich. Um das zeitliches Verhalten der Umgebungsspezifikation zu beschreiben wird sie einer idealisierten Pseudoplattform zugeordnet. Die Möglichkeit, durch elementare ON-Instanzen nebenläufiges Umgebungsverhalten zu modellieren, wird durch diese Pseudoplattform unterstützt. Die Plattform ist daher wie folgt definiert:

- Für jede elementare ON-Instanz existiert ein virtueller Rechnerknoten.
- Jeder Rechnerknoten hat zu allen anderen Knoten eine Kommunikationsverbindung.
- Alle Rechnerknoten besitzen das gleiche Zeitraster; die Knotengrundtaktzeit (*environment-ticktime*) wird als alleinig variabler Parameter durch die Projekt-Spezifikation vorgegeben. Alle weiteren Zeitparameter werden davon abgeleitet. Aktionen und Guards besitzen keine Abarbeitungszeit. Alle Sende- und Empfangswartezeiten betragen die halbe Knotengrundtaktzeit, da diese im Petri-Netz durch eine gemeinsame Wartezeit repräsentiert werden (vgl. Abbildung 5.9).
- Da Guards und Aktionen in der Umgebung keine Verarbeitungszeiten besitzen, entfallen in Abbildung 5.10 die Netze-Elemente *$SC.test*, *$SC.active* und *$SC.finished*. Konflikte zwischen ZM-Netz-Transitionen, die Vorkanten zu einen gemeinsamen ZM-Netz-Platz besitzen, werden zufällig gelöst. Hierdurch wird nichtdeterministisches Verhalten der Umgebung modelliert.

Durch diese Festlegungen wird gewährleistet, daß parallel in der Umgebung ablaufende Vorgänge adäquat im Modell wiedergegeben werden können. Damit das zeitliche Verhalten der modellierten realen Umgebung (z.B. die Änderungsrate eines realen Meßwertes) ohne Informationsverlust im Umgebungsmodell wiedergegeben wird, muß die Knotengrundtaktzeit klein gegenüber den kleinsten Zeitintervallen (z.B. die Änderungsgeschwindigkeit eines Meßwerts) der modellierten Umgebung sein.

5.3 Die Constraints

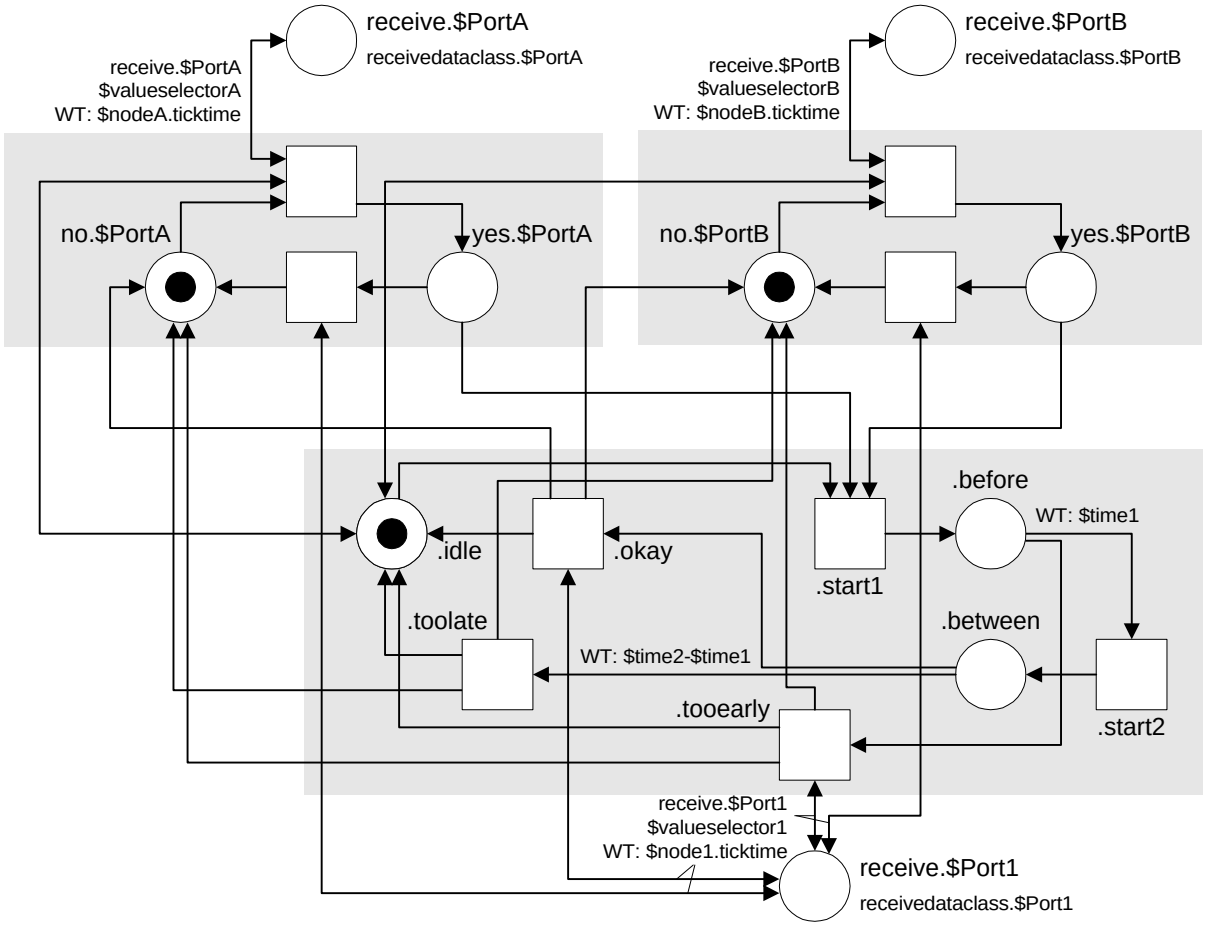
Die in Kapitel 3.3 eingeführte Objektnetz-Constraint-Sprache (ONCL, vgl. Abbildung 3.18) ermöglicht es, auf Klassenebene zeitliche und logische Anforderungen bzw. Zwänge (Constraints) zu formalisieren. Constraints beziehen sich auf Nachrichten, die in den Ports zur Verarbeitung bereitstehen bzw. abgesendet werden. Constraints werden typischerweise den abstrakten Klassen zugeordnet. Im Rahmen der Konkretisierung durch Vererbung werden Constraints an elementare oder hierarchische Klassen vererbt. Objektnetz-Constraints schränken die Art und Weise, wie eine Instanz dieser Klasse mit ihrem Umfeld kommuniziert applikationsspezifisch ein. Es werden

temporal-logische Zusammenhänge zwischen unterschiedlichen Ports einer Klasse formuliert. Constraints werden typischerweise für Klassen der Steuerungsspezifikation definiert.

Die Petri-Netz-Verhaltensbeschreibung bezieht sich auf instantiierte Constraints. Sie wurden an die durch die Entfernung der Hierarchie entstandenen elementaren ON-Instanzen angepaßt (vgl. Kapitel 4). Jeder instantiierte Constraint wird als ein separates Netzfragment dargestellt. Diese Netzfragmente verändern das Verhalten des bisherigen Petri-Netzes nicht. Constraints haben lediglich überwachenden Charakter. Spezielle Transitionen signalisieren, sobald sie Konzession erhalten, eine Verletzung des Constraints.

5.3.1 Empfangsport-Constraints

Abbildung 5.13
Das Transitionsunternetz für Empfangsport-Constraints mit zwei Ports in der Vorbedingung



Ein Empfangsport-Constraint beginnt mit dem Schlüsselwort *receive* und definiert für einen Empfangsport (*$Port1*) eine temporal-logische Beziehung zu einem oder mehreren anderen Empfangsports (z.B. *$PortA* und *$PortB*). Abbildung 5.13 zeigt das korrespondierende Netzfragment für den symbolischen Empfangsport-Constraint: *receive $Port1 $valueselector1 between $time1 $time2 after $PortA $valueselectorA && $PortB $valueselectorB occurred*.

Die Selektoren (*$valueselector1*, *$valueselectorA* und *$valueselectorB*) können optional entfallen. Die Vorbedingung (*precondition*), die zwischen den Schlüsselwörtern *after* und *occurred* steht, erstreckt sich dabei über einen oder mehrere (in Abbildung 5.13 sind es zwei) konjunktiv verknüpfte Empfangsports.

Für jeden Port in der Vorbedingung (z.B. *$PortA*) steht ein getrenntes Netzfragment (grau unterlegt), welches die Erfüllung der jeweiligen Teilvorbedingung eigenständig detektiert. Wurde eine Teilvorbedingung erfüllt, so wird z.B. der Platz *yes.$PortA* markiert. Die Detektierung kann nur stattfinden, wenn der Constraint nicht bereits gestartet wurde, d. h. der Platz *.idle* noch markiert ist. Sind alle Vorbedingungen erfüllt, so startet der Constraint durch die Aktivierung der Transition *.start1*. Die erste Wartezeit (*$time1*) läuft an. Nach deren Ablauf startet (*.start2*) die zweite Wartezeit (*$time2-$time1*). Die Aktivierung der Transition *.okay* signalisiert schließlich die erfolgreiche Beendigung des Constraints; er befindet sich darauf wieder in seiner Ausgangslage.

Die Aktivierung der Transition *.toolate* signalisiert die Verletzung des Constraints aufgrund einer Zeitüberschreitung. Die Transition *.tooearly* signalisiert eine verfrühte „Erfüllung" des Constraints. Vor- und Testkanten ohne explizit dargestellte Wartezeit erhalten als Wartezeit den größten gemeinsamen Teiler der Knotengrundtaktzeiten der beteiligten Rechnerknoten.

5.3.2 Constraints mit Sendeports

Constraints mit Sendeports werden auf die Struktur der Empfangsport-Constraints zurückgeführt. Befinden sich Sendeports in einem Constraint, so werden diese durch einen Empfangsport ersetzt, zu dem der Sendeport eine lokale Nachrichtenverbindung besitzt. Existiert keine lokale Nachrichtenverbindung, so wird der Empfangsport (*In*) der Sendeinstanz (*$sendeoni*) eingesetzt (vgl. Abbildung 5.9). Die weitere Vorgehensweise erfolgt analog zu einem Empfangsport-Constraint.

6. Kapitel

Validierung

Am Ende des Entwicklungsprozesses von Steuerungssoftware steht die Validierung. Diese Stufe innerhalb des Software-Lebenszyklus gewährleistet die korrekte Einhaltung der aufgestellten Anforderungen. Das *Free On-Line Dictionary of Computing* [FOLDOC 98] schreibt zum Begriff *Validation*:

> *„The stage in the software life-cycle at the end of the development process where software is evaluated to ensure that it complies with the requirements."*

Bestimmte Validierungsschritte beruhen auf analytischen Untersuchungen; sie können in der Regel automatisch ablaufen. Sie dienen dazu generelle Anforderungen, die in jeder Applikation zu prüfen sind, analytisch zu validieren. Der Test auf Vollständigkeit ist z.B. dazu zuzählen. Bei analytischer Validierung spricht man auch von *Verifikation*, da hierbei bestimmte Anforderungen analytisch *nachgewiesen* werden

Neben der Verifikation genereller Anforderungen sind für die Validierung applikationsspezifischer Anforderungen, die als Objektnetz-Constraints vorliegen müssen, zwei unterschiedliche Vorgehensweisen möglich. Die erste Möglichkeit ergibt sich durch die sogenannte vollständige Petri-Netz-Simulation, bei der alle erreichbaren Situationen des modellierten Systems durchlaufen werden. Diese Vorgehensweise liefert für ein gegebenes Umgebungsmodell die Verifikation der Constraints. In der Regel muß diese Verifikation jedoch mit einem sehr hohem Rechenaufwand erkauft werden.

Die zweite Möglichkeit bezieht sich auf eine Petri-Netz-Simulation mit Animation der Umgebung. Hierbei werden empirisch ausgewählte Testfälle simuliert, die allerdings keinen Anspruch auf Vollständigkeit erheben. Parallel zur Simulation werden die Auswirkungen auf die Umgebung graphisch animiert. Eine Animation kann dem Auftragegeber die wunschgemäße Umsetzung mit geringem Rechenaufwand plausibel machen. Der Nachteil dabei ist jedoch, daß eine Animation zwar die Nichteinhaltung einer sicherheitskritischen Anforderung (in Form eines Constraints) aufdecken, aber niemals deren Einhaltung garantieren kann. Dies bleibt der vollständigen Simulation vorbehalten.

6.1 Generelle zeitfreie Anforderungen

Eine Reihe von Anforderungen, die eine Objektnetz-Spezifikation erfüllen muß, sind von genereller Natur. Sie müssen durch den Anwender nicht spezifiziert werden, da ihre Einhaltung ohnehin zwingend ist. Sie können durch verschiedene Verfahren vollautomatisch verifiziert werden. Die Verifikationen erfolgen im wesentlichen auf der Basis der Petri-Netz-Verhaltensbeschreibung von Kapitel 5. Die erfolgreiche Durchführung dieser Verifikationen ist Vorbedingung für die Validierung applikationsspezifischer Anforderungen.

6.1.1 Vollständigkeit

Die Vollständigkeit einer Objektnetz-Spezifikation ist Voraussetzung für ihre Implementierung. Eine Objektnetz-Spezifikation wird als vollständig bezeichnet, wenn

- alle beteiligten elementaren ON-Klassen vollständig sind; abstrakte Klassen werden als leere elementare ON-Klassen interpretiert. Mit leeren ON-Klassen, die als unvollständig angesehen werden, kann eine simulative Teilvalidierung erfolgen.
- nach der Auflösung der Hierarchiekonstrukte alle verbleibenden Ports mindestens eine Nachrichtenverbindung besitzen.
- jedes AS-Objekt sowohl von der Steuerung als auch von der Umgebung referenziert wird.
- jede elementare ON-Instanz der Steuerung einem Rechnerknoten zugeordnet wurde.

Eine elementare ON-Klasse ist vollständig, wenn

- alle Zustände, die nicht Endzustand sind, von mindestens einem Zustandswechsel Vorzustand sind und alle Nichtstartzustände von mindestens einem Zustandswechsel als Folge- oder Fehlerfolgezustand referenziert werden.

6.1.2 Konfliktfreiheit

Eine besondere Eigenschaft der Objektnetz-Methodik besteht darin, daß sich Konflikte auf konkurrierende Zustandswechsel innerhalb elementarer ON-Instanzen beschränken. Außerhalb einer elementaren ON-Instanz können keine Konflikte auftreten, da Nachrichten immer exklusiv für eine Empfängerinstanz generiert werden. Durch die Analyse der konfliktträchtigen Konstrukte werden dem Anwender Hinweise auf mögliche Fehler gegeben.

● Analyse konfliktträchtiger Konstrukte in elementaren ON-Klassen:

Konkurrierende Zustandswechsel werden, je nachdem, ob sie in der Steuerung oder der Umgebung auftreten, unterschiedlich behandelt. In beiden Fällen ist es allerdings notwendig, den Anwender über mögliche Konflikte zu informieren. Ob im Einzelfall ein Konflikt zwischen konkurrierenden Zustandswechseln besteht, hängt von den Guards und Portzuordnungen ab.

Abbildung 6.1
Drei konkurrierende Transitionen eines ZM-Netzes mit gemeinsamen Vorplatz *$prestate*

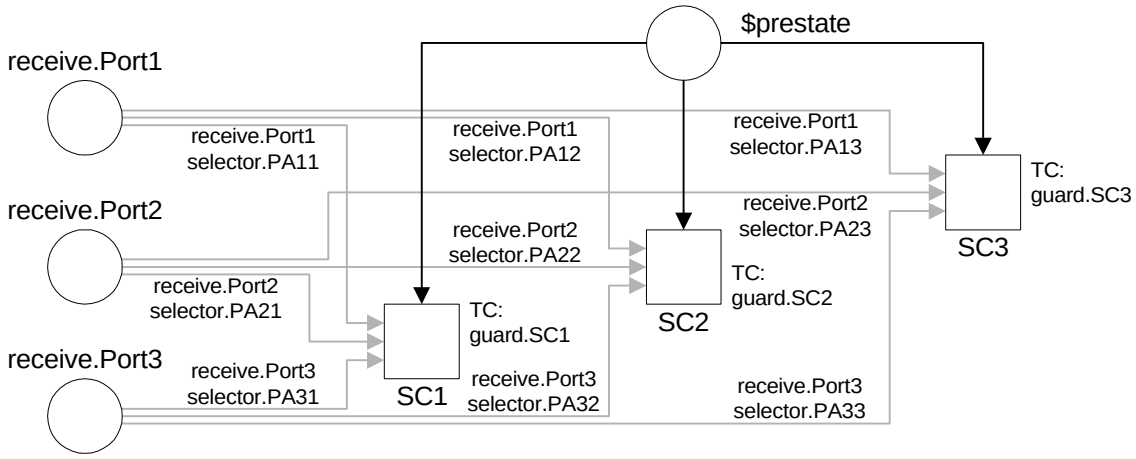


Abbildung 6.1 zeigt das korrespondierende Petri-Netz für drei (allgemein: n) konkurrierende Zustandswechsel, die einen gemeinsamen Vorzustand besitzen und denen bis zu drei (allgemein: m) Empfangsports zugeordnet (graue Vorkanten) wurden. Das gezeigte Konstrukt ist im allgemeinen Fall konfliktfrei, wenn für alle möglichen Wertbelegungskombinationen der Attribute, des ASO-Attributes und der Variablen *receive.Port1*, ... , *receive.Portm* die folgende Bedingung erfüllt ist: Gilt für ein *SCi* der n ZM-Netz-Transitionen mit $i \in \{1, ..., n\}$

$$guard.SCi \wedge \left(\bigwedge_{k=1}^{m} selector.PAki \right) = true,$$

so ist für alle anderen Transitionen *SCj* mit $j \in \{1, ..., n\} \setminus \{i\}$

$$guard.SCj \wedge \left(\bigwedge_{k=1}^{m} selector.PAkj \right) = false,$$

wobei für nichtvorhandene Portzuordnungen die entsprechenden Selektoren *true* gesetzt werden. Für die Verifikation müssen Guards und Selektoren für alle möglichen Belegungen der Variablen und Attribute getestet werden. An dieser Stelle erweist sich eine möglichst starke Einschränkung der Wertbelegungsmenge der beteiligten Datenklassen als hilfreich, da dies die Berechnungszeiten bedeutend verkürzt.

6.1.3 Erreichbarkeit der Zustände

Für eine elementare ON-Klasse ohne Attribute kann die Frage, welche Zustände vom Startzustand aus erreichbar sind, aus der Struktur des ZM-Netzes direkt erschlossen werden. Wurde eine elementare ON-Klasse durch Attribute erweitert, so ist die Frage, welche Belegungen die Attribute in Kombination mit den Zuständen einnehmen können, nicht mehr direkt aus der Struktur des ZM-Netzes erkennbar. Für elementare ON-Klassen mit Attributen müssen zuvor einige Begriffe festgelegt werden:

- Ein *erweiterter* Zustand einer elementaren ON-Klasse bezeichnet eine Kombination bestehend aus einem Zustand – der zu dem erweiterten Zustand korrespondiert – und einer Wertbelegungskombination aller Attribute.
- Der *erweiterte* Startzustand ist die initiale Wertbelegungskombination aller Attribute. Der erweiterte Startzustand einer ON-Klasse mit Parameterports wird auf Instanzebene durch die Parameter an diesen Ports bestimmt.
- Ein *erweiterter* Zustandswechsel bezeichnet den Wechsel von einem erweiterten Zustand zu einem anderen bzw. wieder in den gleichen erweiterten Zustand. Ein solcher erweiterter Zustandswechsel z.B. von *i* nach *j* existiert für eine elementare ON-Klasse unter den folgenden Bedingungen. Vom erweiterten Startzustand der Klasse ist *i* erreichbar ist. Es existiert ein Zustandswechsel von dem zu *i* korrespondierenden Zustand zu dem zu *j* korrespondierenden Zustand, der aktiviert werden kann. Bei der Aktivierung muß dieser die Wertbelegungskombination der Attribute in *i* in die von *j* überführen. Ob diese Überführung existiert, hängt ab von der Belegung des Sensorobjekts, der Empfangspuffervariablen und der Aktion, die der Zustandswechsel referenziert.

- **Erreichbarkeitsgraph einer elementaren ON-Klasse:**

Die Frage, ob ein erweiterter Zustand vom erweiterten Startzustand aus erreichbar ist, kann nun über die Berechnung des Erreichbarkeitsgraphen (EG) beantwortet werden. Die erweiterten Zustände bilden die Knoten des EG. Die erweiterten Zustandswechsel bilden gerichtete Kanten, die die Knoten verbinden. Ein erweiterter Zustand *i* ist vom erweiterten Startzustand aus erreichbar, wenn ein gerichteter Weg existiert, der den Knoten des erweiterten Startzustandes mit dem Knoten von *i* verbindet. Parametrisierbare ON-Klassen besitzen für jede mögliche Parameterbelegung einen eigenen EG, die sich durch unterschiedliche erweiterte Startzustände auszeichnen.

- **Erreichbarkeitsgraph einer elementaren ON-Instanz:**

Die EG der elementaren ON-Instanzen sind alle in dem EG ihrer gemeinsamen Klasse enthalten. Für eine konkrete Instanz sind bestimmte erweiterte Zustandswechsel der elementaren ON-Klasse nicht mehr aktivierbar, wodurch auch bestimmte erweiterte Zustände nicht mehr erreicht werden können.

- **Redundanz:**

Ein Zustand wird als redundant bezeichnet, wenn in dem EG einer Instanz kein erweiterter Zustand erreichbar ist, der mit diesem Zustand korrespondiert. Redundante Zustände deuten auf gravierende Entwurfsfehler hin.

6.2 Generelle zeitliche Anforderungen

Unabhängig von den konkreten Objektnetz-Constraints einer Anwendung müssen die zeitlichen Beziehungen zwischen Steuerung und Umgebung einer Reihe von Forderungen genügen. Die

Beschreibung und Verifikation dieser generellen, zeitlichen Anforderungen bildet den Gegenstand des folgenden Abschnitts.

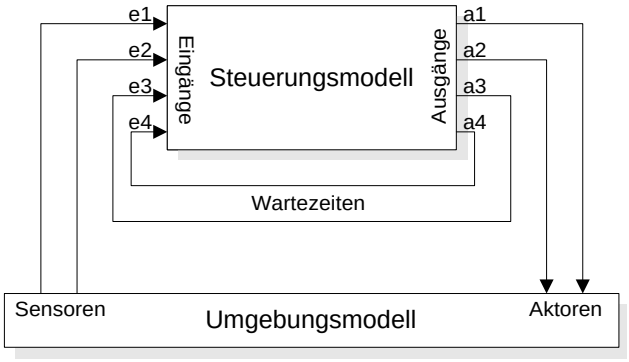
6.2.1 Zeiten in der Steuerung und der Umgebung

Allgemein gilt für eine Steuerung, daß sie möglichst schnell auf jede meßbare Veränderung in der Umgebung reagieren sollte. Da Steuerungen, die auf Digitalrechnern ablaufen, eine gewisse Verarbeitungszeit benötigen, kann ihre Reaktionszeit nicht beliebig kurz und im allgemeinen Fall auch nicht konstant sein. Für eine spezielle Anwendung muß durch den Anwender fest-gelegt werden, welche maximale Reaktionszeit bzw. welche Abtastperiodendauer noch toleriert werden kann. In [Philippow 87, Seite 335] wird für die Abtastperiodendauer 1/16 bis 1/4 der Änderungsrate (bzw. Eigenfrequenz) der Umgebung empfohlen. Die Änderungsrate definiert die kürzeste Periodendauer, die zwischen zwei unterschiedlichen Sensorwerten meßbar ist.

- **Änderungsrate der Umgebung:**

Ein Objektnetz-Modell für eine Steuerung besitzt normalerweise mehrere Eingänge und mehrere Ausgänge (in Abbildung 6.2 vier Ein- und vier Ausgänge). Aufgrund der immer vorhandenen Trägheit der Umgebung kann davon ausgegangen werden, daß für jeden Sensor s eine charakteristische Zeitspanne $T_e(s)$ existiert, die angibt, daß zwischen je zwei meßbaren Änderungen des Sensorwertes eine Zeitspanne von mindestens $T_e(s)$ vergeht. Das kleinste $T_e(s)$ wird als Änderungsrate der gesamten Umgebung T_e bezeichnet. Für jede in der Steuerung definierte Wartezeit wird jeweils ein zusätzlicher „Eingang" zu den Sensoreingängen und ein zusätzlicher „Ausgang" zu den Aktorausgängen hinzugefügt. Die definierte Wartezeit bestimmt dabei die Zeitdifferenz zwischen der Aktivierung des Ausgangs und der Reaktion am Eingang. Ist eine Wartezeit kleiner als T_e, so reduziert sich T_e, so daß sie dieser kleinsten Wartezeit entspricht.

Abbildung 6.2
Steuerungs- und Umgebungsmodell



- **Reaktionszeit der Steuerung:**

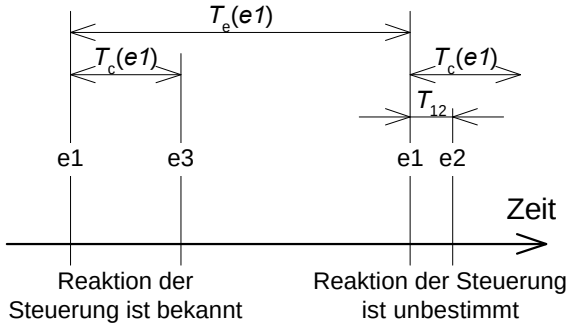
Wenn sich z.B. die Belegung des Eingangs *e1* verändert, so benötigt die Steuerung eine gewisse Zeit, um diese Änderung vollständig zu bearbeiten. Während dieser Reaktionszeit durchläuft die typischerweise verteilte Steuerung diverse interne Zwischenzustände, bevor das erwartete Ergebnis sich einstellt, welches allerdings nicht zwangsläufig eine Änderung an den Ausgängen nach sich ziehen muß. Die Zeitpunkte, an denen die Zwischenzustände erreicht, bzw. wieder verlassen werden, sind aufgrund der asynchronen Kommunikation zwischen den verteilten Rechnerknoten der Steuerung nicht beliebig genau feststellbar. Es kann lediglich für die maximale

Länge der gesamten Raktionszeit ein maximaler Wert $T_c(e1)$ ermittelt werden. Die Kenntnis dieser Schranke ist eine wesentliche Voraussetzung für die Verifikation der Objektnetz-Constraints. Bevor allerdings ihre Berechnung (Kapitel 6.2.3) durchgeführt wird, ermittelt ein separater Verifikationsschritt, ob die Eingänge der Steuerung paarweise voneinander abhängig sind.

6.2.2 Gegenseitige Abhängigkeit von Steuerungseingängen

Für die Betrachtung der gegenseitigen Abhängigkeit von Steuerungseingängen dient das in Abbildung 6.3 dargestellte Szenario. Für das Maximum der Reaktionszeit der Steuerung auf eine Änderung an *e1* wurde $T_c(e1)$ ermittelt. Erfolgt nun während der Zeitspanne von 0 bis $T_c(e1)$ nach der Zeit T_{12}, wobei T_{12} im Bereich von 0 bis $T_c(e1)$ schwankt, eine zusätzliche Änderung an *e2*, so kann nicht generell angenommen werden, daß die Steuerung sich für unterschiedliche T_{12} immer gleich verhält. Bereits kleinste Schwankungen der Verweilzeiten in den Zwischenzuständen können eine eindeutige Zuordnung der zweiten Änderung unmöglich machen. Dies hat zur Folge, daß die Reaktion der Steuerung auf die beiden Ereignisse an *e1* und *e2* nicht mehr eindeutig bestimmbar ist. Der Eingang *e1* ist innerhalb der Zeit $T_c(e1)$ von *e2* abhängig.

Abbildung 6.3
Zeitliche Abfolge externer Ereignisse



Dieses potentiell nichtdeterministische Verhalten einer zu langsamen verteilten Steuerung kann nicht allgemein für alle Eingangspaare (für n Eingänge existieren $(n^2-n)/2$ Paare) ausgeschlossen werden. Es muß jedoch zumindest für die spezielle Anwendung, d. h. für ein spezielles Umgebungsmodell mit speziellen Constraints, ausgeschlossen werden. Hierbei sind zwei unterschiedliche Wege möglich.

Erstens; der Anwender hat durch spezielle Constraints ausgeschlossen, daß die Steuerung diesen kritischen Fall in der Umgebung hervorruft. Als Beispiel wird eine Richtungserkennung mittels zweier Sensoren betrachtet, bei der die Überschreitung einer Grenzgeschwindigkeit, die die beiden Sensorsignale zeitlich zu eng zueinander bringen würde, ausgeschlossen wird. Die beiden Constraints, die den zeitlichen Mindestabstand (*t1*) zweier Empfangsport (*Port1* und *Port2*) fordern, könnten wie folgt lauten: *receive Port1 between t1 t2 after Port2 occurred* und *receive Port2 between t1 t2 after Port1 occurred*.

Zweitens; dieser Fall darf nicht ausgeschlossen werden, weil er z.B. als Fehlerfall durch die Steuerung getrennt behandelt werden muß. Die Steuerung muß sich folglich auch im kritischen Fall deterministisch verhalten. Dies wird dadurch gewährleistet, daß sich beide Sensoren auf dem gleichen Rechnerknoten befinden, und beide Sensoren sich *gleichzeitig* über *eine* Sensormethode auslesen lassen. Die Sensoren müssen hierzu an einem gemeinsamen Peripherie-

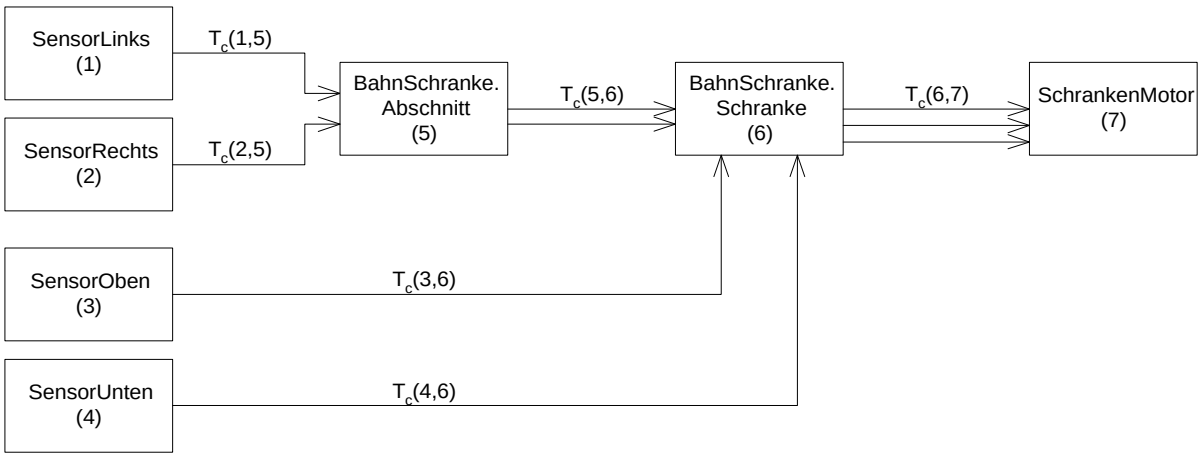
Port des Rechners angeschlossen sein. Analoges gilt, wenn mehrere Steuerungsausgänge sich gleichzeitig aufgrund einer Reaktion in der Steuerung ändern sollen.

Grundsätzlich gilt für zwei *getrennte* Sensoren auf einem Rechner, daß der Scheduler die Zeitspanne T_{12} (vgl. Abbildung 6.3) bestimmt, in der die zeitliche Abfolge der Sensorwertänderungen (*e1* und *e2*) nicht mehr sicher ermittelt werden kann. Es ist die Zeitdauer, die benötigt wird, um alle Instanzen mit einer Sensorreferenz einmal zu bearbeiten.

- **Reaktionskette und Reaktionsgraph:**

Die Feststellung der bereits in Abbildung 6.3 dargestellten Abhängigkeit zweier Steuerungseingänge e_i und e_j erfolgt anhand des sogenannten Reaktionsgraphen (RG). Für die Verifikation der gegenseitigen Unabhängigkeit zweier Steuerungseingänge e_i und e_j werden die Reaktionsgraphen $RG(e_i)$ und $RG(e_j)$ beider Eingänge überlagert (vgl. Abbildung 6.4).

Abbildung 6.4
Die Überlagerung aller vier Reaktionsgraphen der Bahnschrankensteuerung



Jede elementare ON-Instanz, die externe Ereignisse, wie Wartezeiten und Sensorsignale verarbeitet, ist Ausgangspunkt einer Reaktionskette. Entlang dieser Kette pflanzt sich die Reaktion über die Nachrichtenverbindungen von Instanz zu Instanz fort, bis die Steuerung ihren neuen Zustand erreicht hat. Es besteht dabei an jeder Instanz die Möglichkeit, daß die Reaktionskette sich in mehrere Äste aufspaltet; die Reaktionskette verwandelt sich in einen Reaktionsgraphen (RG). Die Instanzen bilden die Knoten, die Nachrichtenverbindungen die Kanten des Graphen, die gerichtet vom ASP zum ARP verlaufen. Synchrone Nachrichten-verbindungen finden sich als bidirektionelle Kanten im RG wieder. Rückführungen auf schon durchlaufene Knoten (sog. Kreise), werden als Entwurfsfehler behandelt und abgewiesen. Berechnungen mit rekursiven Charakter müssen auf der Ebene der Aktionen und Guards gekapselt bleiben. Eine bidirektionelle Kante wird dabei nicht als Kreis gewertet.

Die gegenseitige Unabhängigkeit zweier Eingänge wird wie folgt bestimmt: Die Instanz des Knotens, an dem die beiden überlagerten Reaktionsgraphen sich erstmalig treffen, muß die beiden Nachrichten/Ereignisse, die aus den beiden Graphen stammen, immer unabhängig voneinander

verarbeiten. Eine solche unabhängige Behandlung zweier Nachrichten erfolgt, wenn zwei Bedingungen erfüllt werden. Erstens; die eine Nachricht muß auch ohne die andere verarbeitet werden können. Zweitens; das Ergebnis bleibt unverändert, wenn die beiden Nachrichten in unterschiedlicher Reihenfolge durch die Instanz verarbeitet werden. Werden die beiden Nachrichten am gleichen Empfangsport empfangen, so sind die Nachrichten, sofern es sich um Steuernachrichten handelt, nicht unterscheidbar. Ihre Behandlung muß daher zwangsläufig unabhängig voneinander erfolgen.

Ebenso erfolgt eine unabhängige Behandlung für eine Nachricht und das Ablaufen einer Wartezeit, wenn der Empfangsport keinem Zustandswechsel zugeordnet wurde, der mit dem Zustandswechsel, an dem sich die Wartezeit befindet, konkurriert.

In Abbildung 6.4 entscheidet z.B. die Instanz (6) (*BahnSchranke.Schranke*) über die Unabhängigkeit von *SensorLinks* und *SensorOben*. Für das deterministische Verhalten der Steuerung muß in der Realisierung gewährleistet werden, daß die beiden Nachrichten von *SensorLinks* und *SensorOben* nur mit einem bestimmten zeitlichen Mindestversatz voneinander entstehen können. Dieser Mindestversatz wird in diesem Fall durch das Maximum von $T_c(1, 5) + T_c(5, 6)$ und $T_c(3, 6)$ bestimmt. Die innerhalb dieser Zeiten als abhängig erkannten Steuerungseingänge müssen bei der weiteren Validierung durch zusätzliche, automatisch generierte Constraints überwacht werden. Diese müssen gewährleisten, daß ein minimaler zeitlicher Abstand zwischen den Änderungen an diesen Eingängen nicht unterschritten wird. Anders ausgedrückt heißt dies; die Steuerung muß ausreichend schnell sein, damit die Änderungen getrennt verarbeitet werden können. Um zu überprüfen, ob dies möglich ist, müssen die konkreten maximalen Reaktionszeiten der Steuerung ermittelt werden.

6.2.3 Berechnung der maximalen Reaktionszeit

Die Berechnung der maximalen Reaktionszeit der Steuerung erfolgt für alle Eingänge einzeln. Ein Eingang e_i (vgl. Abbildung 6.2) ist dabei Ausgangspunkt eines Reaktionsgraphen $RG(e_i)$. Die größte Zeitspanne, die vergehen kann, um von einer Wurzel e_i beginnend, ein beliebiges Blatt im $RG(e_i)$ zu erreichen, bestimmt die maximale Reaktionszeit $T_c(e_i)$ für den Eingang e_i. Blätter sind dabei Knoten, die keine wegführenden Kanten besitzen.

Die Berechnung der maximalen Reaktionszeit erfolgt indem entlang eines Bogenzuges in Kantenrichtung von der Wurzel bis zu einem Blatt die Maximalzeiten an den Kanten addiert werden. Nach der Überlagerung mehrerer Reaktionsgraphen können zwischen zwei Instanzen mehrere Kanten bestehen (in Abbildung 6.4 z.B. drei Kanten zwischen Instanz (6) und (7)). Diese Kanten erhalten einen gemeinsamen Wert zugeordnet, der durch die größte Maximalzeit der einzelnen Kanten bestimmt wird. Bidirektionelle Kanten erhalten für jede Richtung einen Wert. Die größte für eine Wurzel e_i ermittelbare Summe bestimmt die maximale Reaktionszeit $T_c(e_i)$. Für den ersten Eingang in Abbildung 6.4 ergibt sich die maximale Reaktionszeit wie folgt: $T_c(1) = T_c(1, 5) + T_c(5, 6) + T_c(6, 7)$. Die größte Reaktionszeit aller Eingänge bestimmt die maximale Reaktionszeit T_c der Steuerung.

- **Lokale Verbindungen:**

Die maximale Zeit $T_c(i, j)$, die benötigt wird um eine Reaktion von einer Instanz *i* an eine Instanz *j* (auf den gleichen Rechnerknoten) weiterzuleiten, hängt zum großen Teil vom Zeitregime des Schedulers ab. Grundsätzlich wird dabei die Zeitspanne betracht, die zwischen dem Beginn der Bearbeitung des auslösenden Ereignisses in der Instanz *i* und dem Beginn der Bearbeitung des generierten Ereignisse in der Instanz *j* vergeht. Für alle lokalen Fälle ist $T_c(i, j)$ immer kleiner gleich der maximalen Scheduler-Umlaufzeit T_s.

- **Die maximale Bearbeitungszeit T_i einer elementaren ON-Instanz:**

Die maximale Bearbeitungszeit T_i einer elementaren ON-Instanz *j* (auf *$node*) ermittelt sich gemäß den Ausführungen von Kapitel 5.2.3 wie folgt: $T_i = 2 \cdot \$node.ticktime + T_{sc}$. T_{sc} ist hierbei die maximale Bearbeitungszeit für einen beliebigen aktivierbaren Zustandswechsel *k*. Wenn kein Zustandswechsel aktivierbar ist, ergibt sich mit dem Zweifachen der Knotengrundtaktzeit (*ticktime*) die Bearbeitungszeit, die benötigt wird, den Fokus der Instanz zuzuteilen und ihn ihr wieder zu entziehen. Die maximale Bearbeitungszeit $T_{sc}(k)$ eines ausgewählten Zustands-wechsels *k* wird wie folgt (vgl. Abbildung 5.10) ermittelt:

- $T_{sc}(k)$ = *$node.ticktime* + *rmaxduration*(*$action*) + *rmaxduration*(*$guard*), wenn *k* die Aktion *$action* und den Guard *$guard* referenziert und dabei keinerlei konkurrierende Zustandswechsel besitzt. Wenn *k* keine Aktion referenziert, entfällt der zweite Summand. Referenziert *k* keinen Guard, so entfällt der dritte Summand.
- Besitzt der Zustandswechsel *k* konkurrierende Zustandswechsel, die höher priorisiert sind als *k*, so ist der zusätzliche Zeitbedarf für den sequentiellen Test dieser Zustandswechsel zu berücksichtigen. Die maximale Dauer für den Test eines einzelnen Zustandswechsels *l* mit einem Guard *$guard* beträgt *$node.ticktime* + *rmaxduration(*$guard*)*. Der zweite Summand entfällt, wenn *l* keinen Guard referenziert.
- Für eine Kommunikationsinstanz (vgl. Abbildung 5.9) existiert nur ein einziger Zustandswechsel. Seine maximale Bearbeitungszeit T_{sc} hängt von der speziellen Kommunikationsschnittstelle *$cio* des Rechnerknotens *$node* und der Datenklasse *$dc* der verarbeitenden Nachrichten ab. Bei einer Sendeinstanz beträgt die maximale Bearbeitungszeit *sendtime*(*$node.$cio, $dc*) und bei einer Empfangsinstanz *receivetime*(*$node.$cio, $dc*).

- **Die maximale Scheduler-Umlaufzeit T_s:**

Die maximale Scheduler-Umlaufzeit T_s eines Rechnerknotens (*$node*) berechnet sich aus der Summe der maximalen Bearbeitungszeiten T_{s1}, ..., T_{s4} aller vier Scheduler-Phasen (*phase1* ... *phase4* in Abbildung 5.11). Für die Berechnung dieser Maximalzeiten, muß zum einen das spezifische Verhalten des Schedulers (siehe Kapitel 5.2.3) berücksichtigt und zum anderen die jeweils maximale Bearbeitungszeit $T_i(j)$ einer beliebigen Instanz *j* bestimmt werden. Die

Bearbeitungszeit einer Instanz beschreibt die Zeitspanne, die zwischen der Vergabe des Steuerfokus an diese Instanz bis zur Weitergabe an die nächste Instanz vergeht.

- **Die Maximalwerte T_{s1}, ..., T_{s4} für die vier Scheduler-Phasen:**

In der ersten Phase (vgl. Tabelle 6.1) des Scheduler-Umlaufs werden die dynamisch (in *important1*) verwalteten Empfangsinstanzen abgearbeitet. Die Anzahl der Nachrichten, die ein Rechnerknoten während des letzten Scheduler-Umlaufs, bis zum Beginn der ersten Phase, von anderen Rechnerknoten empfangen hat, bestimmt die Bearbeitungszeit für diese Phase. Der Maximalwert T_{s1} wird erreicht, wenn für jeden externen Sendeport, der über eine Nachrichtenverbindung mit dem Rechnerknoten verbunden ist, genau eine Nachricht angenommen wird. Durch die Forderung nach Kreisfreiheit für den Reaktionsgraphen, ist diese Annahme gerechtfertigt. Steht ein Rechnerknoten nun mit *m* solchen externen Sendeports in Verbindung, so berechnet sich die maximale Bearbeitungszeit T_{s1} der ersten Phase durch Summation der maximalen Bearbeitungszeiten T_i der *m* Empfangsinstanzen, die den Empfang dieser *m* Nachrichten durchführen.

Tabelle 6.1
Die vier Phasen eines Scheduler-Umlaufs

Phase1	Phase2	Phase3 (Teil 1 und 2)	Phase4
Empfangs-instanzen (*important1*)	Instanzen mit Sensorreferenz (*standard1*)	Stat. verwaltete Inst. ohne Sensorref. (*standard2*) und dyn. Inst., die in Phase1, Phase2 und im ersten Teil von Phase3 adressiert wurden (*important2*)	Dyn. verwaltete Instanzen, die im zweiten Teil der Phase3 adressiert wurden (*important2*)

In der zweiten Phase werden die (in *standard1* verwalteten) Instanzen mit einer Sensorreferenz abgearbeitet. Im Gegensatz zur ersten Phase ist hier die Anzahl der Instanzen in jedem Umlauf gleich. Die Berechnung der maximalen Bearbeitungszeit T_{s2} der zweiten Phase erfolgt durch die Summation der maximalen Bearbeitungszeiten T_i aller Instanzen mit Sensorreferenz.

In der dritten Phase werden die Instanzen der Tabelle *standard2* und die dynamisch (über *important2*) verwalteten Instanzen abgearbeitet. Analog zur zweiten Phase ist dabei die Anzahl der Instanzen in *standard2* konstant. Die Anzahl der dynamisch verwalteten Instanzen ist dagegen variabel. Jede Instanz, die einen *important*-ARP besitzt, mit Ausnahme der Sende-instanzen, kann ein- oder mehrfach dynamisch in *important2* eingetragen werden. Die Maximal-zahl wird erreicht, wenn für jede Nachrichtenverbindung, die einen ihrer *important*-ARPs mit einer statischen oder einer Sendeinstanz verbindet, genau eine empfangene Nachricht angesetzt wird. Die Beschränkung auf eine Nachricht ist durch die bereits Kreisfreiheit gerechtfertigt. Die maximale Bearbeitungszeit T_{s3} für Phase3 ergibt sich durch Summation der maximalen Bearbeitungszeiten T_i der Instanzen in *standard2* und der maximalen Bearbeitungszeiten T_i der Instanzen, die im Maximalfall in *important2* dynamisch eingetragen werden können.

Jede Instanz mit einem *important*-ARP kann in der vierten Phase erneut ein- oder mehrfach (dynamisch über *important2* verwaltet) bearbeitet werden. Die Maximalzahl der in dieser Phase verwalteten Instanzen wird erreicht, wenn für jede Nachrichtenverbindung an ihren *important*-ARPs, die zu einer im zweiten Teil von Phase3 dynamisch verwalteten Instanzen geht, genau eine Nachricht (Kreisfreiheit vorausgesetzt) angesetzt wird. Die maximale Bearbeitungszeit T_{s4} für Phase4 ergibt sich durch Summation der maximalen Bearbeitungszeiten T_i dieser Instanzen.

- **Schnellere lokale Verbindungen:**

Für spezielle Verbindungen können kleinere obere Schranken für $T_c(i, j)$ ermittelt werden:

- $T_c(i, j) = T_{s2} + T_{s3}$ gilt, wenn i eine Instanz mit Sensorreferenz bezeichnet und j eine beliebige andere lokale Instanz.
- $T_c(i, j) = T_{s3}$ gilt, wenn i eine statisch verwaltete Instanz ohne Sensorreferenz und j eine lokale Instanz, die über einen *important*-ARP mit i verbunden ist, bezeichnet.

- **Verbindungen zwischen Instanzen unterschiedlicher Rechnerknoten:**

Für eine Verbindung zwischen zwei Instanzen i und j, die sich auf unterschiedlichen Rechnerknoten (i auf *Node1* und j auf *Node2*) befinden, setzt sich die maximale Zeit $T_c(i, j)$ für die Weiterleitung einer Reaktion aus mehreren Summanden zusammen. Auf *Node1* fällt als Maximum die Zeit $T_{s2}(Node1) + T_{s3}(Node1) + T_{s4}(Node1)$ an. Auf *Node2* fällt maximal die Zeit $T_s(Node2) + T_{s2}(Node2) + T_{s3}(Node2)$ an. Zu diesen Summen sind noch die entsprechenden Sende- und Empfangswartezeiten hinzuzurechnen (vgl. Abbildung 3.40). Sendet die Instanz i über die Kommunikationsschnittstelle *CIO1* von *Node1* Nachrichten der Datenklasse *$dc* und werden diese von der Kommunikationsschnittstelle *CIO1* von *Node2* entgegengenommen, so ergibt sich schließlich die maximale Zeit wie folgt:

$$T_c(i, j) = T_{s2}(Node1) + T_{s3}(Node1) + T_{s4}(Node1) + sendwait(Node1.CIO1, \$dc) + receivewait(Node2.CIO1, \$dc) + T_s(Node2) + T_{s2}(Node2) + T_{s3}(Node2).$$

Falls i eine Instanz ohne Sensorreferenz bezeichnet, entfällt der Summand $T_{s2}(Node1)$.

6.3 Einschränkungen und verbotene Konstrukte

Im Rahmen der Validierung werden für einige Objektnetz-Konstrukte Einschränkungen formuliert. Ohne diese Einschränkungen ließen sich die Puffer nicht beschränken; ebenso wäre das zeitliche Verhalten der Steuerung nicht vollständig beschreibbar. Eine Verifikation wäre entweder unmöglich bzw. aufgrund einer zu hohen Komplexität nicht handhabbar.

6.3.1 Elementare ON-Klassen mit ASO-Referenz

ON-Instanzen mit Sensorreferenz müssen auf Änderungen ihres Sensorwertes unmittelbar reagieren. Folglich dürfen Klassen dieser Instanzen, und nur diese Klassen, automatische

Zustandswechsel enthalten. Die Aktivierung dieser Zustandswechsel hängt alleinig von einem Guard, der eine Sensormethode aufruft, ab. Es ist aber unzulässig, zwei solcher Zustands-wechsel unmittelbar nacheinander zu aktivieren (bzw. anzuordnen). Der zweite *muß* dann eine Wartezeit besitzen. Durch diese Einschränkung wird gewährleistet, daß die Instanz nicht häufiger, als durch die Wartzeit spezifiziert, Ereignisse generieren kann. Die Abtastperioden-dauer für den Sensor wird somit durch diese Wartezeit bestimmt und bleibt von der Ver-arbeitungsgeschwindigkeit des Rechnerknotens unabhängig. Des weiteren dürfen ON-Klassen mit Sensorreferenz nur ASPs besitzen. Durch diese Einschränkungen wird gewährleistet, daß eine Sensorabfrage nicht vom Empfang zusätzlicher Nachrichten abhängt.

6.3.2 Empfangspuffer

Versendete Nachrichten werden grundsätzlich durch die empfangende Instanz gepuffert. Dabei können die Empfangspuffer prinzipiell mehrere Nachrichten zwischenspeichern. Für die korrekte Funktion des Systems muß aber unter allen Umständen sichergestellt werden, daß eine bestimmte Anzahl an Nachrichten in den Puffern nicht überschritten wird. Da sich über die Petri-Netz-Theorie die Puffer asynchron kommunizierender Zustandsmaschinen nicht generell beschränken lassen, müssen zusätzliche Einschränkungen formuliert werden, damit dies dennoch gewährleistet werden kann.

- **Asynchrone Empfangsports mit mehreren Nachrichtenverbindungen:**

Sind mehrere Nachrichtenverbindungen an einem Empfangsport angeschlossen, so bedeutet dies, daß verschiedene Instanzen ein und dieselbe Aktivität in der empfangenden Instanz anstoßen können. Die sendenden Instanzen können dies in der Regel auch nebenläufig tun. Die Reihenfolge ist in einem verteilten System nicht fest. Sind die am Port empfangenen Nachrichten nun zusätzlich inhaltlich nicht unterscheidbar, so müssen diese Nachrichten auch jederzeit unabhängig von zuvor verarbeiteten Nachrichten behandelt werden können. Steuernachrichten sind z.B. inhaltlich nicht unterscheidbar.

Die geforderte unabhängige Behandlung von mehreren Nachrichten eines Ports wird dadurch gewährleistet, daß erstens: die Nachrichten hinreichend lange gepuffert werden und zweitens: die Instanz Nachrichten dieses Ports zyklisch verarbeitet. Ob diese Forderung eingehalten wird, kann mit Hilfe des Situationsgraphens festgestellt werden.

- **Pufferzeiten:**

Falls eine Nachricht eine Zustandsänderung in der Steuerung bewirken soll, so muß sie auch innerhalb der Reaktionszeit (der Steuerung) durch die empfangende Instanz verarbeitet werden. Hat die Nachricht unter bestimmten Umständen keine Bedeutung, weil sie z.B. das Schließen eines bereits geschlossenes Ventil auslösen soll, so muß die Nachricht innerhalb der Reaktionszeit gelöscht werden. Andernfalls würde der Puffer für die Speicherung einer Zustandsänderung der Steuerung zweckentfremdet werden. Diese Aufgabe übernehmen

ausschließlich Zustände und Attribute; Puffer dienen lediglich dazu, *geringe* Zeitverzögerungen zwischen den asynchron kommunizierenden Instanzen einer verteilten Steuerung auszugleichen.

Allgemein betrachtet ist diese Zweckentfremdung unmöglich, wenn die Pufferzeiten der Empfangsports nicht größer als $T_c(e_i)$ sind, wobei e_i das Ereignis ist, aufgrund dessen der Port eine Nachricht erhielt. Für einen speziellen Fall kann eine größere Pufferzeit als unproblematisch betrachtet werden, wenn aufgrund der Struktur der elementaren ON-Klasse ein längeres bzw. unbegrenztes Puffern für einen Empfangsport ausgeschlossen ist.

Aufgrund dieser Überlegungen ergeben sich äußerst restriktive Einschränkungen für die Auswahl der Pufferzeiten (nur 0 oder unendlich), was eine einfache Implementierung ohne Timer und mit Puffern fester Länge ermöglicht.

- **Kapazität der Empfangspuffer:**

Durch die Einschränkungen für die Puffer, der Kreisfreiheit des Reaktionsgraphen und den Einschränkungen für Instanzen mit Sensorreferenz ergibt sich die obere Grenze für die Pufferbelegung aus der Anzahl der angeschlossenen Nachrichtenverbindungen.

6.4 Vollständige Petri-Netz-Simulation

Nach der Verifikation genereller Anforderungen kann die Verifikation applikationsspezifischer Anforderungen, die in Form von Objektnetz-Constraints vorliegen, durchgeführt werden. In der sogenannten vollständigen Petri-Netz-Simulation wird die Einhaltung der Constraints für die jeweilige Umgebung formal nachgewiesen. Für sicherheitskritische Anwendungen kann ein solcher Nachweis zwingend notwendig sein, wodurch sich auch der sehr hohe Rechenaufwand rechtfertigt.

- **Unterschiede zu einer gewöhnlichen Petri-Netz-Simulation:**

Objektnetz-Constraints sind Bestandteil einer Steuerungsspezifikation. Sie gelten bezüglich einer gewählten Objektnetz-Umgebungsspezifikation als verifiziert, wenn sie während einer vollständigen Petri-Netz-Simulation eingehalten werden. Die Petri-Netz-Simulation bezieht dabei die Steuerung, die Umgebung und die Constraints ein. Einhalten heißt, daß die Transitionen *.tooearly* und *.toolate* der Constraints-Unternetze (vgl. Abbildung 5.13) während der Simulation nie Konzession erhalten. Vollständige Simulation bedeutet, daß jeder Knoten des in Kapitel 5.1 eingeführten Situationsgraphen mindestens einmal erreicht werden muß.

Eine vollständige Simulation, die ohne Animation und Nutzerinteraktion abläuft, unterscheidet sich in den folgenden Punkten von einem gewöhnlichen Simulationslauf:

- Der Situationsgraph wird während der Simulation aufgebaut.
- Erreicht die Simulation eine konfliktbehaftete Situation, so werden alle alternativen Wege im Rahmen einer Tiefensuche [WalNäg 87] nacheinander durch den Simulator begangen. Im Gegensatz zur Steuerungsspezifikation bleiben konkurrierende Zustandswechsel in der

Umgebungsspezifikation konfliktbehaftet. Ein auftretender Konflikt drückt den Nichtdeterminismus der Umgebungsspezifikation aus. Im Situationsgraphen ist ein Konflikt an einem Knoten mit mehreren wegführenden Kanten erkennbar.

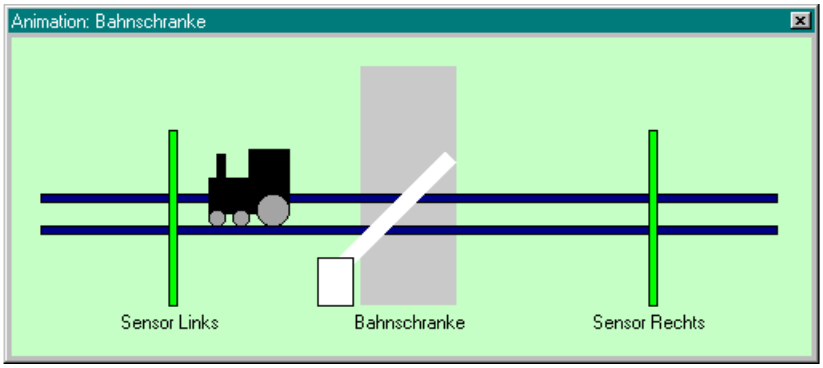
- Erreicht der Simulator eine solche konfliktbehaftete Situation, so werden alle alternativen Folgesituation ermittelt. Diese alternativen Folgesituation dienen im späteren Verlauf als Ausgangspunkt für die Fortführung einer unterbrochenen Simulation.
- Sobald eine bereits durchlaufene Situation oder eine zu dieser vergleichbare Situation erreicht wird, wird die Simulation unterbrochen und an einer bisher nicht durchlaufenen alternativen Folgesituation fortgesetzt. Die Fortsetzung an einer alternative Folgesituation erfolgt ebenfalls, wenn eine Situation (i, s) erreicht wird, bei der das OPN (ohne Zeiten) mit der Markierung $m_{(i, s)}$, die der Situation (i, s) entspricht, nicht mehr schaltfähig ist.
- Die vollständige Simulation ist beendet, wenn alle alternativen Folgesituation durchlaufen wurden.

6.5 Petri-Netz-Simulation mit Animation der Umgebung

Neben der vollständigen Petri-Netz-Simulation kann bereits durch eine gewöhnliche Petri-Netz-Simulation, bei der empirisch ausgewählte Testfälle durchlaufen werden, eine Validierung der prinzipiellen Funktion einer Steuerung durchgeführt werden.

Erfahrungen des Autors zeigten jedoch, daß eine Simulation, die ihre Ergebnisse ausschließlich auf der Abstraktionsebene der Objektnetze präsentiert, selbst für die Validierung einfachster Anforderungen nicht ausreicht. Typischerweise kann eine komplexe und verteilte Funktionalität nicht über die Beobachtung des Nachrichtenflusses zwischen den Instanzen bzw. der Zustandswechsel in den Instanzen erschlossen werden. Spätestens, wenn der Auftraggeber, der in der Regel nicht mit der Objektnetz-Methodik vertraut ist, eine Funktion abnehmen soll, werden anschaulichere Darstellungen benötigt. Erst durch eine mit der Simulation gekoppelten Animation der gesteuerten Umgebung auf einem Abstraktionsniveau, welches der Auftraggeber mit seinen bereichsspezifischen Kenntnissen durchschaut, ist eine Validierung der prinzipiellen Funktion einer Steuerung möglich.

Abbildung 6.5
Tk-Canvas zur Animation des Bahnschrankenbeispiels



Auf die Steuerung einer Bahnschranke übertragen heißt dies, daß eine Animation benötigt wird, die anschaulich zeigt, wie die Steuerung, auf die Ein- bzw Ausfahrt eines Zuges, mit dem Schließen bzw. Öffnen der Schranke reagiert.

Abbildung 6.6
Teil des Aktionscodes zur Initialisierung der Bahnschrankenanimation

```
Tcl_AnimationInit Bahnschranke {
  -bg lightgreen -width 500 -height 200}
Tcl_AnimationCode Bahnschranke {
 ...
 $canvas create rect 25 90 75 112 -fill black -tags Train
 $canvas create rect 50 72 75 90 -fill black -tags Train
 $canvas create rect 30 75 35 90 -fill black -tags Train
 $canvas create oval 25 110 35 120 -fill darkgray -tags Train
 $canvas create oval 40 110 50 120 -fill darkgray -tags Train
 $canvas create oval 55 100 75 120 -fill darkgray -tags Train
}
```

Die Objektnetz-Methodik bietet dem Anwender die Möglichkeit, die Simulation mit einer graphischen Animation zu koppeln. Über die vordefinierte Tcl-Prozedur *Tcl_AnimationInit* kann er ein Animationsfenster öffnen, das ein sogenanntes Tk-Canvas (Zeichenfläche) enthält. Auf dieser Tk-Zeichenfläche lassen sich über spezielle Canvas-Kommandos, die in die Guards bzw. Aktionen der Umgebungsspezifikation eingebaut werden, verschiedene Objekte, wie z.B. Polygone, Linien, Ovale, Texte oder Fenster mit Knöpfen plazieren bzw. manipulieren. Abbildung 6.6 zeigt exemplarisch die Initialisierung des Animationsfensters und der Canvas-Objekte, die den Zug (von Abbildung 6.5) darstellen. Auf die Animation des Dampfes wird dabei verzichtet. Eine ausführliche Beschreibung der Tk-Canvas-Kommandos, die innerhalb der Tcl-Prozedur *Tcl_AnimationCode* übergeben werden können, findet sich in [Ousterhout 95] oder [Welch 97].

Die typischerweise automatisch ablaufenden Animationen können durch zusätzlich eingekoppelte Tk-Steuerelemente (z.B. *Buttons*, *Scaler* oder *Entries*) interaktiv gestaltet werden. Der Anwender wird über diese Steuerelemente gewissermaßen ein Teil der Umgebung. Durch die Betätigung der Steuerelemente beeinflußt er zu Beginn (Auswahl eines Testfalls) oder während der Animation das Verhalten der modellierten Umgebung. Im zweiten Demonstrationsbeispiel des Anhang wird gezeigt, wie der Datenwert einer *double*-Nachricht über einen Tk-*Scaler* im Bereich von 0.0 bis 10.0 verändert werden kann. Die Einkopplung dieses Steuerelements erfolgt über einen Guard, der den aktuellen Wert des Scalers ausliest und in den Sendepuffers eines Ports überträgt.

7. Kapitel

Implementierung

Der folgende abschließende Abschnitt dient der Darstellung der automatischen Code-Generierung. Auf die vollständige Darstellung des Code-Generators wurde zugunsten einer leichten Verständlichkeit des Grundprinzips verzichtet. Seine Darstellung erfolgt anhand des Bahnschrankenbeispiels. Die Umsetzung erfolgt für ANSI-C [Hickersberger 95], da ANSI-C-Cross-Compiler praktisch für alle Plattformen verfügbar sind. Der Einsatz eines C++-Compilers [Stroustrup 98] läge wegen des objektorientierten Aufbaus einer Objektnetz-Spezifikation näher, scheidet aber zugunsten einer allgemeingültigeren Lösung, die auch auf kleinsten Mikrocontrollern ablauffähig ist, aus. Für leistungsstärkere Controllersysteme stellt C++ unter Beachtung bestimmter in [Gruber 95] aufgeführter Regeln eine sinnvolle Alternative dar. Ebenso kann eine Implementierung mit dem neu entwickelten Embedded C++ [GREENHILL 97] erfolgen.

7.1 Umsetzung elementarer ON-Klassen

Ausgangspunkt für die Code-Generierung bildet die Objektnetz-Spezifikation der Steuerung (ohne Vererbungs- und Hierarchiekonstrukte) in Verbindung mit der Plattformspezifikation und den Mapping-Informationen.

7.1.1 Grundgerüst für elementare ON-Klassen

Für jede elementare ON-Klasse wird eine spezielle Funktion, die sogenannte Klassen-Funktion generiert. Sie bildet über eine *switch-case*-Anweisung das funktionale Grundgerüst einer elementaren ON-Klasse. Abbildung 7.1 zeigt die Klassen-Funktion für die ON-Klasse *MotorVerhalten* (vgl. Abbildung 3.7). Abbildung 7.5 zeigt die Klassen-Funktion für die Klasse *KontaktSensorVerhalten*.

Abbildung 7.1
Die Klassen-Funktion für die elementare ON-Klasse *MotorVerhalten* (vgl. Abbildung 3.7)

```
void MotorVerhalten_Func (void* Data, unsigned* WaitTime) {
  struct MotorVerhalten_Struct* EONI = Data;
  switch (EONI->State) {
    case Stop :
       if (EONI->Port_Hoch) {
          MotorVerhalten_Action_Action2(EONI);
          EONI->Port_Hoch--;
          EONI->State = Dreht;
          if (EONI->Port_Runter) EONI->Port_Runter--;
          if (EONI->Port_Stop) EONI->Port_Stop--;
          return;
       }
       if (EONI->Port_Runter) {
          MotorVerhalten_Action_Action3(EONI);
          EONI->Port_Runter--;
          EONI->State = Dreht;
          if (EONI->Port_Stop) EONI->Port_Stop--;
          return;
       } else return;
    case Dreht :
       if (EONI->Port_Stop) {
          MotorVerhalten_Action_Action1(EONI);
          EONI->Port_Stop--;
          EONI->State = Stop;
          if (EONI->Port_Runter) EONI->Port_Runter--;
          if (EONI->Port_Hoch) EONI->Port_Hoch--;
          return;
       } else return;
  }
}
```

Erhält eine ON-Instanz den Steuerfokus, so wird die Klassen-Funktion ihrer ON-Klasse durch den Scheduler aufgerufen. Zur Unterscheidung unterschiedlicher Instanzen der gleichen Klasse wird der Funktion ein Zeiger (*Data*) übergeben, über den die Funktion auf die Daten der jeweiligen Instanz zugreifen kann. Diese instanzspezifischen Daten werden in einem

klassenspezifischen Strukturtyp, der sogennaten Klassen-Struktur, abgelegt. Abbildung 7.2 zeigt exemplarisch die Struktur für die Klasse *MotorVerhalten*. Neben dem Zustand (*State*) sind die drei Empfangssteuerports zu sehen.

Abbildung 7.2
Die Klassen-Struktur für die elementare ON-Klasse *MotorVerhalten* (vgl. Abbildung 3.7)

```
struct MotorVerhalten_Struct {
  enum { Stop=0, Dreht } State;
  unsigned char Port_Stop;
  unsigned char Port_Hoch;
  unsigned char Port_Runter;
} SchrankenMotor;
...
SchrankenMotor.State = Stop;  /* Def. init state */
SchrankenMotor.Port_Stop = 0; /* Empty receive ports */
...
```

Damit der Scheduler die Klassen-Funktionen aufrufen kann, müssen zu Beginn die jeweiligen Instanzen, nach den Festlegungen von Kapitel 5.2, in die beiden Tabellen (*standard1* bzw. *standard2*) eingetragen werden. Abbildung 7.3 zeigt den Aufbau der Struktur (*Scheduler_Struc*), mit der diese Tabellen als Arrays gebildet werden. In Gegensatz zu den beiden bereits vor Beginn der Abarbeitung initialisierten Arrays (*standard1* und *standard2*) bleiben *important1* und *important2* uninitialisiert. Der eigentliche Scheduler beschränkt sich schließlich auf eine Endlosschleife (in der *main*-Funktion), in der zyklisch die Tabelleneinträge nach den Vorgaben von Kapitel 5.2 abgearbeitet werden.

Abbildung 7.3
Die Scheduler-Datenstruktur

```
struct Scheduler_Struc {
   void* Data;
   void (*Func) (void* Data, unsigned* WaitTime);
   unsigned WaitTime;
} standard1[S1NMB], standard2[S2NMB], important1[I1NMB],
 important2[I2NMB];

standard2[0].Data = &SchrankenMotor;
standard2[0].Func = MotorVerhalten_Func;
standard2[0].WaitTime = 0;
```

Durch die Sequenz *standard2[0].Func(Standard2[0].Data, &standard2[0].WaitTime)* erhält z.B. die elementare ON-Instanz *SchrankenMotor* den Fokus durch den Scheduler.

7.1.2 Sendeport-Funktionen

Für jeden Sendeport erhält die Klassen-Struktur einen zusätzlichen Funktionszeiger, über den in der Klassen-Funktion eine instanzspezifische Sendeport-Funktionen aktiviert wird. Sendeport-Funktionen realisieren die spezifischen Nachrichtenverbindungen eines Ports. Für die Klasse *KontaktSensorVerhalten* wird für den Port *GehtZu* die folgende Deklaration in die Klassenstruktur aufgenommen: *void (*Port_GehtZu) (void);*.

Abbildung 7.4
Die Sendeport-Funktion für die Instanz *SensorUnten*

```
void SensorUnten_Port_GehtZu (void) {
 BahnSchranke_Schranke.Port_Unten++;
}
...
SensorUnten.Port_GehtZu = SensorUnten_Port_GehtZu;
```

Für die Instanz *SensorUnten* wird dieser Funktionszeiger mit der Funktion *SensorUnten_Port_GehtZu* initialisiert (siehe Abbildung 7.4). In der Klassen-Funktion (vgl. Abbildung 7.5) wird eine Sendeaktion schließlich über den Aufruf *EONI->Port_GehtZu()* angestoßen.

Abbildung 7.5
Die Klassen-Funktion für die elementare ON-Klasse *KontaktSensorVerhalten*

```
void KontaktSensorVerhalten_Func(void* Data, unsigned* WaitTime){
  struct KontaktSensorVerhalten_Struct* EONI = Data;
  switch (EONI->State) {
    case Offen :
      if (KontaktSensorVerhalten_Guard_Guard1(EONI)) {
        EONI->Port_GehtZu();
        EONI->State = Zu;
        *WaitTime = 10;
        return;
      } else return;
    case Zu :
      if (KontaktSensorVerhalten_Guard_Guard2(EONI) &&
          !*WaitTime) {
        EONI->State = Offen;
        return;
      } else return;
  }
};
```

Handelt es sich bei dem Sendeport um einen Datenport, so wird in den jeweiligen Sendeport-Funktionen der Inhalt des Sendepuffer, welcher zuvor durch eine Aktion manipuliert wurde, in die entsprechenden Empfangspuffer kopiert. Besitzt ein Empfangsport, der über eine Nachrichtenverbindung mit dem Sendeport verbundenen ist, *important*-Priorität, so sorgt die Sendeport-Funktion dafür, daß die Instanz des Empfangsport dynamisch in das Array *important2* eingetragen wird.

Im Gegensatz zur lokalen Kommunikation der Instanz *SensorUnten* wird bei der Instanz *SensorLinks*, die auf den zweiten Rechnerknoten (*Node2*, vgl. Tabelle 3.5) lokalisiert wurde, durch die Sendeport-Funktion eine Kommunikation über eine Kommunikationsschnittstelle (*CIO1*) initiiert. Über die plattformspezifische Konkretisierung der Methode *SendMessage* (vgl. Abbildung 3.38) wird hierbei eine spezielle Sendeinstanz adressiert (vgl. Abbildung 7.6).

Abbildung 7.6
Die Sendeport-Funktion für die Instanz *SensorUnten*

```
void SensorLinks_Port_GehtZu (void) {
 CIO1_SendMessage(BahnSchranke_Abschnitt_Port_Links,NULL);
};
```

Diese dynamisch verwaltete Instanz leitet, nachdem sie in das Array *important2* eingetragen wurde, die eigentliche Kommunikation mit *Node1* ein. Der Empfang an *Node1* wird durch die Empfangs-Interrupt-Routine dieses Knotens behandelt. Diese Routine kopiert über die plattformspezifische Implementierung der Methode *ReceiveMessage* (vgl. Abbildung 3.38) die empfangene Nachricht aus dem Puffer der Kommunikationsschnittstelle in den Empfangspuffer einer speziellen Empfangsinstanz. Danach wird diese Instanz in das Array *important1* eingetragen. Die Empfangsinstanz schließlich adressiert den Port (*Links*) der Anwenderinstanz (*BahnSchranke.Abschnitt*).

7.1.3 Wartezeiten

Der Zustandswechsel von *Zu* nach *Offen* in der Klasse *KontaktSensorVerhalten* enthält neben dem Guard (*Guard2*), der den Kontaktsensor ausliest, eine Wartezeit. In der Klassen-Funktion (vgl. Abbildung 7.5) wird die Wartezeitvariable der Scheduler-Struktur auf den spezifizierten Wert (10) gesetzt. Am Beginn des Scheduler-Umlaufs (vgl. Abbildung 5.8) verringert der Scheduler alle Wartezeitvariablen um den Wert einer globalen Variablen, die die Zeit für den letzten Scheduler-Umlauf repräsentiert (Unterläufe werden auf Null begrenzt). Nach dieser Operation erhält diese globale Variable den Wert Null zugewiesen. Durch die Timer-Interrupt-Routine wird die Variable während des Scheduler-Umlauf zyklisch pro Aufruf um eins erhöht. Die Zykluszeit des auslösenden Timers wird durch den Skalierungsfaktor (*timescale*) der beteiligten ON-Klassen bestimmt.

7.2 Umsetzung der Datenklassen

Attribute und Puffervariablen besitzen definierte Datenklassen. Bevor die Grundstrukturen einer elementaren ON-Klassen generiert werden können, müssen zuvor die verwendeten Datenklassen umgesetzt werden. Basisdatenklassen werden auf die grundlegenden Datentypen von ANSI-C umgesetzt. Komplexe Datenklassen werden in C-Datenstrukturen (*struct*) überführt. Die sogenannten Valuechecker, die während der Validierung eine Wertebereichsüberschreitung erkennen, werden nicht implementiert. Für eine komplexe Datenklasse (z.B. *myDC*), die einen Integer- und einen Double-Wert (*a* und *b*) enthält, würde folgende Struktur generieren werden: *struct myDC { int a; double b; };*.

7.2.1 Attribute und Puffervariablen

Besitzt eine elementare ON-Klasse Attribute, Sende- und Empfangsdatenpuffer, so finden sich diese in ihrer Klassen-Struktur wieder. Ein Attribut mit dem Namen *i* und der komplexen Datenklasse *myDC* wird z.B. über die Zeile: *struct myDC i;* in die Klassen-Struktur aufgenommen. Für einen Datensendeport mit dem Namen *Port1* und der gleichen Datenklasse würde die Zeile: *struct { struct myDC Port1; } send;* in die Klassen-Struktur einfügt werden. Anlog wird für die Antwortpuffer (*reply*) synchroner Ports verfahren.

7.2.2 Empfangspuffer-Arrays

Hinter der Empfangspuffervariablen (*receive.Port2* für den ARP *Port2*) steht im allgemein Fall nicht wie bei den Sendepuffern eine einzelne Variable, sondern ein Array, welches mehrere empfangene Nachrichten zwischenspeichern kann. Die Elemente dieses Arrays haben die Struktur der Datenklasse des zugehörigen Empfangsport. Die Länge des Array wird nach den in Kapitel 6.3.2 getroffenen Vereinbarungen festgelegt. Dabei bestimmt die Instanz mit dem größten Pufferbedarf die Länge des Arrays in der Klassen-Struktur. Neben dem Array werden drei *unsigned-char*-Variablen (*Port_Port2*, *Last_Port2* und *First_Port2* für den ARP *Port2*) generiert, die den aktuellen Füllstand des Puffer-Arrays widerspiegeln. *Port_Port1* speichert analog zu den Steuerports die Anzahl der gespeicherten Nachrichten. *Last_Port1* und *First_Port1* zeigern die älteste und die neueste Nachricht im Puffer-Array. Die Verwaltung des Arrays erfolgt über die Sendeport-Funktionen, die ihren Sendepuffer in das Array kopieren und darauf die drei Variablen entsprechend aktualisieren. In der Klassen-Funktion werden, sobald eine Nachricht aus dem Puffer entfernt wird, ebenfalls die drei Variablen modifiziert.

Abbildung 7.7
Aktion und ein Guard der Klasse *MyEONC* mit Datenzugriff

```
int MyEONC_Guard_Guard1(void* Data){
  struct MyEONC_Struct* EONI = Data;
  #define i EONI->i
  /* begin implementation code */
  i.a = i.a + 1;
  return (i.a == 5);
  /* end implementation code */
  #undef i
};
void MyEONC_Action_Action1(void* Data){
  struct MyEONC_Struct* EONI = Data;
  #define receive.Port2 EONI->receive[Last_Port2].Port2
  #define send EONI->send
  /* begin implementation code */
  send.Port1.a = 0;
  send.Port1.b = receive.Port2.b;
  /* end implementation code */
  #undef send
  #undef receive
};
```

7.2.3 Aktionen und Guards mit Datenzugriff

Für jede Aktion bzw. Guard wird eine C-Funktion generiert. Den unterschiedlichen Namensräumen (*name spaces*) von Aktionen und Gurards wird durch eine spezielle Namensgebung Rechnung getragen. Nach dem vorangestellten Klassennamen wird vor dem eigentlichen Namen zusätzlich *Action* bzw. *Guard* eingeschoben. Die C-Funktion für die Aktion *Action1* der Klasse *MyEONC* erhält z.B. den Namen *MyEONC_Action_Action1* (vgl. Abbildung 7.7). Damit diese Funktionen mit den Daten unterschiedlicher Instanzen der gleichen Klasse arbeiten können, wird ihnen ein Zeiger übergeben, über den sie auf die Daten der jeweiligen Instanz zugreifen.

Der Zugriff auf Attribute, Puffervariablen und Puffer-Arrays erfolgt im Implementierungscode der Aktionen und Guards in der für ANSI-C üblichen Art und Weise. Durch spezielle Definitionen (*#Define*), die vor dem Implementierungscode plaziert werden, wird die Eingliederung der Daten in die Klassenstruktur und der Aufbau der Puffer-Arrays für den Implementierungscode verborgen.

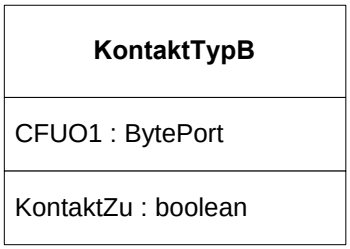
7.3 Umsetzung der Plattformabstraktion

Durch das Mapping auf eine konkrete Plattform (*Node1* in Tabelle 3.5) wurden den Objekten abstrakter AS-Klassen konkrete implementierbare AS-Klassen zugeordnet (*MotorTypA* für *SchrankenMotor* und *KontaktTypB* für *SensorUnten*, siehe Abbildung 3.39). Dies liefert den Ausgangspunkt für die Umsetzung der Plattformabstraktion.

7.3.1 ASO-Codesynthese

Die in der Steuerung der Bahnschranke eingesetzte abstrakte AS-Klasse *KontaktSensor* besitzt zwei konkrete Unterklassen (vgl. Abbildung 3.34). Auf dem ersten Rechnerknoten konkretisiert die Unterklasse *KontaktTypB* (vgl. Abbildung 7.8) das AS-Objekt *SensorUnten*. Die konkretisierte Methode *KontaktZu* dieser Klasse verwendet für den Hardware-Zugriff das CFU-Objekt *CFUO1*. Durch den Anwender wurde der Methode *KontaktZu* die Anweisungsfolge: *return (CFUO1.getByte() & 0x01);* zugewiesen; sie testet, ob Bit 0 des Byte-Ports gesetzt ist.

Abbildung 7.8
Definition der konkreten AS-Klasse *KontaktTypB*



Da *KontaktTypB* rechnerknotenunabhängig entworfen wurde (wie alle konkreten AS-Klassen), benutzt sie abstrakte CFU- bzw. PIC-Klassen für die Hardware-Ansteuerung. Erst bei der Definition einer Plattform (vgl. *NodeClass1* in Abbildung 3.39), bei der ein Objekt (*ASO2*) der Klasse *KontaktTypB* entsteht, erhält dieses Objekt ein konkretes rechnerknotenabhängiges CFU-Objekt (*C167_Port1*) zugeordnet. In Abbildung 3.39 wurde *C167_Port1* nicht dargestellt.

Abbildung 7.9
Umsetzung des AS-Objektes *ASO2*

```
unsigned char CFUO_C167_Port1_getByte(void) {
  /* begin 80C167 specific ASM implementation code */
  ...
  /* end implementation code */
};

int ASO2_KontaktZu(void){
  #define CFUO1.getByte CFUO_C167_Port1_getByte
  /* begin implementation code */
  return (CFUO1.getByte() & 0x01);
  /* end implementation code */
  #undef CFUO1.getByte
};
```

Die Umsetzung der konkreten AS-Klassen (wie z.B. *KontaktTypB*) erfolgt auf der Ebene ihrer plattformspezifisch zugeordneten Objekte. Abbildung 7.9 zeigt die Umsetzung des Objekts *ASO2*, welches sich auf dem Rechnerknoten *Node1* befindet. Das abstrakte CFU-Objekt wurde dabei bereits durch die plattformspezifische Klasse *C167_Port1* ersetzt (konkretisiert).

Abbildung 7.10
Umsetzung von *Guard1* der Klasse *KontaktSensorVerhalten* mit AS-Zugriff

```
int KontaktSensorVerhalten_Guard_Guard1(void* Data){
  struct KontaktSensorVerhalten_Struct* EONI = Data;
  #define KontaktZu EONI->KontaktZu
  /* begin implementation code */
  return (KontaktZu() == 1);
  /* end implementation code */
  #undef KontaktZu
};
```

Für jede Methode des referenzierten AS-Objekts erhält die Klassen-Struktur einen zusätzlichen Funktionszeiger. Für die Klasse *KontaktSensor* wird die folgende Deklaration aufgenommen: *int (*KontaktZu) ();*. Für die ON-Instanz *SensorUnten* wird dieser Zeiger wie folgt initialisiert: *SensorUnten.KontaktZu = ASO2_KontaktZu;*. Abbildung 7.10 zeigt die Umsetzung von *Guard1*, in dem die Methode *KontaktZu* verwendet wird.

7.3.2 Bereitstellen der Kommunikationssoftware

Die Plattformspezikation für den ersten Rechnerknoten (vgl. Abbildung 3.39) enthält ein Kommunikationsinterface (CIO1) der konkreten, knotenspezifischen CI-Klasse *CAN_C167*. Die Erstellung der Kommunikationssoftware für diesen Rechnerknoten konzentriert sich darauf, die knotenspezifischen Implementierungen der durch die Meta-CI-Klasse (vgl. Abbildung 3.38) vorgegebenen Methoden dem Code-Generator bereitzustellen. Die Implementierung der Methode

SendMessage der Klasse *CAN_C167* würde für den betrachteten Rechnerknoten durch die C-Funktion *CIO1_SendMessage* bereitgestellt.

8. Kapitel

Zusammenfassung und Ausblick

Gegenwärtig sind durchgängige und praktikable Entwurfsmethoden für verteilte eingebettete Echtzeitsysteme nur in Ansätzen im Einsatz. Bislang erfolgte der Entwurf, die Validierung und die Implementierung eines eingebetteten Systems häufig völlig getrennt voneinander. Diese mangelnde Durchgängigkeit reduziert dabei die Produktivität des Entwicklers erheblich. Die wachsende Komplexität und die steigenden Sicherheitsanforderungen von Echtzeitsystemen werden allerdings in naher Zukunft den Einsatz von integrierenden Entwurfsmethoden zwingend erforderlich machen. Diese Methoden müssen zum einen die Komplexität durch moderne graphische und objektorientierte Notationsformen meistern und zum anderen die erstellten Entwürfe einer rechnergestützten Validierung zugänglich machen. Damit die durch eine vollständige Simulation gewonnen Aussagen über die Einhaltung sicherheitsrelevanter Anforderungen auch nach der Implementierung ihre Gültigkeit behalten, muß der Validierung eine automatische Code-Generierung nachgeschaltet sein. Nur dieser direkte Übergang auf die verteilte Zielplattform, deren spezifisches Zeitverhalten in der Validierung bereits berücksichtigt wurde, garantiert die Einhaltung von verifizierten Eigenschaften auch für die implementierten Entwürfe.

Ziel dieser Arbeit war es, eine Entwurfsmethodik – basierend auf den Objektnetzen – zu entwickeln und vorzustellen. Wobei der geforderte durchgängige Entwurf für verteilte eingebettete Echtzeitsysteme ermöglicht wird. Die Objektnetz-Methodik verbindet hierzu die graphische Notationsform einer datenflußorientierten Moduldarstellung mit der zustands-orientierten Darstellung hierarchischer erweiterter Zustandsmaschinen. Es werden hierarchische Strukturierungsmöglichkeiten sowohl für die Modulebene als auch für die Zustandsebene angeboten. Für die schrittweise Konkretisierung wird auf das objektorientierte Konzept der Vererbung zugegriffen. Zur formalisierten Notation wichtiger, sicherheitsrelevanter Anforderungen wurde die Objektnetz-Constraint-Sprache entwickelt. Für Constraints, die in dieser Sprache formuliert wurden, besteht der Zugang zur automatischen Verifikation.

Damit der Implementierungsvorgang unbeeinflußt von der jeweiligen Zielplattform erfolgen kann, wurde ein objektorientiertes Framework zur Abstraktion verteilter Rechnerplattformen eingeführt. Über die Klassen dieses Framework werden die genutzten Hardware-Ressourcen gekapselt; die unterschiedlichen Ansteuerungsdetails plattformabhängiger Peripheriekomponenten bleiben verborgen.

Die Validierung einer Objektnetz-Spezifikation, die in die zu implementierende Steuerung und in die Umgebung zerfällt, erfolgt auf Basis einer Verhaltensbeschreibung mit höheren Petri-Netzen. Das Verhalten der Objektnetz-Spezifikationen läßt sich hierzu vollständig mit einer speziellen höheren Petri-Netz-Klasse beschreiben. Durch eine Simulation, die mit einer graphischen Animation der gesteuerten Umgebung gekoppelt wird, erfolgt die Validierung der prinzipiellen Funktion. Durch die sogenannte vollständige Simulation, bei der alle Varianten des Umgebungsmodells durchgespielt werden, lassen sich die ON-Constraints verifizieren. Vor der vollständigen Simulation erfolgt die Verifikation genereller Anforderungen, wie z.B. die Vollständigkeit oder der korrekte zeitliche Bezugs der Steuerung zur Umgebung. Wurden die Zeitparameter für die Rechnerknoten und für das Kommunikationsnetzwerk ermittelt, so berücksichtigt die Validierung bereits das reale Zeitverhalten der späteren Implementierung.

Der durchgängige Entwurfsvorgang der Objektnetz-Methodik wird durch die automatische Generierung der Software für die Steuerung abgeschlossen. Die Code-Generierung erfolgt auf der Basis von ANSI-C, wodurch eine größtmögliche Universalität gewährleistet wird. Der Code-Generator kombiniert Software-Komponenten aus der Plattformspezifikation, die die spezifische Hardware-Struktur kapseln, mit der Umsetzung der Objektnetz-Steuerungs-spezifikation. Für die Koordinierung mehrerer Instanzen setzt die Objektnetz-Methodik ein kooperatives Abarbeitungsmodell ein.

Mit dem *Object System Specification Inventory* (OSSI) wurde begleitend zu dieser Arbeit eine Tool-Unterstützung realisiert. OSSI nutzt moderne Software-Techniken für die graphische Nutzerschnittstelle und die Organisation der Entwurfsdaten. Objektnetz-Spezifikationen werden über eine zentrale Datenbank verwaltet, wodurch die Konsistenz einer Spezifikation jederzeit gewährleistet bleibt. Im Anhang erfolgt die detaillierte Beschreibung von OSSI.

Durch die Auseinandersetzung des Autors mit dem Entwurf verteilter eingebetteter Systeme zeigten sich verschiedene Punkte für die Weiterführung der Forschungstätigkeiten:

- Die erste Forschungsrichtung betrifft die Einbeziehung des Integrationstest auf der Zielplattform; die Validierung sollte bereits vorhandene Sensorik und Aktorik einbeziehen.
- Eine weitere Forschungsaktivität muß der Optimierung der Validierung gelten. Es müssen Verfahren gefunden werden, die eine Verifikation von Teilmodellen ermöglichen.
- Die Forschungslinie, der der Autor die größte Bedeutung zumißt, betrifft die Verfahren zur Plattformabstraktion, sowie des Hardware-Software-Co-Designs. Es bleibt zu untersuchen inwieweit applikationsspezifische „Objektnetz-Schaltkreise” eine sinnvolle Alternative zu einer traditionellen Mikrocontroller-Realisierung darstellen.

A-Anhang

Toolunterstützung

Die Akzeptanz einer modernen Entwurfsmethode ist durch eine Reihe von teilweise sehr unterschiedlichen Faktoren geprägt. Neben kommerziellen und markt-strategischen Einflüssen ist zweifellos die Art und Weise der Werkzeugunterstützung ein wichtiger Faktor. Mögliche Anwender kommen in erster Linie über das bereitgestellte Entwurfswerkzeug mit einer neuen Methode in Kontakt. Das heißt für den Entwerfer der Objektnetz-Methodik, daß er neben einer fundierten und theoretisch abgesicherten Methode zusätzlich ein tragfähiges Konzept für den computerunterstützten Entwurf bereitstellen muß. Der mit den Objektnetz-Methodik verbundene Entwurfsvorgang muß in allen Phasen optimal begleitet werden, so daß der Anwender jederzeit das Gefühl hat, daß der Rechner seine Kreativität unterstützt und nicht behindert.

Der folgende Abschnitt gibt einen kurzen Architekturüberblick. Er zeigt schematisch den internen Aufbau der Objektnetz-Entwurfsunterstützung OSSI (*Object System Specification Inventory*). Anschließend erfolgt eine zusammenfassende Darstellung der Hauptfunktionen von OSSI. Neben dem zentralen Objektnetz-Entwurf wird die Plattformabstraktion, die Projektdefinition, die Validierung sowie die Codegenerierung behandelt.

A.1 Architekturüberblick

Parallel zur Entwicklung der Objektnetz-Methodik erfolgte die Konzeption und Umsetzung einer Entwurfsumgebung. OSSI (*Object System Specification Inventory*) verfolgt nicht alleinig das Ziel den Anwender durch den gesamten Entwurfsablauf zu begleiten, zusätzlich soll es eine tragfähige Architektur für zukünftige Erweiterungen bereitstellen. Modifikationen bzw. Ergänzungen der Funktionalität sollen einfach realisierbar sein. Um dies zu erreichen wurde eine größtmögliche Entkopplung von Speicherung und Verarbeitung der Projektdaten angestrebt.

Abbildung A.1
Architektur der Objektnetz-Entwurfsumgebung OSSI

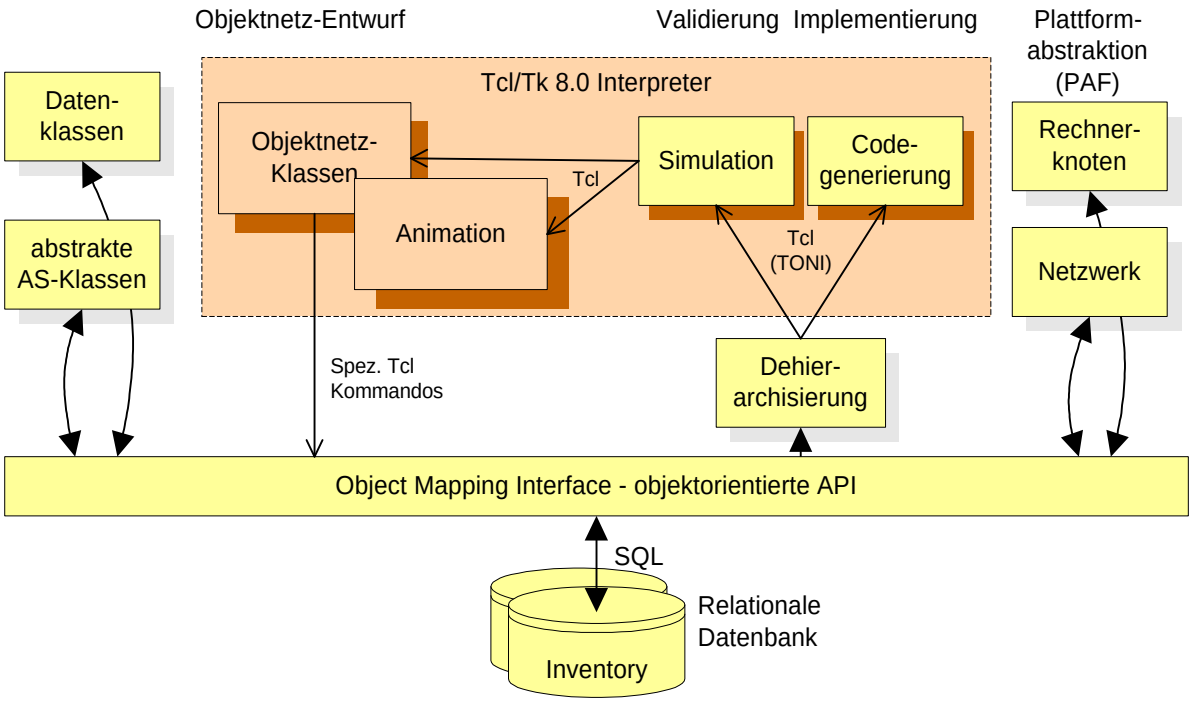


Abbildung A.1 zeigt die Grundarchitektur der Entwurfsumgebung. Das Grundgerüst von OSSI wurde mit dem RAD-Werkzeug (*Rapid Applikation Development*) Delphi entwickelt. Delphi unterstützt die objektorientierte Entwicklung interaktiver Applikationen für Windows 9x und Windows NT. Neben den sehr kurzen Übersetzungszeiten des Objekt-Pascal-Compilers und dem einfachen Umgang mit der Delphi-Oberfläche gab das breite Angebot an vorgefertigten Software-Komponenten den Ausschlag. Im besonderen begünstigten die Komponenten für die Verwaltung relationaler Datenbanken diese Entscheidung.

A.1.1 Verwaltung der Entwurfsdaten

OSSI verwaltet sämtliche Entwurfsdaten in einer relationalen Datenbank. Diese spezialisierte Datenbank wird *Inventory* genannt. Es werden zwei Standarddatenbanken unterstützt. Zum einen können die Entwurfsdaten über Paradox-Tabellen verwaltet werden. Zum anderen besteht die Möglichkeit, einen Interbase-SQL-Server mit dieser Aufgabe zu betrauen. In beiden Fällen

werden die Tabellen über die standardisierte Abfragesprache SQL (*Structured Query Language*) angesprochen. Spezielle Delphi-Komponenten sorgen dafür, daß die SQL-Anweisungen entweder an den SQL-Server durchgereicht, oder in direkte Datenbankoperationen für die Paradox-Tabellen umgewandelt werden. Der Einsatz eines getrennt von OSSI laufenden SQL-Servers ermöglicht sowohl einen größeren Datendurchsatz als auch eine höhere Datensicherheit. Läuft der Server auf einem getrennten Rechner, so ist es möglich mehrere Projektteilnehmer auf diese Datenbestände zugreifen zu lassen.

Durch den objektorientierten Aufbau und die damit verbundene objektorientierte Sicht von OSSI auf die Projektdaten ergab sich die Notwendigkeit, die Tabellenstruktur der Daten zu kapseln. Eine spezielle Programmierschnittstelle API (*Application Programing Interface*) – das sogenannte *Object Mapping Interface* [Kahnert 98] – realisiert diese objektorientierte Sicht (vgl. Abbildung A.1).

A.1.2 Kopplung mit dem Tcl/Tk-Interpreter

Neben Delphi bildete Tcl/Tk [Ousterhout 95] das zweite Standbein bei der Entwicklung von OSSI . Der Tcl-Interpreter wurde zusammen mit dem Graphik-Toolkit Tk in das mit Delphi erstellte OSSI-Grundgerüst eingebettet. Tk bietet durch sein Canvas-Widget (Zeichenfläche) eine einfache Möglichkeit, interaktive Applikationen zu erstellen. Das Tk-Canvas stellt bereits eine eigene Verwaltung von graphischen Objekten (*display list*) zur Verfügung. Neben der mächtigen Graphikunterstützung bietet die Einbindung eines Interpreters, dessen Befehlsumfang durch Delphi-Prozeduren dynamisch erweitert werden kann, weitere nutzbringende Eigenschaften.

Der eingebundene Tcl-Interpreter definiert für OSSI eine Makrosprache. Sie ermöglicht eine flexible Kopplung mit weiteren Tool-Komponenten. Der Code-Generator bzw. der Simulator erhalten ihren Input als spezielles Tcl-Skript. Dieses Skript in der Syntax von Tcl enthält alle Informationen, die für die Simulation bzw. Implementierung benötigt werden. Durch diese textuelle Schnittstelle – TONI (*Textual Object net Notation Interface*) – können externe Codegeneratoren und Simulatoren flexibel angebunden werden. Die vollständige Beschreibung aller TONI-Kommandos findet sich in [TONI 98].

Über Aktionen bzw. Guards einer Objektnetz-Umgebungsspezifikation, die Tcl-Skripts enthalten, können flexible Animationsszenarien aufgebaut werden. Während der Simulation werden diese über weitere Tcl-Kommandos fortlaufend dem Simulationsfortschritt angepaßt.

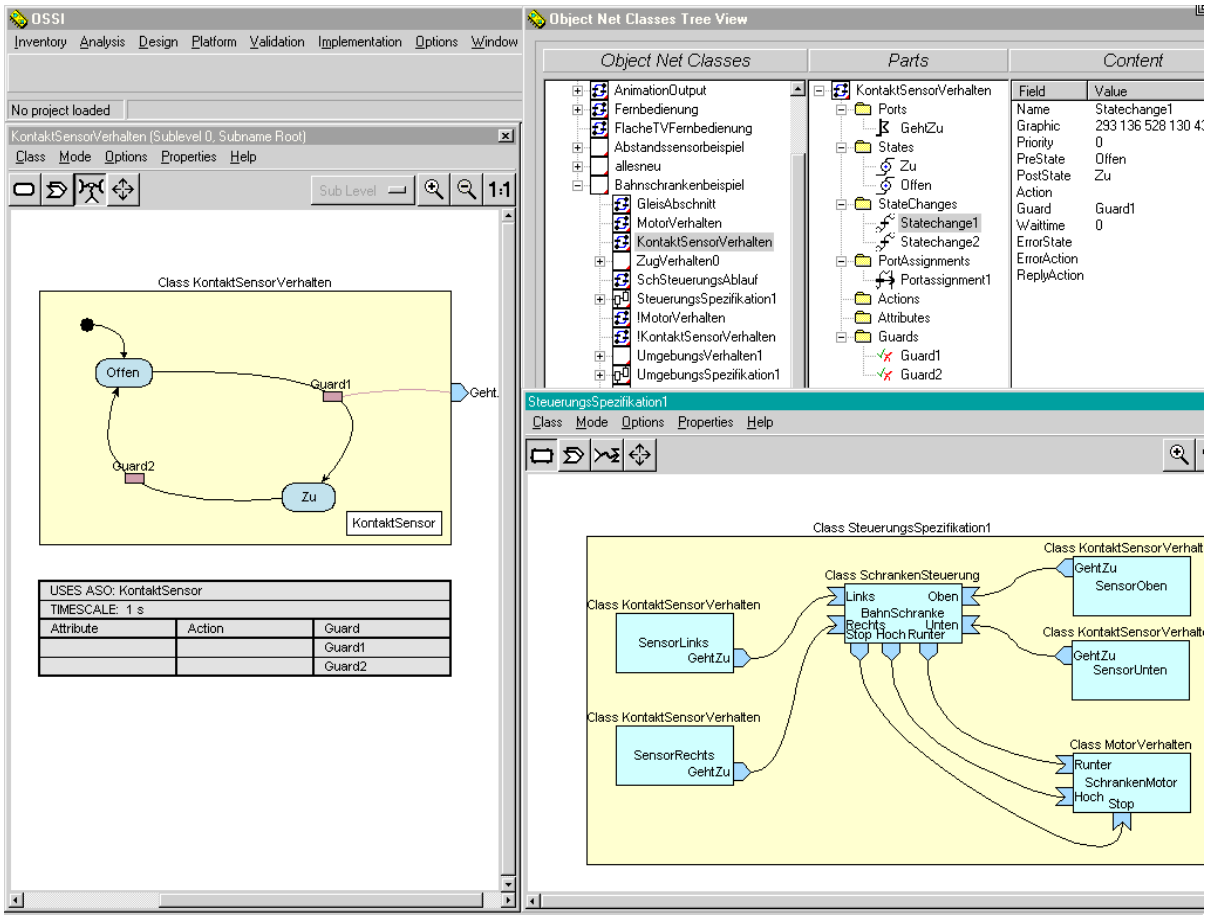
A.2 Funktionsüberblick

In Abbildung A.1 sind die vier Säulen der Objektnetz-Entwurfsunterstützung zusehen. Der eigentliche Objektnetz-Entwurf mit den Datenklassen, den abstrakten AS-Klassen sowie den Objektnetz-Klassen bildet die Hauptsäule. Die zweite und dritte Säule trägt die Validierung und Implementierung. Die vierte Säule – die Plattformabstraktion – liefert die Voraussetzung für eine Implementierung auf einer verteilten Plattform. Verbunden werden die vier Säulen durch die Projektdefinition.

A.2.1 Objektnetz-Entwurf

Der Entwurf abstrakter, elementarer bzw. hierarchischer ON-Klassen erfolgt vollständig graphisch. Unter dem Menüpunkt „*Design*" findet sich der Eintrag „*Object Nets*". Über ihn wird eine Baumdarstellung aller Objektnetz-Klassen (vgl. Abbildung A.2, rechts oben) aktiviert. Es können bestehende Klassen gelöscht bzw. neue Unterklassen angelegt werden.

Abbildung A.2
Bearbeitung elementarer und hierarchischer ON-Klassen mit OSSI



Neben Darstellung der Vererbungsbeziehungen als Baum (*tree view*) können im zweiten und dritten Fensterabschnitt Detailinformationen über einzelne Klassen abgerufen werden. In Abbildung A.2 werden die Informationen über den ersten Zustandswechsel (*Statechange1*) der elementaren ON-Klasse *KontaktSensorVerhalten* angezeigt. Aus dem Klassenbaum heraus lassen sich mehrere Objektnetz-Editoren starteten. Die Tk-Editorfenster, die eine vollgraphische Bearbeitung der ON-Klassen ermöglichen, besitzen ihre eigenen Menüs und Steuerelemente (vgl. Abbildung A.2). Sie dienen z.B. der Skalierung und dem Umschalten unterschiedlicher Bearbeitungsmodi (z.B. Einfügen und Verschieben).

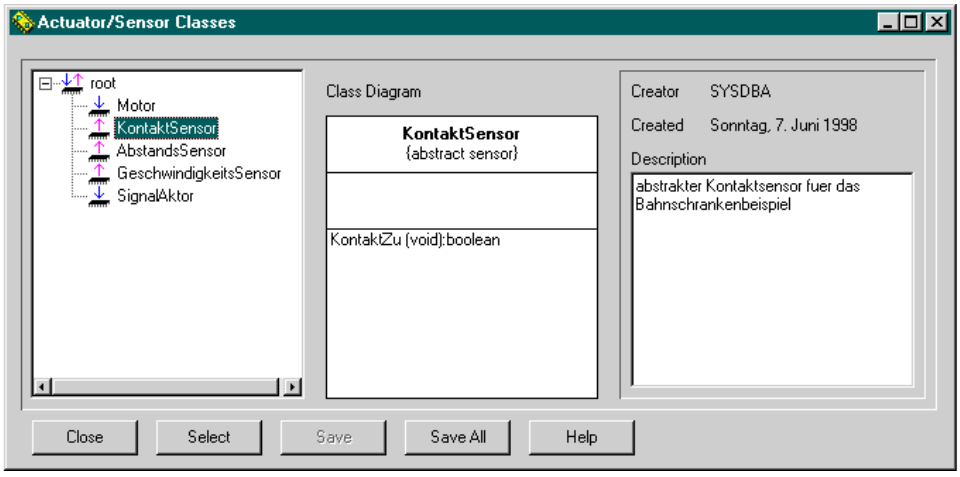
- **Datenklassen:**

Einige ON-Klassen verwenden anwendungsspezifische Datenklassen. Unter dem Menüpunkt „*Design*" kann über den Eintrag „*Data Classes*" ein Dialog geöffnet werden, mit dem sich alle Datenklassen des Inventories bearbeiten lassen. Die Veränderung und Neudefinition erfolgt über vier Eingabefelder. Neben dem Namen wird Beschreibung der Datenklasse eingegeben. Optional erfolgt die Angabe des sogenannten Value-Checkers (vgl. Kapitel 3.3). Im vierten Eingabefeld wird die Default-Belegung definiert.

- **Aktor/Sensor-Klassen:**

Über die abstrakten Aktor/Sensor-Klassen erfolgt die Kopplung von Steuerung und Umgebung. Die elementare Klasse *KontaktSensorVerhalten* in Abbildung A.2 referenziert hierzu ein Objekt der abstrakten AS-Klasse *KontaktSensor*. Der Eintrag „*Actuator/Sensor Classes*" unter dem Menüpunkt „*Design*" öffnet ein Eingabeformular (vgl. Abbildung A.3) mit dem AS-Klassenbaum; er enthält alle Aktor/Sensor-Klassen des Inventories. Anlog zu den ON-Klassen beginnt die Vererbungshierarchie mit einer abstrakten Wurzelklasse (*root*). Nur die direkten Unterklassen der Wurzelklasse können innerhalb einer elementaren ON-Klasse eingesetzt werden; sie sind automatisch abstrakt; ihre Methoden besitzen keine Implementierung. In der zweiten (und letzten) Vererbungsebene finden sich bereits konkrete Klassen; sie gehören einer speziellen Plattformabstraktion an.

Abbildung A.3
Eingabeformular für Aktor/Sensor-Klassen mit Baumdarstellung



Neben dem Klassenbaum zeigt das Eingabeformular das UML-Klassen-Diagramm der jeweils selektierten Klasse. In oberen Teil dieses Diagramms wird der Name und die spezielle Eigenschaft der Klasse (z.B. *{abstract sensor}* in Abbildung A.3) angezeigt. Der Attributteil mit den PIC- und CFU-Referenzen ist bei abstrakten Klassen grundsätzlich leer. Im unteren Drittel werden die Methoden aufgelistet. Die rechte Formularseite bietet die Möglichkeit, für jede Klasse eine informelle Beschreibung zu hinterlegen.

A.2.2 Plattformabstraktion

Voraussetzung für die Plattformabstraktion ist die Existenz konkreter AS-Klassen. Im AS-Eingabeformular (vgl. Abbildung A.3) werden diese durch Vererbung aus den abstrakten AS-Klassen abgeleitet. Die dabei entstehenden konkreten Klassen referenzieren zusätzlich abstrakte CFU- bzw. PIC-Objekte. Die Definition dieser CFU- und PIC-Klassen erfolgt analog zu den AS-Klassen über zwei separate Eingabeformulare (unter dem Menüpunkt „*Platform*"). Die Definition der Kommunikationsschnittstellen-Klassen (CI-Klassen) geschieht in gleicher Weise. Allerdings werden konkrete CI-Klassen in der Regel durch die Einbindung bereits vorhandener Kommunikationsbibliotheken realisiert; die konkrete CI-Klassen kapselt diese Bibliothek.

- **Rechnerknotenklassen:**

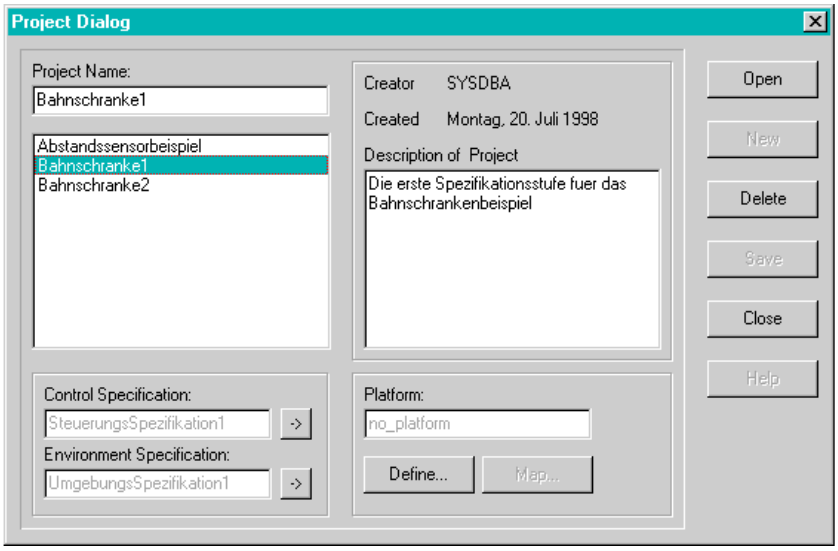
Die Funktionen eines konkreten Rechnerknotentyps wird durch eine Rechnerknotenklasse (*NodeClass*) abstrahiert (vgl. Kapitel 3.4). Unter dem Menüpunkt „*Platform*" öffnet der Eintrag „*Controller Nodes*" das Eingabeformular für die Bearbeitung dieser Klassen. Im Formular sind alle Klassen des Inventories über eine Baumdarstellung erreichbar. Der Baum besitzt, analog den AS-Klassen, zwei Verebungsebenen. In der ersten Ebene werden abstrakte Knotenklassen definiert; sie enthalten nur Referenzen auf abstrakte AS- und CI-Objekte. In der zweiten Ebene befinden sich die abgeleiteten konkreten Knotenklassen. Nachdem eine konkrete Klassendefinition abgeleitet wurde, können die konkreten CFU-Objekte, die den direkten Zugriff auf die Hardware kapseln, eingefügt werden. Auf diese CFU-Objekte aufbauend werden – optional – konkrete PIC-Objekte hinzugefügt. Hierbei werden Referenzen auf abstrakte CFU-Objekte durch Referenzen auf konkrete – bereits in der Knotenklasse vorhandene – CFU-Objekte aufgelöst. In anloger Weise werden konkrete CI- und AS-Objekte in die konkrete Knotenklasse aufgenommen. Ihre Referenzen auf abstrakte CFU- und PIC-Objekte werden ebenfalls durch vorhandene konkrete Objekte aufgelöst. Bestimmte CI-Klassen referenzieren weder CFU- noch PIC-Objekte; für sie entfällt die Auflösung der Referenzen.

Abgeschlossen wird die Definition einer Rechnerknotenklassen durch die Angabe organisatorischer Informationen, die für die automatische Implementierung benötigt werden. Dazu zählen die Auswahl des Compilers, die Angabe eines Makefiles sowie die Bereitstellung Rechnerknoten-spezifischer Programmdateien (z.B. Header-Dateien zur Initialisierung).

A.2.3 Projektdefinition

Über die Definition eines Objektnetz-Projektes wird schließlich eine Objektnetz-Spezifiktion mit einer Plattform verbunden. Unter dem Menüpunkt „*Inventory*" findet sich der Eintrag „*Projects...*"; er öffnet einen Dialog über den sowohl vorhandene Projekte zur Validierung und Codegenerierung ausgewählt als auch neue angelegt werden können. Im Rahmen der Projektdefinition erfolgt die Angabe zweier hierarchischer ON-Klassen. Die erste Klasse definiert die Steuerung (*Control*), die zweite die Umgebung (*Environment*). In Abbildung A.4 wurde für das Projekt *Bahnschranke1* die Klassen *SteuerungsSpezifiktion1* und *UmgebungsSpezifikation1* ausgewählt.

Abbildung A.4
Der Projektdialog verknüpft eine ON-Spezifikation mit einer Plattformabstraktion



Vom Projektdialog aus gelangt man über den *Button* mit der Aufschrift „*Define...*" zum Formular zur Definition einer Plattform (in Abbildung A.4 wurde durch den Anwender noch keine Plattform definiert).

- **Plattformdefinition:**

An einer Plattformdefinition sind ein oder mehrere Instanzen (*Nodes*) der im Inventory abgelegten abstrakten oder konkreten Knotenklassen beteiligt. Zwischen diesen Instanzen werden die möglichen Kommunikationsverbindungen (*Lines*) festgelegt. An die *Nodes* und *Lines* werden schließlich die in Kapitel 3.4.2 definierten Zeitparameter notiert. Eine auf diese Weise erstellte Plattformdefinition kann in mehreren Projekten eingesetzt werden.

- **Mapping:**

Nachdem eine neue Plattform definiert bzw. eine vorhanden ausgewählt wurde erfolgt die Zuordnung (*Mapping*) der ON-Steuerungsspezifikation. Dies geschieht (vgl. Kapitel 3.5) indem durch den Anwender alle elementaren ON-Instanzen einer Rechnerknoteninstanz zugeordnet werden. Für Instanzen ohne Aktor/Sensor-Referenz existieren hierfür keine Einschränkungen. Instanzen mit Sensor- bzw. Aktorbezug können dagegen nur auf den Knoten, die die entsprechenden AS-Objekte bereitstellen, angeordnet werden. Stellt die Rechnerknotenklasse mehrere AS-Objekte der gleichen Klasse bereit, so muß manuell das gewünschte AS-Objekt ausgewählt werden.

A.2.4 Validierung

Für die Validierung eines Objektnetz-Projektes muß dieses zuvor über den Projektdialog (vgl. Abbildung A.4) ausgewählt werden. Die Validierung (unter dem Menüpunkt „*Validation*") unterscheidet zwischen der Simulation mit Animation und der vollständigen Simulation (vgl. Kapitel 6.4 und 6.5). Die Simulation mit Animation besitzt zwei unterschiedliche Ausführungs-varianten. Die erste Variante läuft Petri-Netz-basiert und nutzt das erweiterte Tcl als Modellierungssprache. In der zweiten Varianten wird die Objektnetz-Spezifikation automatisch compiliert. Der compilierte Code deckt sich weitestgehend mit dem, der für die eigentliche Zielplattform (vgl. Kapitel 7) generiert wird. Für die Übersetzung auf der Entwicklungs-plattform (Windows 9x und NT) wird Visual C++ von Microsoft eingesetzt.

A.2.5 Code-Generierung

Ein separater Menüpunkt („*Implementation*") ermöglicht, nachdem ein entsprechende Projekt geöffnet wurde, die automatische Code-Generierung. Über ein weiteres Dialogfenster wird die Rechnerknoteninstanz ausgewählt, für die die Erstellung der Software erfolgen soll.

B-Anhang

Demonstrationsbeispiele

Durch die Beschreibung zweier Demonstrationsbeispiele, soll dem Leser die vorgestellte Objektnetz-Methodik weiter vertraut werden. Begleitend wird ein Vorschlag für ein zweckmäßiges Vorgehensmodell unterbreitet. Wie bereits zu Beginn von Kapitel 3 ausgeführt, besteht nicht die Absicht, im Rahmen der vorliegenden Arbeit ein vollständiges Vorgehensmodell für die Objektnetz-Methodik einzuführen. Dennoch kann die Arbeit nicht schließen, bevor nicht anhand eines überschaubaren Beispiels einige ergänzende Regeln für ein pragmatisches Vorgehen formuliert werden.

Das folgende Kapitel beginnt mit dem bereit auf Seite 14 eingeführten *Generalized Railroad Crossing*. Der Schwerpunkt des Bahnschrankenbeispiels liegt in der Veranschau-lichung des Vorgehens. Es wurden bewußt komplexere Modellierungsmöglichkeiten zugunsten einer klaren Beschreibung des Entwurfsvorgangs weggelassen.

Das zweite Demonstrationsbeispiel, ein Abstandswarnsystem für Straßenfahrzeuge, soll die Fähigkeit der Objektnetz-Methodik demonstrieren, auch komplexere Aufgaben mit kontinuier-lichen Anteilen zu bewältigen. Im zweiten Beispiels wurde auf eine breite Darstellung des methodischen Vorgehens zugunsten relevanter Modelldetails verzichtet.

B.1 Das Bahnschrankenbeispiel

Das Bahnschrankenbeispiel begleitete den Leser bereits durch den Hauptteil der Arbeit. Die Aufgabenstellung für das auch als *Generalized Railroad Crossing* bezeichnete Standardbeispiel findet sich auf Seite 14. Um die Durchgängigkeit der Entwurfsmethodik zu unterstreichen, werden als zusätzliche Randbedingungen die anzusteuernden Sensoren und Aktoren in die Aufgabenstellung integriert. Die Ein- und Ausfahrt eines Zuges wird durch einen Sensor am Anfang und am Ende des Abschnitts festgestellt. Es existiert ein Motor, der die Schranke bewegt und zwei Sensoren, die die beiden Endlagen der Schranke signalisieren.

An verschiedenen Stellen der Arbeit diente die Bahnschranke als Anschauungsbeispiel; wobei bereits verschiedene Teilaspekte des Entwurfs dargestellt wurden. Im folgenden werden die fehlenden Teile ergänzt. Gleichzeit werden Hinweise für ein schrittweises Vorgehen innerhalb der Objektnetz-Methodik präsentiert.

Abbildung 3.3 zeigt bereits die drei Hauptstufen der Objektnetz-Methodik. Die drei Stufen untergliederen sich jeweils in mehrere Schritte. Die Identifikation der Aktoren und Sensoren bildet den Einstieg in den Entwurfsprozeß.

● Schritt 1 - Identifikation der physikalischen Aktoren und Sensoren:

Die Identifikation der Sensoren und Aktoren erfolgte bereits in der Aufgabenstellung. Die folgenden eindeutigen Bezeichner werden ihnen zugewiesen: der Motor in der Schranke wird mit *SchrankenMotor* bezeichnet; die beiden Sensoren, die die obere bzw. die untere Endlage der Schranke signalisieren, werden mit *SensorOben* und *SensorUnten* bezeichnet; die Sensoren, die die Ein- und Ausfahrt von links bzw. rechts signalisieren, werden mit *SensorLinks* und *SensorRechts* bezeichnet.

● Schritt 2 - Klassifikation der Aktoren und Sensoren:

Die identifizierten und bezeichneten Aktoren und Sensoren werden im zweiten Schritt aufgrund ihrer Gemeinsamkeiten in der Ansteuerung zu abstrakten Klassen zusammengefaßt. Die dabei gefundenen abstrakten AS-Klassen liefern zum einem den Einstieg für die spätere Konkretisierung durch die Plattformabstraktion (PAF), zum andern erfolgt über Objekte dieser Klassen die Kopplung von Steuerungs- und Umgebungsspezifikation.

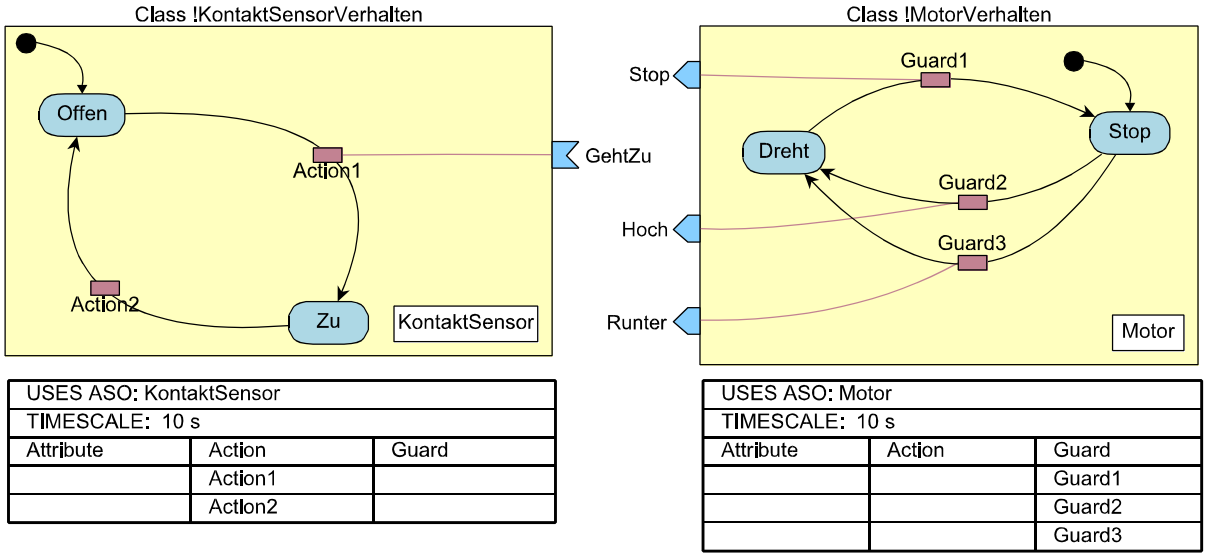
Die Befehle Motorrechts- und Motorlinkslauf können als Befehle zum Absenken bzw. Öffnen der Schranke gedeutet werden. Welche Drehrichtung die Schranke öffnet bzw. schließt, muß als Randbedingung bekannt sein. Hinter dem Kontaktsensor könnte sich bei der Realisierung z.B. eine Lichtschranke verbergen, die eine Unterbrechung des Lichtstrahls als geschlossenen Kontakt interpretiert. Damit währe die Ein- und Ausfahrt eines Zuges feststellbar. Auf der Ebene der Modellierung und Validierung sind jedoch solche Realisierungsdetails noch ohne Bedeutung. Es lassen sich folglich alle im ersten Schritt identifizierten Sensoren unter der abstrakten Klasse *KontaktSensor* zusammenfassen.

- **Schritt 3 - Spezifikation der Schnittstelle der AS-Klassen:**

Im Rahmen der Beschreibung elementarer ON-Klassen wurde bereits in Kapitel 3 der prinzipielle Aufbau von AS-Methoden skizziert. In Abbildung 3.20 wurde, in UML-Notation, die Schnitt-stellendefinitionen der beiden abstrakten AS-Klassen *KontaktSensor* und *Motor* dargestellt. In der die Sensor-Klasse *KontaktSensor* die Methode *KontaktZu* definiert. Sie besitzt einen Boole'schen Rückgabewert; er signalisiert den aktuellen Zustand des Kontaktes. In der Aktor-Klasse *Motor* werden die parameterlosen Methoden *MotorStop*, *MotorRechtsLauf* und *MotorLinksLauf* definiert; sie aktivieren die drei Betriebsarten des Motors. Nach der in Kapitel 5.2 beschriebenen Verfahrensweise werden aus den AS-Methoden automatisch komplementäre AS-Methoden abgeleitet (vgl. Abbildung 3.20). Mit diesen komplementären Methoden erfolgt innerhalb der Umgebungsspezifikation die Modellierung des dynamischen Verhaltens der Aktoren und Sensoren.

- **Schritt 4 - Spezifikation des dynamischen Aktor/Sensor-Verhaltens:**

Abbildung B.1
Die elementaren ON-Klassen, die das dynamische Verhalten der Aktoren und Sensoren modellieren



USES ASO: KontaktSensor		
TIMESCALE: 10 s		
Attribute	Action	Guard
	Action1	
	Action2	

USES ASO: Motor		
TIMESCALE: 10 s		
Attribute	Action	Guard
		Guard1
		Guard2
		Guard3

Die Spezifikation der Umgebung muß für jeden identifizierten Aktor bzw. Sensor (fünf im Bahnschrankenbeispiel) eine Instanz einer elementare ON-Klasse bereitstellen. Diese Klassen referenzieren jeweils ein Objekt einer definierten abstrakten AS-Klasse. Für die benötigten fünf Instanzen werden die beiden elementaren ON-Klassen *!KontaktSensorVerhalten* und *!MotorVerhalten* definiert. *!KontaktSensorVerhalten* (siehe linke Seite von Abbildung B.1) beschreibt für alle vier Sensoren, wie sich der Kontakt nach dem Empfang einer Nachricht schließt (durch die Aktion *Action1*) und nach einer festgelegten Wartezeit, die die Trägheit des Kontaktes modelliert, selbständig wieder öffnet (durch die Aktion *Action2*). In *Action1* wird dabei die komplementäre Methode *!KontaktZu* mit einer Eins als Parameter gerufen; in *Action2* wird diese Methode mit einer Null als Parameter gerufen. Die elementare ON-Klasse für den

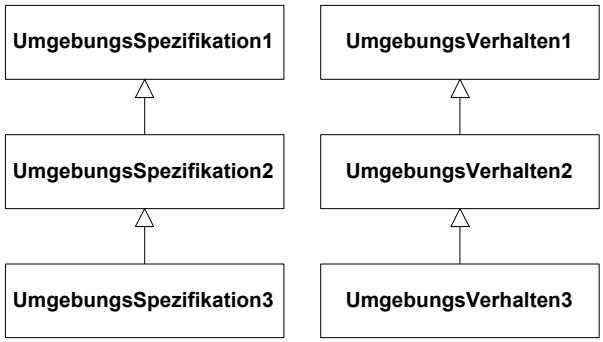
Motor (vgl. rechte Seite von Abbildung B.1) generiert Steuernachrichten an den drei Ports *Stop*, *Hoch*, *Runter*. Die Ports signalisieren einen durchgeführten Wechsel in der Betriebsart des Motors. Drei verschiedene Guards, die jeweils den Boole´schen Rückgabewert einer komplementären Methode (*!MotorStop*, *!MotorRechtsLauf* oder *!MotorLinksLauf*) zurückliefern, entscheiden über Aktivierung des zugehörigen Sendeports. Die spezielle Struktur der Zustandsmaschine beschreibt eine Verriegelung, die verhindert, daß ohne vorheriges Anhalten die Drehrichtung des Motors und somit die Bewegungsrichtung der Schranke geändert werden kann.

• Schritt 5: Erstellen der abstrakten ON-Umgebungsspezifikation:

Die abstrakte ON-Umgebungsspezifikation für das Bahnschrankenbeispiel, die hierarchische ON-Klasse *UmgebungsSpezifikation1*, wurde bereits in Kapitel 3 im unteren Teil von Abbildung 3.4 dargestellt. Für jeden identifizierten Aktor bzw. Sensor enthält diese Klasse eine Instanz der in Abbildung B.1 gezeigten Klassen. Die Instanznamen (z.B. *SensorLinks*) dieser Klassen liefern die Verbindung zu den in Schritt Eins identifizierten Aktoren und Sensoren. Vervollständigt wird die hierarchische Klasse durch eine Instanz (*GleisAbschnitt*) einer abstrakten ON-Klasse (*UmgebungsVerhalten1*), die das gesamte Verhalten der Umgebung abstrahiert. Für jeden Port der Instanzen mit Aktor/Sensor-Bezug besitzt diese abstrakte Klasse einen komplementären Port.

• Schritt 6 - Konkretisieren der Umgebung:

Abbildung B.2
Zwei Konkretisierungsschritte

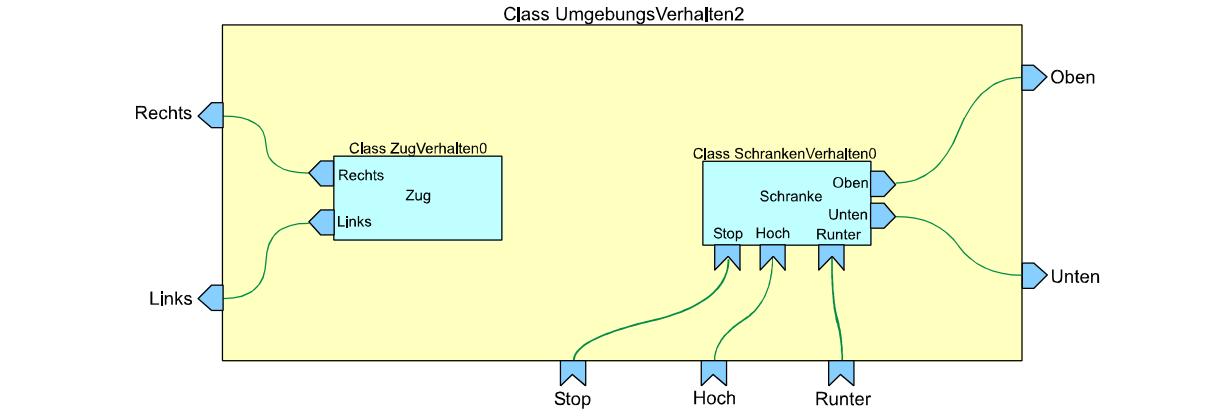


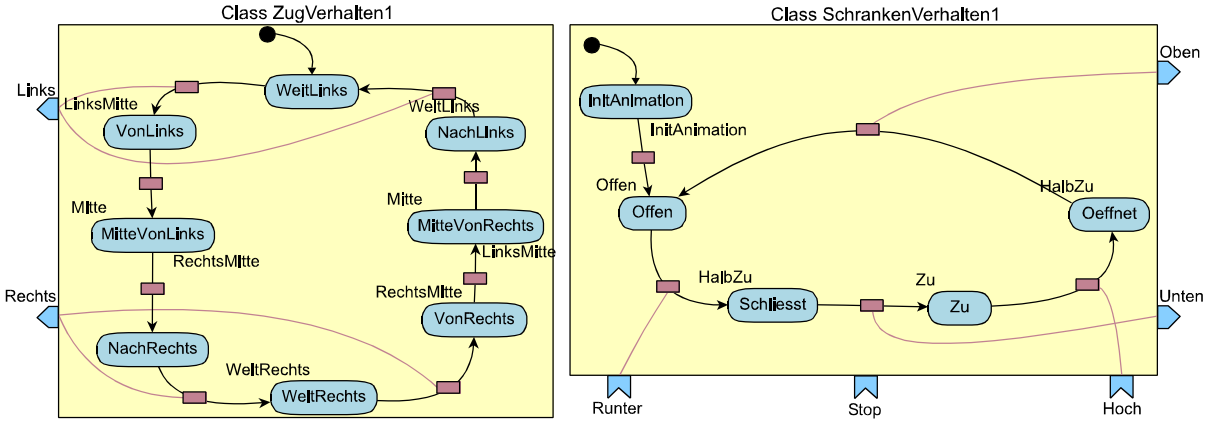
Die abstrakte Umgebungsspezifikation wird schrittweise weiter konkretisiert. Aus der Klasse *UmgebungsSpezifikation1* werden durch Vererbung konkretere Unterklassen abgeleitet. In diesen Klassen wird die Instanz *GleisAbschnitt* durch Instanzen jeweils konkreterer Unterklassen (der Klasse *UmgebungsVerhalten1*) überschrieben (vgl. Abbildung B.2). In der Klasse *Umgebungs-Spezifikation2* konkretisiert die hierarchische Klasse *UmgebungsVerhalten2* die Instanz *GleisAbschnitt*. In dieser Klasse wird das unabhängige Verhalten des Zuges und der Schranke durch zwei Instanzen (*Zug* und *Schranke*) zweier unterschiedlicher abstrakter Klassen (*ZugVerhalten0* und *SchrankenVerhalten0*) verkörpert (vgl. Abbildung B.3). Im zweiten Konkretisierungsschritt wird die Instanz *GleisAbschnitt* durch die Klasse *UmgebungsVerhalten3* konkretisiert (siehe Abbildung B.3). In dieser Klasse werden die Instanzen *Zug* und *Schranke* schließlich durch zwei konkrete Szenarien, die Klassen *ZugVerhalten1* und *SchrankenVerhalten1*, konkretisiert (vgl. Abbildung B.3 unten).

Die elementare ON-Klasse *ZugVerhalten1* (Konkretisierung von *ZugVerhalten0*) modelliert einen Zug, der mit der zulässigen Maximalgeschwindigkeit den Gleisabschnitt wechselweise von links nach rechts und von rechts nach links durchfährt. Spezielle Wartezeiten an den

Zustandswechseln realisieren die Maximalgeschwindigkeit. Bei der Ein- bzw- Ausfahrt des Zuges wird am Steuerport *Links* bzw. *Rechts* eine entsprechende Nachricht abgesendet. Die ebenfalls elementare ON-Klasse *SchrankenVerhalten1* modelliert, stark vereinfacht, die Bewegung der Schranke. Durch die Wartenzeiten an den Zustandswechseln wird eine Schranke mit maximaler Trägheit beschrieben. Die Möglichkeit die Schranke in einer Zwischenstellung anzuhalten wurde bei diesem Szenario außer Acht gelassen.

Abbildung B.3
Die ON-Klassen *UmgebungsVerhalten2*, *ZugVerhalten1* und *SchrankenVerhalten1*





USES ASO: <no Input>		
TIMESCALE: 1 s		
Attribute	Action	Guard
Int pos	Mitte	
	WeitLinks	
	WeltRechts	
	LinksMitte	
	RechtsMitte	

USES ASO: <no Input>		
TIMESCALE: 1 s		
Attribute	Action	Guard
	InitAnimation	
	Zu	
	Offen	
	HalbZu	

Beide elementare ON-Klassen realisieren eine spezielle Variante für den zweiten Konkretisierungsschritt. Indem von der Klasse *UmgebungsVerhalten2* (vgl. Abbildung B.2) weitere Unterklassen, die alternativ zu *UmgebungsVerhalten3* eingesetzt werden können, abgeleitet werden, entstehen weitere, alternative Szenarien. Diese alternativen Szenarien, können wesentlich komplexer ausfallen, ohne daß dies den restlichen Entwurf beeinflußt.

- **Schritt 7 - Erstellen der Animation:**

Die Aktionen in den Klassen von Abbildung B.3 dienen ausschließlich der, in Kapitel 6.5 bereits vorgestellten, Animation (vgl. Abbildung 6.5). Die Aktion *InitAnimation* (Ausschnitt siehe Abbildung 6.6) baut das Animationsszenario auf. Die Aktionen der Klasse *ZugVerhalten1* bewegen die graphische Repräsentation des Zuges. Das Attribut *pos* speichert dabei die x-Koordinate der aktuellen Zugposition. Die Aktion *Mitte* z.B., die Eintrittsaktion der Zustände *MitteVonLinks* und *MitteVonRechts* ist, plaziert mit der Sequenz:

Tcl_AnimationCode Bahnschranke "\$canvas move Train [expr 200-[Get pos]] 0"; Set pos 200

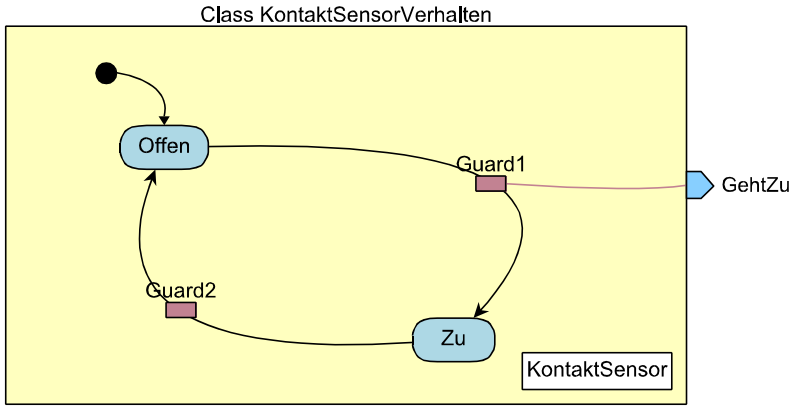
den Zug direkt auf den Bahnübergang (an der Position 200):

Die Aktionen *Zu*, *Offen* und *HalbZu* animieren die durch ein weißes Polygon dargestellte Schranke, indem die Koordinaten der vier Ecken variiert werden. Das Öffnen und Schließen wird jeweils durch eine halboffene Schranke dargestellt. Die Aktion *Offen* z.B. enthält die folgende Codesequenz:

Tcl_AnimationCode Bahnschranke {\$canvas coords Gate 200 50 210 50 210 150 200 150}

- **Schritt 8 - Ableiten der abstrakten ON-Steuerungsspezifikation:**

Abbildung B.4
Die elementare ON-Klasse, die Zugriff auf einen Kontaktsensor hat



USES ASO: KontaktSensor		
TIMESCALE: 1 s		
Attribute	Action	Guard
		Guard1
		Guard2

Aus der abstrakten Spezifikation der Umgebung (*UmgebungsSpezifikation1*) kann nahezu automatisch die abstrakte ON-Steuerungsspezifikation (*SteuerungsSpezifikation1*, siehe oberer Teil von Abbildung 3.4) abgeleitet werden. Sie enthält folglich für jeden, der im ersten Schritt identifizierten Aktoren bzw. Sensoren, eine Instanz; über die Instanznamen werden Koppelstellen zwischen Steuerung und Umgebung festgelegt.

Die Klassen *MotorVerhalten* und *KontaktSensorVerhalten* verhalten sich komplementär zu den Klassen *!MotorVerhalten* und *!KontaktSensorVerhalten.* Die Klasse *MotorVerhalten* wurde bereits in Kapitel 3 (vgl. Abbildung 3.7) dargestellt. Die Klasse *KontaktSensorVerhalten* zeigt Abbildung B.4. Über den Steuersendeport *GehtZu* werden immer dann Nachrichten abgesendet, wenn der entsprechende Kontakt sich gerade geschlossen hat. Die beiden Guards machen über die Sensor-Methode *KontaktZu* den jeweiligen Zustandswechsel vom Zustand des Kontaktsensors abhängig. *Guard1* liefert *true* zurück, wenn der Kontakt geschlossen ist; *Guard2,* wenn der Kontakt offen ist. Zusätzlich zu *Guard2* bedingt eine Wartezeit den Wechsel von *Zu* nach *Offen*. Diese Wartezeit bestimmt die maximale Abtastrate für den Sensor.

Die sechste Instanz (*BahnSchranke*) wird analog zur Umgebungsspezifikation gebildet. An die abstrakte ON-Klasse (*SchrankenSteuerung1*) dieser Instanz werden die Constraints für die Steuerung der Bahnschranke notiert. In Abbildung 3.5 wurden bereits zwei Constraints angegeben, die für den Sicherheitsaspekt verantwortlich sind. Abbildung B.5 zeigt zwei weitere Constraints, die die Lebendigkeit der Steuerung gewährleisten sollen.

Abbildung B.5
Zwei weitere *Constraints* für die Klasse *SchrankenSteuerung1*

```
Constraint Lebendigkeit1
  { receive Oben between 0 t2 after Links && Unten occurred }
Constraint Lebendigkeit2
  { receive Oben between 0 t2 after Rechts && Unten occurred }
```

Die Constraints formalisieren den Zwang, daß spätestens nach *t2* Zeiteinheiten, nachdem der Zug den Abschnitt nach rechts oder links verlassen hat, die Schranke wieder vollständig geöffnet sein muß.

• Schritt 9 - Konkretisierung der Steuerung:

Anlog zur Konkretisierung der Umgebung verläuft die schrittweise Konkretisierung der Steuerungsspezifikation. Die vollständig konkretisierte Steuerungsspezifikation in Form der hierarchischen ON-Klasse *SteuerungsSpezifikation2* entsteht allerdings bereits nach dem ersten Konkretisierungsschritt (vgl. Abbildung B.6). Die abstrakte Instanz *BahnSchranke* wird durch die hierarchische ON-Klasse *SchrankenSteuerung2* konkretisiert.

Abbildung B.6
Ein Konkretisierungsschritt

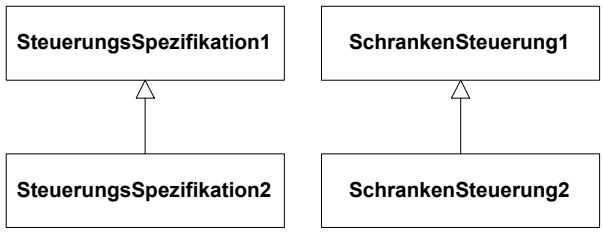
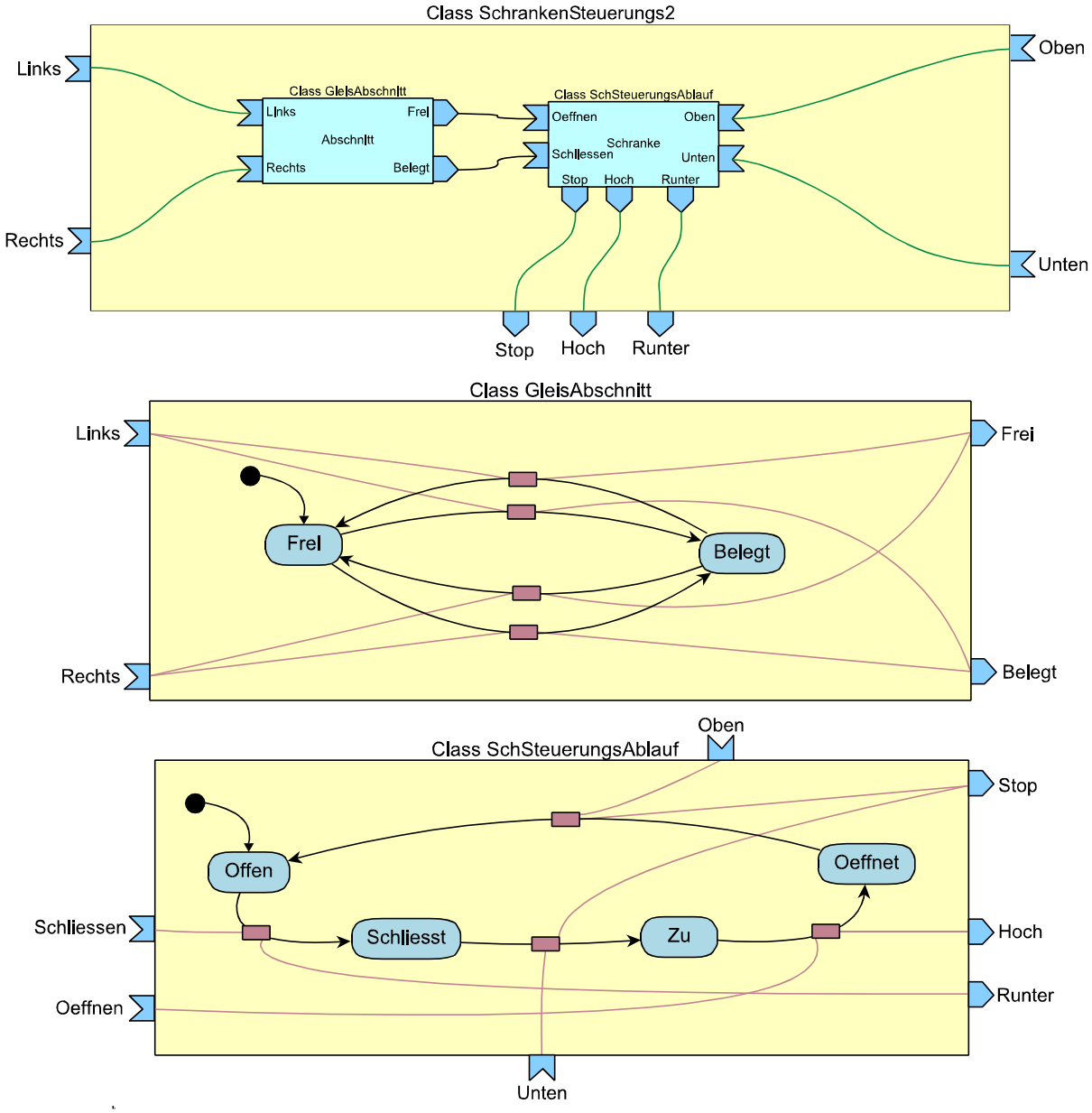


Abbildung B.7
Die elementare ON-Klasse, die Zugriff auf einen Kontaktsensor hat



Eine Instanz (*Abschnitt*) der elementaren ON-Klasse *GleisAbschnitt* ermittelt, ob ein Zug den Gleisabschnitt belegt bzw. ob der Abschnitt frei ist. Ein Wechsel zwischen diesen beiden Zuständen wird über Steuernachrichten an eine weitere Instanz (*Schranke*) übertragen. Die Klasse *SchSteuerungsAblauf* (der Instanz *Schranke*) ermittelt schließlich die Steuernachrichten (*Stop*, *Hoch* und *Runter*) für den Schrankenmotor.

- **Schritt 10 - Plattformdefinition und Mapping:**

Nachdem sowohl die ON-Steuerungs- als auch die ON-Umgebungsspezifikation vollständig konkretisiert wurden, kann eine erste Validierung der Gesamtfunktionalität erfolgen. Hierzu wird die Steuerung automatisch einem einzelnen abstrakten Rechnerknoten zugewiesen.

Für eine Validierung, die das Zeitverhalten einer verteilten Plattform miteinschließt, muß eine Plattformbeschreibung erstellt werden. Die Beschreibung enthält mehrere Instanzen (*Nodes*) abstrakter Rechnerknotenklassen. Zwischen den Knoteninstanzen werden die Kommunikationsverbindungen (*Lines*) festgelegt (vgl. Kapitel 3.4.2). In Kapitel 3.5 wurde bereits eine exemplarische Zuordnung angegeben.

- **Schritt 11 - Validierung:**

Nachdem innerhalb der Projektdefinition die Zeitparameter der Zielplattform an die *Nodes* und *Lines* notiert wurden erfolgt die Validierung.

- **Schritt 12 - Konkretisierung der Plattformbeschreibung:**

Die abstrakten Rechnerknotenklassen werden durch konkrete Unterklassen ersetzt. In diesen Klassen kapseln Objekte konkreter CFU- und PIC-Klassen den direkten Hardware-Zugriff. Die abstrakten CI- und AS-Klassen aus der Oberklasse werden durch konkrete Klassen überschrieben. Vervollständigt wird die Plattformbeschreibung durch die Konfiguration der eingesetzten Compiler.

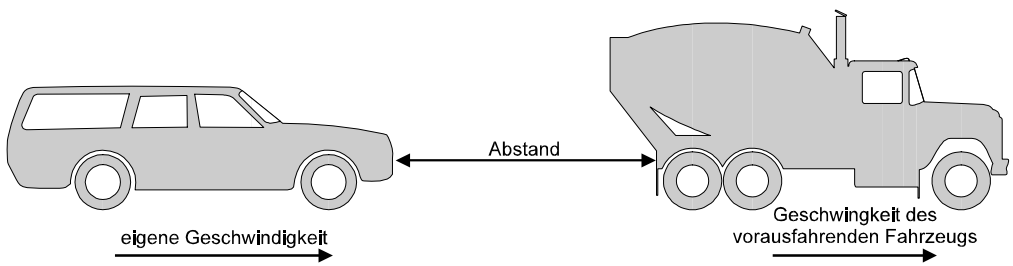
- **Schritt 13 - Code-Generierung:**

Mit den Informationen aus den konkreten Rechnerknotenklassen kann die Code-Generierung für die einzelnen Rechnerknoten automatisiert ablaufen.

B.2 Abstandswarner

Das zweite Demonstrationsbeispiel zeigt den Entwurf eines Abstandswarnsystems für ein Straßenfahrzeug. Aufgabe dieses Systems ist es, während der Fahrt den Abstand zum vorausfahrenden Fahrzeug kontinuierlich zu überwachen; wird der Abstand für ein sicheres Halten zu gering, so wird ein Warnsignal generiert. Abbildung B.8 zeigt das Szenario.

Abbildung B.8
Das Szenario für das Abstandswarnsystem



Das Fahrzeug (ein PKW) in dem sich der Abstandswarner befindet und das vorausfahrende Fahrzeug (ein LKW) fahren mit unterschiedlichen Geschwindigkeiten. Die eigene Geschwindigkeit und der Abstand zum zweiten Fahrzeug werden durch Sensoren ermittelt. Ein Signal im Fahrzeug zeigt den Fahrer eine Unterschreitung des kritischen Abstands an.

Abbildung B.9
Die ON-Spezifikation für die Steuerung und die Umgebung des Abstandswarnsystems

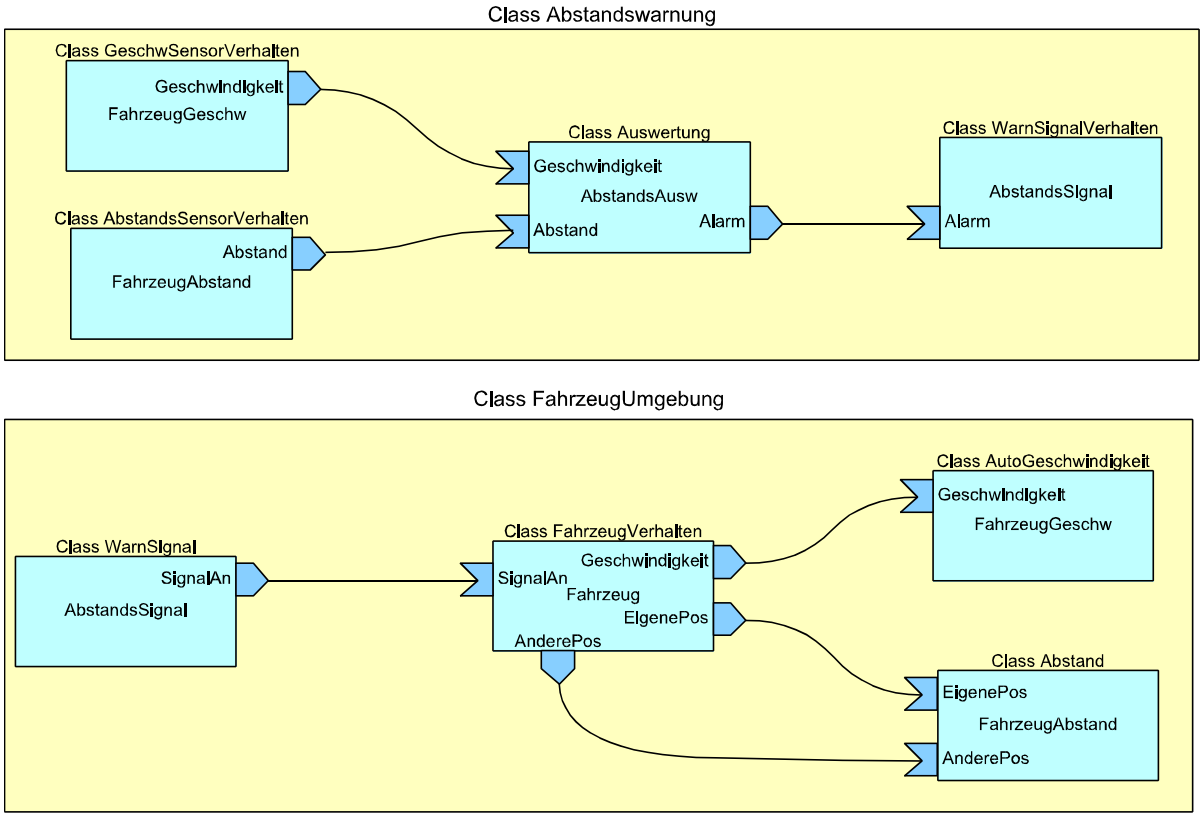
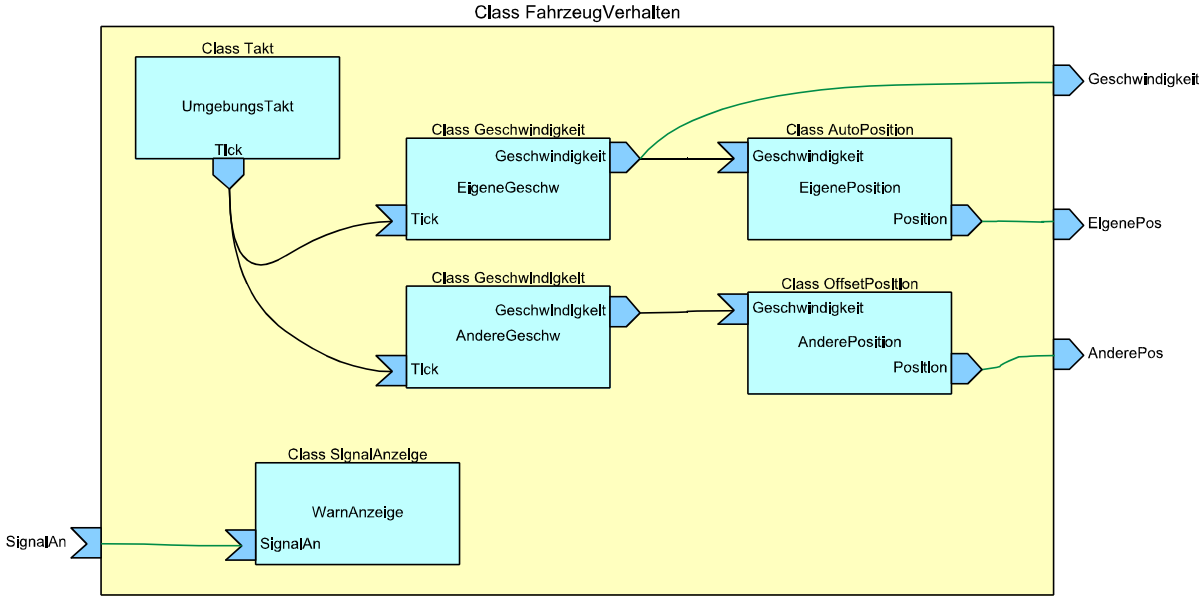


Abbildung B.9 zeigt die gesamte Objektnetz-Spezifikation. Im oberen Teil der Abbildung definiert die hierarchische ON-Klasse *Abstandswarnung* die Steuerung. Im unteren Teil beschreibt die hierarchische Klasse *FahrzeugUmgebung* ein Umgebungsszenario für diese Steuerung. Die drei elementaren ON-Instanzen *AbstandsSignal*, *FahrzeugGeschw* und *FahrzeugAbstand* modellieren das Verhalten der identifizierten Aktoren und Sensoren. In der Steuerung existieren korrespondierende Klassen mit gleichen Instanznamen. Die Klasse *WarnSignal* referenziert die abstrakte AS-Klasse *SignalAktor*, die eine Aktor-Methode mit Boole'schen Übergabeparameter besitzt. Über diese Methode wird durch die Steuerung eine Warnanzeige für zu geringen Abstand aktiviert (*true*) bzw. deaktiviert (*false*). Die Klasse *AutoGeschwindigkeit* referenziert die AS-Klasse *GeschwindigkeitsSensor*, die eine Sensor-Methode mit *int* als Rückgabeparameter besitzt. Über den Datenport (Datenklasse: *double*) nimmt die Klasse Geschwindigkeitswerte entgegen. Eine Aktion konvertiert diese auf *int* und ruft mit diesem Wert die komplementäre AS-Methode. Die Klasse *Abstand* ermittelt den Abstandswert (*int*) der durch den Abstandssensor (AS-Klasse *AbstandsSensor*) gelesen werden kann. Die von

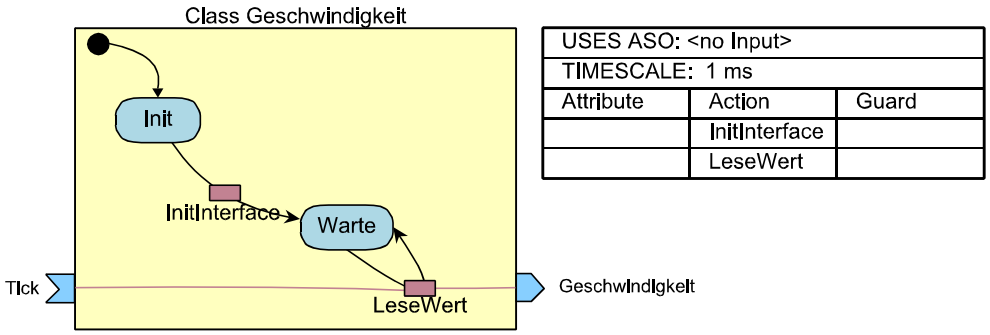
den beiden Datenports (*EigenePos* und *AnderePos*) empfangenen *double*-Werte werden hierzu verrechnet und in der Genauigkeit verringert (Simulation eines schlechten Sensors). Zusätzlich wird durch die Klasse ein Tk-Fenster erzeugt, welches den aktuellen Abstand anzeigt.

Abbildung B.10
Das dynamische Verhalten des Szenarios



Die Instanz *Fahrzeug* der hierarchischen Klasse *FahrzeugVerhalten* (vgl. Abbildung B.10) modelliert das dynamische Verhalten des Szenarios. *UmgebungsTakt* generiert kontinuierlich Steuernachrichten in einem festen zeitlichen Abstand. Die beiden Instanzen der Klasse *Geschwindigkeit* (siehe Abbildung B.11) werden durch diese Nachrichten angestoßen, den Geschwindigkeitswert des eigenen bzw. des vorausfahrenden Fahrzeugs, der über ein Steuerelement (einen Schieberegler in Abbildung B.12) durch den Anwender variiert werden kann, auszulesen und über den Datenport zu versenden.

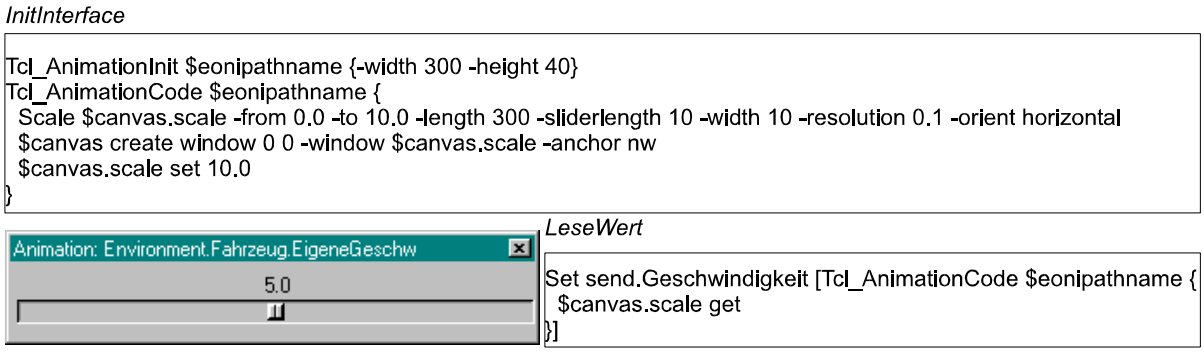
Abbildung B.11
Die elementare ON-Klasse *Geschwindigkeit*



Die Aktion *InitInterface* erzeugt in einem eigenen Animationsfenster ein Tk-Scale-Widget. Der Startwert wird auf 10.0 (m/s) voreingestellt. Abbildung B.12 zeigt die beiden Aktionen und

das Fenster mit dem Scale-Widget, welches bereits auf den Wert 5.0 gestellt wurde. Die Aktion *LeseWert* ließt den Wert aus und sendet ihn über den Port *Geschwindigkeit* aus. Die interne Tcl-Variable *$eonipathname* stellt den vollen Namen (z.B. *Environment.Fahrzeug.EigeneGeschw*) der rufenden Instanz zur Verfügung.

Abbildung B.12
Zwei Aktionen über die eine Nutzerinteraktion ermöglicht wird



In den beiden elementaren Instanzen *EigenePosition* und *AnderePosition* berechnen aus den zyklisch empfangenen Geschwindigkeitsnachrichten die aktuelle Position des eigenen Fahrzeug und des vorausfahrenden Fahrzeug. Die neu berechnete Position wird über einen Datenport (*Position*) ausgesendet. Die elementare ON-Klasse *AutoPosition* besitzt hierzu das Attribut *Position*, welches die berechnete Position speichert. Der Initialwert beträgt 0.0 (m). Die Klasse *OffestPosition* ist eine Unterklasse von *AutoPosition* und erbt folglich die gesamte Funktionalität von *AutoPosition*. Durch die Veränderung des Initialwert des Attributes *Position* von 0.0 auf 10.0 (m) unterscheiden sich beide Klassen. Die Instanz *WarnAnzeige* generiert lediglich eine Ausschrift auf der Konsole, falls eine Steuernachricht empfangen wurde.

Der zentrale Auswertungsteil der Steuerung wird durch die elementare Klasse (*Auswertung*) realisiert. Die beiden *int*-Empfangsports triggern jeweils zwei alternative Zustandswechsel. Eine Nachricht an *Abstand* startet eine Aktion (*NeuerAbstand*), die ein *int*-Attribute (*Abstand*) aktualisiert. Eine Nachricht am Port *Geschwindigkeit* startet die Aktion *NeueGeschwindigkeit*; sie aktualisiert das *int*-Attribute *Geschwindigkeit*. Beide Aktionen führen nach der Aktualisierung eine weitere Berechnung durch. Hierbei wird feststellt, ob der kritische Abstand unterschritten ist. Das Boole‘sche Ergebnis dieser Berechnung wird am *boolean*-Port *Alarm* ausgesendet.

Literaturverzeichnis

[AaBa 97] *van der Aalst, W.M.P.; Basten, T.: Life-Cycle Inheritance, A Petri-Net-Based Approach. Proceedings of the 18th Internatinal Conference on the Application and Theory of Petri Nets, LNCS 1248, p. 62-81, Toulouse, Frankreich, Springer-Verlag, 1997*

[AbeLem 98] *Abel, D.; Lemmer, K.: Theorie ereignisdiskreter Systeme, Tutorium des GMA-Fachausschusses 1.8 „Methoden der Steuerungstechnik", Oldenbourg, 1998*

[AlDi 94] *Alur, R.; Dill, D. L.: A Theory of Timed Automata, Theoretical Comp. Science 126, 1994, pp. 183-235.*

[Arnold 93] *Arnold, Matthias: Feldbussysteme, ELRAD Heft 4, 1993, S. 56-61.*

[AwKuZi 96] *Awad, M.; Kuusela, J.; Ziegler, J.: Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion, Prentice Hall, 1996*

[BAFeDäNü 98] *Ben Achour, K.; Fengler, W.; Däne, B.; Nützel, J.: Modelling and Distributed Simulation of Production of Single Piece and Small-Lot Series Using Fuzzy Coloured Petri Nets; IPMU'98, Paris, Frankreich, Juli, 1998*

[Bender 92] *Bender, K. (Hrsg.): Profibus - Der Feldbus für die Automation, Carl Hanser, München, 1992*

[BenTom 97] *Bender, K.; Tomaszunas, J.: Erfahrungsbericht zur qualitativen Modellierung mechanischer Systeme zum Steuerungstest in Echtzeit, 5. Fachtagung EKA'97, Braunschweig, Mai 1997*

[Blume 95] *Blume, Ralf: Anpassung eins CAN-Bus-Controllers für den Einsatz in einem modularen z8-Microrechnersystem, Ilmenau, Techn. Univ.,Diplomarbeit im Fachbereich Rechnerarchitekturen, 1995, Inv.-Nr.: 200-95D-001*

[Booch 91] *Booch, Grady: Object oriented design with applications, The Benjamin/Cummings Publishing Company, 1991*

[Booch 95] *Booch, Grady: Objektorientierte Analyse und Design: Mit praktischen Anwendungsbeispielen, 1. korrigierter Nachdruck, Addison-Wesley, 1995*

[Burkhardt 94] *Burkhardt, Rainer: Modellierung dynamischer Aspekte mit dem Objekt-Prozeß-Modell, Ilmenau, Tech. Univ., Dissertation, 1994*

[Burkhardt 97] *Burkhardt, Rainer: UML – Unified Modeling Language, Objektorientierte Modellierung für die Praxis, Addison-Wesley, 1997*

[Busse 96] *Busse, Robert: Feldbussysteme im Vergleich: mit 4 Tabellen, Richard Pflaum Verlag, 1996*

[Cooling 91] *Cooling, J. E.: Software Design for Real-time Systems, Chapman and Hall, 1991*

[DeMarco 78] *DeMarco, T.: Structured analysis and system specification. Yourdon Press, Engelwodd Cliffs, NY, USA, 1978*

[Dietrich 98] *Dietrich, Dietmar: LON-Technologie : verteilte Systeme in der Anwendung, Huethig Verlag, Heidelberg, 1998*

[Dijkstra 68] *Dijkstra, E. W.: Cooperating sequentiel processes, in: Genuys, F. (Ed.), Programing Languages, Academic Press, 1968*

[DIN 19245] *PROFIBUS-Norm DIN 19245 Teil 1/2, Beuth-Verlag, Berlin 1991*

[DIN 44300] *DIN 44300: Informationsverarbeitung, Begriffe, Beuth-Verlag, Berlin, 1985*

[Engesser 88] *Engesser, Hermann (Hrsg.): Duden „Informatik" : Ein Sachlexikon für Studium und Praxis; Mannheim, Wien, Zürich; Dudenverlag 1988*

[Etschberger 94] *Etschberger, Konrad: CAN Controller-Area-Network - Grundlagen, Protokolle, Bausteine, Anwendungen, Hanser, 1994*

[FenPhi 91] *Fengler, W.; Philippow, I.: Entwurf industrieller Mikrocomputersysteme, Carl Hanser Verlag, München 1991*

[FIP 88] *Union Techique de l´Electricité (UTE): French Draft Standard Proposal FIP bus C46-601...C46-605; UTE, 1988*

[Fowler 97] *Fowler, Martin: UML Distilled – Applying the Standard Object Modeling Language, Addison-Wesley, 1997*

[GenLau 79] *Genrich, H.; Lautenbach, K.: The analysis of distributed systems by means of predicate/transition nets, LNCS 70, p. 123-146, Springer-Verlag, 1979*

[Gruber 95] *Gruber, Wilhelm: Doppeltes Plus mit Einschränkungen, Erfahrungen beim Einsatz von C++ bei Echtzeitanwendungen, Elektronik 6/95, S. 124*

[Günther 97] *Günther, Manfred: Kontinuierliche und zeitdiskrete Regelungen, Teubner, Stuttgart, 1997*

[Hagen 95] *Hagen, Klaus Ten: Abstrakte Modellierung digitaler Schaltungen, Springer-Verlag, 1995*

[HaLüRa 96] *Hanisch, H.-M.; Lüder, A.; Rausch, M.: Controller Synthesis for Net Condition/Event Systems with Incomplete State Observation, Rensselaer's Fifth Int. Conf. On Computer Integrated Manufacturing and Automation Technology (CIMAT'96), Grenoble, Frankreich, 29.-31. Mai, S. 351-356, 1996*

[Hanisch 92] *Hanisch, H.-M.: Petri-Netze in der Verfahrenstechnik: Modellierung und Steuerung verfahrenstechnischer Systeme, Oldenbourg-Verlag, München, 1992*

[Harel 87] *Harel, David: Statecharts: A visual formalism for complex systems, Science of Computer Programming, Vol. 8, p. 231-274, 1987*

[Harel 90] *Harel, David: Statemate: A working enviroment for the development of complex reactive systems, IEEE Trans. Software Engineering 16(4), p. 403-414, 1990*

[HeLy 94] *Heitmayer, Constance; Lynch, Nancy: The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems. NRL Memorandum Report 7619, Naval Research Laoratory, Washinton D.C., Dezember 1994*

[HenPat 93] *Hennessy, J. L.; Patterson, D. A.: Computer Architecture. A Quatitative Approach, 2nd Ed., Morgan Kaufmann Publishers, Inc., California, 1993*

[HerHom 89] *Herrtwich, R. G.; Hommel, G.: Kooperation und Konkurrenz – Nebenläufig, verteilte und echtzeitabhängige Programmsysteme, Springer Verlag, 1989*

[Hickersberger 95] *Hickersberger, Arnold: C heute : ANSI-C für Ein- und Umsteiger, 2. Aufl. Huethig Verlag, 1995*

[HNSY 94] *Henziger, T. A.; Nicollin, X.; Sifakis, J.; Yovine, S.: Symbolic Model Checking for Real-Time Systems. Information and Computation 111 (2), 1994, pp. 193-244*

[Hoare 85] *Hoare, C. A. R.: Communicating Sequential Processes. Prentice-Hall International, UK, LTD., London, 1985.*

[Hogrefe 89] *Hogrefe, Dieter; Estelle, LOTOS und SDL - Standard-Spezifikationssprachen für verteilte Systeme, Springer-Verlag, 1989*

[Holbe 97] *Holbe, Mario; Konzeption und Implementierung einer Programmierschnittstelle zur Integration von Petrinetz-Toolkits in Interpretersprachen,Ilmenau, Techn. Univ.,Studienarbeit im Fachbereich Rechnerarchitekturen, 1997*

[Hüsener 94] *Hüsener, Thomas: Entwurf komplexer Echtzeitsysteme: state of the art, BI-Wissenschaftsverlag, 1994*

[Hüsener 95] *Hüsener, Thomas: Objektorientierter Entwurf von nebenläufigen, verteilten und echtzeitfähigen Softwaresystemen, Spektrum Akademischer Verlag, 1995*

[IEC 1131-3] *International Electrotechnical Commission: IEC Draft International Standard 1131 for programmable controllers, Part 3 - Programming Languages, 1992*

[Jacobson 92] *Jacobson, I.; u.a.: Object-Oriented Software Engineering, Addison-Wesley, 1992*

[Jensen 92] *Jensen, K.: Coloured Petri Nets, Springer-Verlag, 1992*

[Kahnert 98] *Kahnert, Dirk: Realisierung der Objektpersistenz in einem CASE-Tool über eine SQL-Schnittstelle, Ilmenau, Techn. Univ., Studienarbeit im Fachbereich Rechnerarchitekturen, 1998*

[Knorr 94] *Knorr, Roger: Spezifikation, Verifikation, Leistungsbewertung und Implementierung von Kommunikationsprotokollen mit hierarchischen High-Level-Netzen, Ilmenau, Tech. Univ., Dissertation, 1994*

[KönQuä 88] *König, R.; Quäck, L.: Petri-Netze in der Steuerungstechnick. VEB Verlag Technik, Berlin, 1988*

[KoPrEn 97] *Kowalewski, S.; Preußig, J.; Engell, S.: Analyse zeitbewerteter Bedingung/Ereignis-Systeme mittels Echtzeitautomaten-Tools, 5. Fachtagung EKA'97, Braunschweig, Mai 1997*

[Kovach 97] *Kovach, Waren: User Interface Design With Delphi 3, Prentice Hall, 1997*

[Krapp 88] *Krapp, Michael: Digitale Automaten, VEB Verlag Technik, Berlin 1988*

[KrMaBe 95] *Kriesel, W.; Madelung, O. W.;Bender K.: ASI : the actuator sensor interface for automation, Carl Hanser Verlag, München 1995*

[Langbein 98] *Langbein, Marko; Implementierung eines grafischen Objektnetzeditors, Ilmenau, Techn. Univ., Studienarb. im Fachbereich Rechnerarchitekturen, 1998*

[Laplante 93] *Laplante, P. A.: Real-time systems design and analysis – an engineer's handbook, IEEE Computer society Press, 1993*

[Lawrenz 98] *Lawrenz, W.: CAN Controller-Area-Network, 2. Aufl., Hüttig Verlag, Heidelberg 1998*

[Leler 90] *Leler, W.: Linda meets unix, COMPUTER, Februar 1990*

[Li 93] *Li, Yue: Bewertung der Echtzeitfähigkeit von Feldbus-Systemen, Fortschr.-Ber. VDI Reihe 10. 235, VDI-Verlag, Düsseldorf, 1993*

[LöwFis 97] *von Löwis, Martin; Fischbeck, Nils: Das Python-Buch,Addison-Wesley, 1997*

[Merlin 74] *Merlin, P.: A study of the recoverability od computer systems. Thesis, Dep. Of Information and Computer Science, University California, Irvine, 1974*

[Miguel 96] *Miguel, A.; u.a.:Early validation of real-time systems by model execution, in: 13th Triennial World Congress IFAC, San Francisco, USA, p. 315-320, 1996*

[Mühlpfordt 98] *Mühlpfordt, Angela: Objektorientierte Geschäftsprozeßmodellierung auf der Basis der Unified Modeling Language, Ilmenau, Tech. Univ., Dissertation, 1998 (Draft)*

[Neumann 95] *Neumann, Peter; u.a.: SPS-Standard: IEC 1131: Programmierung in verteilten Automatisierungssystemen, Oldenbourg, München, Wien, 1995*

[NüBlFe 97] *Nützel, J.; Blume, R.; Fengler, W.: Erweitertes Bus-Monitoring, Elektronik 2/97, S.70-73*

[NütBöh 96] *Nützel, J.; Böhme, T.: Analyse und Optimierung von Steuerungssoftware mittels Petri-Netzen. Interne Studie der TU-Ilmenau für die Firma ASEM/Mühlbauer. Ilmenau, Juli 1996.*

[NüDäFe 98] *Nützel, J.; Däne, B.; Fengler, W.:Object Nets for the Design and Verification of Distributed and Embedded Applications, EHPC'98 Orlando, USA, 1998, In: Jose Rolim (Ed.): Parallel and Distributed Processing, S. 953-962. LNCS 1388, Springer Verlag 1998*

[NüFeBö 97] *Nützel, J.; Fengler, W.; Böhme, T.:Objektorientiertes Entwurfsmodell für Steuerungssysteme auf Basis der Petri-Netz-Theorie, 5. Fachtagung EKA'97, Braunschweig, Mai 1997*

[NütFen 95a] *Nützel, J.; Fengler, W.: Analysis and Verification of High-Level-Nets in Combination with Formal Estelle Specification, Workshop of the 16th Int. Conf. on Application and Theory of Petri-Nets, Torino, Italy, June, 1995*

[NütFen 95b] *Nützel, J.; Fengler, W.: Using Formal Description Technics for Fielbus Protcol Implementation, The 11th ISPE/IEE/IFAC Int. Conf. on CAD/CAM Robotics and Factories of the Future, Pereira, Colombia, August 1995*

[NütFen 98] *Nützel, J.; Fengler, W.: World Wide Token Flow Using Object Petri Nets Based on Tcl, Distributed Computing on the Web DCW'98, Rostock, Juni, 1998*

[Oestereich 97] *Oestereich, Bernd: Objektorientierte Softwareentwicklung mit der Unified Modeling Language, 3. akt. Aufl., Verlag R. Oldenbourg, München, 1997*

[Ousterhout 95] *Ousterhout, John K.: Tcl und Tk, Entwicklung grafischer Benutzerschnittstellen für das X Window System, Addison-Wesley, 1995*

[Petri 62] *Petri, C. A.: Kommunikation mit Automaten. Dissertation, Institut für Instrumentelle Mathematik der Universität Bonn, 1962*

[PhiIva 97] *Philippow, I.; Ivanov, E.: Methodische Entwicklung und Anwendung von Frameworks, 42. IWK, Ilmenau, 1997*

[Philippow 87] *Philippow, Eugen (Hrsg.):Taschenbuch Elektrotechnik, Bd. 2. Grundlagen der Informationstechnik, 3. Aufl., VEB Verlag Technik, Berlin, 1987*

[Polke 94] *Polke, M. (Hrsg.): Prozeßleittechnik, Oldenbourg Verlag, München, 1994*

[Pree 95] *Pree, W.: Design Patterns for Object-Oriented Development, Addison-Wesley, 1995*

[Pree 96] *Pree, W.: Framework Patterns,White Paper, SIGS Books, New York 1996*

[Quäck 92] *Quäck, Lothar: Aspeckte der Modellierung und realisierung der Steuerung technologischer Prozesse mit Petri-Netzen, in Schnieder, E. (Hrsg.):Petrinetze in der Automatisierungstechnick, Oldenbourg Verlag, München, 1992*

[Ramchandani 74] *Ramchandani, C.: Analysis of asynchronous concurrent systems by timed Petri nets. MIT, Project MAC, Technical Report 120, 1974*

[Rausch 96] *Rausch, Mathias P.: Modulare Modellbildung, Synthese und Codegenerierung ereignisdiskreter Steuerungssysteme, Magdeburg, Univ., Dissertation, 1996*

[Reisig 82] *Reisig, W.: Petri-Netze. Eine Einführung, Springer-Verlag, 1982*

[Richter 97] *Richter, Thomas: Entwurf eines Entwicklungstools zur automatischen Übersetzung formaler Spezifikationen in Softwarekomponenten für mikrocontollerbasierte Meßsysteme, Ilmenau, Techn. Univ.,Diplomarbeit im Fachbereich Rechnerarchitekturen, 1997, Inv.-Nr.: 200-96D-027*

[RiMaDe 83] *Riedewald, G.; Matuszyński, J.; Dembiński, P.: Formale Beschreibung von Programmiersprachen, R. Oldenbourg Verlag, München, 1983*

[RiNüFe 97] *Richter, T.; Nützel, J.; Fengler, W.: Objektnetze für die Softwaregenerierung von verteilten eingebetten Steuerungssystemen, 42. IWK, Ilmenau, September 1997*

[Roscoe 97] *Roscoe, A. W.: The Theory and Practice of Concurrency, Prentice-Hall, 1997*

[Royce 70] *Royce, W. W.:Managing the Development of Large Software Systems, in Proc. WESTCON, San Francisco, CA, 1970*

[Rumbaugh 91] *Rumbaugh, J; u. a.: Object-Oriented Modeling and Design, Prentice Hall, 1991.*

[SaStHe 92] *Sander, P.; Stucky, W.; Herschel, R.: Automaten, Sprachen, Berechenbarkeit, Teubner Verlag, Stuttgart, 1992*

[Schmidt 98] *Schmidt, A. G.: Konzept und Realisierung eines Übersetzers für hierarchische Objektnetze, Ilmenau, Techn. Univ., Studienarbeit im Fachbereich Rechnerarchitekturen, 1998*

[Schossig 93] *Schossig, Dieter: Mikrocontroller: Aufbau, Anwendung und Programmierung, tewi-Verlag, 1993*

[Schulze 97] *Schulze, Ralf: Konzeption eines objektorientierten Entwurfsmodells für diskrete Steuerungssystem, Ilmenau, Techn. Univ.,Diplomarbeit im Fachbereich Rechnerarchitekturen, 1997, Inv.-Nr.: 200-96D-032*

[ScSoWi 95] *Schöf, S.; Sonnenschein, M.; Wieting, R.: Efficient Simulation of Thor Nets. Proceedings of the 16th Internatinal Conference on the Application and Theory of Petri Nets, LNCS 935, p. 412-431, Turin, Italien, Springer-Verlag, 1995*

[SeGuWa 94] *Selic, B.; Gullekson, G.; Ward, P. T.: ROOM - Real-Time Object-Oriented Modelling, John Wiley & Sons, 1994*

[ShlMel 92] *Shlaer, S.; Mellor, Stephen J.: Object Lifecycles: Modeling the World in States, Yourdon Press, 1992*

[Starke 80] *Starke, P. H.: Petri-Netze: Grundlagen, Anwendungen, Theorie. Verlag der Wissenschaften, 1980*

[Starke 90] *Starke, P. H.: Analyse von Petri-Netz-Modellen, Teubner Verlag, Stuttgart, 1990*

[Staub 95] *Staub, Matthias: Mit dem Worst Case kalkuliert, Elektronik 12/1995, Seite 86-91*

[Stein 97] *Stein, Wolfgang: Objektorientierte Analysemethoden,Vergleich, Bewertung, Auswahl, 2. korr. Aufl., Spektrum Akademischer Verlag, 1997*

[Stroustrup 98] *Stroustrup, Bjarne: Die C++-Programmiersprache, 3. Aufl., Addison-Wesley, 1998*

[Tanenbaum 92] *Tanenbaum, Andrew S.: Computer-Netzewerke, 2. Aufl., Wolfram's Fachverlag, 1992*

[Tanenbaum 95] *Tanenbaum, Andrew S.: Verteilte Betriebssysteme, Prentice-Hall, 1995*

[UIT-T 92] *UIT-T: Z.120 – Message Sequence Charts. UIT-T, 1992*

[UnDäNü 98] *Unger, H.; Däne, B.; Nützel, J.: Experiences Simulating the Load Sharing System LYDIA with High Level PN, HPC'98, Boston, April 1998*

[Venners 97] *Venners, Bill: Inside the Java Virtual Machine, McGraw-Hill, 1997*

[VHDL 88] *IEEE-1076 Standard VHDL Language Reference Manual. The Institute of Electrical and Electronics Engineers Inc., 1988*

[WalNäg 87] *Walther, Hansjoachim; Nägler, Günter: Graphen, Algorithmen, Programme, Fachbuch-Verlag, Leipzig, 1987*

[WaLuSt 92] *Wang, Z.; Lu, H.; Stone, M.: A Message Assigment Algorithm for CAN based Networks, ACM Computer Science Conference March 3-5 1992, Kansas City, in Communication Proceddings, Seite 25 -32*

[Welch 97] *Welch, Brent: Practical programing in Tcl and Tk, 2nd Ed., Prentice Hall, 1997*

[Zöller 91] *Zöller, Horst: Wiederverwendbare Softwarebausteine in der Automatisierungstechnik, VDI Verlag, 1991*

- **Online-Quellen, URLs:**

[FOLDOC 98]	*Howe, Denis: Free On-Line Dictionary of Computing, 1998: http://wombat.doc.ic.ac.uk/*
[GREENHILL 97]	*Green Hills Embedded C++, 1997: http://www.ghs.com/ec++.html*
[HYTECH 98]	*HyTech Homepage, 1998: http://www-cad.eecs.berkeley.edu/~tah/HyTech/*
[ILOGIX 98]	*i-Logix Homepage, 1998: http://www.ilogix.com/*
[KRONOS 98]	*KRONOS Homepage, 1998: http://www.imag.fr/VERIMAG/TEMPORISE/kronos/*
[MAXIM 98]	*Maxim Integrated Products Homepage, 1998: http://www.maxim-ic.com/*
[MOTOROLA 98]	*Motorola Semiconductor Homepage, 1998: http://www.mot-sps.com/*
[OBJECTIME 98]	*ObjecTime Homepage, 1998: http://www.objectime.com/*
[OMG 98]	*Object Managing Group Homepage, 1998: http://www.omg.org/*
[OPNTCL 98]	*Nützel, Jürgen: Opntcl (Object Petri Nets based on Tcl) Homepage, 1998: http://www.theoinf.tu-ilmenau.de/opntcl/*
[OSEK 97]	*OSEK/VDX COM 2.0a, Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug/Vehicle Distributed eXecutive Communication, Version 2.0a, 1997, http://www-iiit.etec.uni-karlsruhe.de/~osek/*
[OSSI 98]	*Nützel, Jürgen: OSSI (Object System Specification Inventory) Homepage, 1998: http://www.theoinf.tu-ilmenau.de/ossi-project/*
[SCRIPTICS 98]	*Scriptics Homepage, 1998: http://www.scriptics.com/*
[TCL 98]	*The Tcl/Tk Consortium Homepage, 1998: http://www.tclconsortium.org/*
[TONI 98]	*Nützel, Jürgen: TONI - Textual Object Notation Interface, 1998: http://www.theoinf.tu-ilmenau.de/ossi-project/toni/*
[UML 97]	*Rational Software Corporation: Unified Modeling Language 1.1, Rational Software Corporation, 1997: http://www.rational.com/uml/*
[UPPAAL 98]	*UPPAAL Homepage, 1998: http://www.docs.uu.se/docs/rtmv/uppaal/*

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Dissertation selbständig angefertigt und alle von mir benutzten Hilfsmittel und Quellen in der Arbeit angegeben habe.

Weiterhin versichere ich, daß ich bisher weder die vorliegende Dissertationsschrift oder Teile von ihr als Prüfungsarbeit oder zum Zweck der Promotion eingereicht bzw. verwendet habe.

Ilmenau, 18.12.1998

Jürgen Nützel

Tabellarischer Lebenslauf

Persönliche Daten:

Name:	Nützel, Jürgen Ralf
Wohnort:	Homburger Platz 4, 98693 Ilmenau
Geburtstag:	16.2.1967
Geburtsort:	Schweinfurt
Familienstand:	Verheiratet, zwei Kinder (geb. 1996 und 1998)

Schulausbildung:

1973	*1977*	Grundschule in Schweinfurt
1977	*1986*	Alexander-von-Humboldt-Gymnasium Schweinfurt Abschluß mit Abitur (Leistungskurse: Mathematik und Physik)

Studium:

1986	*1991*	Studium der Elektrotechnik mit Schwerpunkt Nachrichtentechnik an der Fachhochschule-Würzburg-Schweinfurt
1.9.1990	*28.2.91*	Abschluß mit der Diplomarbeit bei der Siemens AG (Zentralabteilung ZPL 1 IF 32) in Erlangen
März 1988		Aufnahme in den Siemens Studentenkreis
1990	*30.3.94*	Postgraduales Fernstudium Automatisierungstechnik, Vertiefungsrichtung: Technische Informatik / Computertechnik an der TU Ilmenau (Diplom mit dem Prädikat „Ausgezeichnet“)

Beruflicher Werdegang:

1.10.1987	*12.2.88*	1. FH-Praxissemester bei MTS Fresenius, Schweinfurt (Medizin-Technische-Systeme, Fertigung von Dialysegeräten)
6.3.1989	*21.7.89*	2. FH-Praxissemester bei der Siemens AG, Bereich Medizintechnik, Erlangen (Entwicklung von Röntgengeneratoren)
1.5.1991	*30.9.94*	Entwicklungsingenieur im Werk für Kombinationstechnik Fürth der Siemens AG, Bereich Automatisierungstechnik, Entwicklung von analog/digital-gemischten Baugruppen
seit 1.10.94		Wiss. Mitarbeiter am Fachgebiet Rechnerarchitekturen der TU-Ilmenau

Ilmenau, 18.12.1998

Jürgen Nützel

Liste der Veröffentlichungen von Jürgen Nützel, Stand 12/98

[NütFen 95a] *Nützel, J.; Fengler, W.: Analysis and Verification of High-Level-Nets in Combination with Formal Estelle Specification, Workshop of the 16th Int. Conf. on Application and Theory of Petri-Nets, Torino, Italy, June, 1995*

[NütFen 95b] *Nützel, J.; Fengler, W.: Using Formal Description Technics for Fielbus Protcol Implementation, The 11th ISPE/IEE/IFAC Int. Conf. on CAD/CAM Robotics and Factories of the Future, Pereira, Colombia, August 1995*

[NütBöh 96] *Nützel, J.; Böhme, T.: Analyse und Optimierung von Steuerungssoftware mittels Petri-Netzen. Interne Studie der TU-Ilmenau für die Firma ASEM/Mühlbauer. Ilmenau, Juli 1996*

[NüBlFe 97] *Nützel, J.; Blume, R.; Fengler, W.: Erweitertes Bus-Monitoring, Elektronik 2/97, S.70-73*

[NüFeBö 97] *Nützel, J.; Fengler, W.; Böhme, T.:Objektorientiertes Entwurfsmodell für Steuerungssysteme auf Basis der Petri-Netz-Theorie, 5. Fachtagung EKA'97, Braunschweig, Mai 1997*

[RiNüFe 97] *Richter, T.; Nützel, J.; Fengler, W.: Objektnetze für die Softwaregenerierung von verteilten eingebetten Steuerungssystemen, 42. IWK, Ilmenau, September 1997*

[NüDäFe 98] *Nützel, J.; Däne, B.; Fengler, W.:Object Nets for the Design and Verification of Distributed and Embedded Applications, EHPC'98 Orlando, USA, 1998, In: Jose Rolim (Ed.): Parallel and Distributed Processing, S. 953-962. LNCS 1388, Springer Verlag 1998*

[UnDäNü 98] *Unger, H.; Däne, B.; Nützel, J.: Experiences Simulating the Load Sharing System LYDIA with High Level PN, HPC'98, Boston, April 1998*

[NütFen 98] *Nützel, J.; Fengler, W.: World Wide Token Flow Using Object Petri Nets Based on Tcl, Distributed Computing on the Web DCW'98, Rostock, Juni, 1998*

[BAFeDäNü 98] *Ben Achour, K.; Fengler, W.; Däne, B.; Nützel, J.: Modelling and Distributed Simulation of Production of Single Piece and Small-Lot Series Using Fuzzy Coloured Petri Nets; IPMU'98, Paris, Frankreich, Juli, 1998*