Ein flexibler, CORBA-basierter Ansatz für die verteilte, komponentenorientierte Simulation

Dissertation

zur Erlangung des Doktorgrades
am Fachbereich Informatik
der Universität Hamburg

vorgelegt von

Ralf Bachmann
aus Hamburg

Hamburg 2003

Genehmigt vom Fachbereich Informatik der Universität Hamburg

auf Antrag von Prof. Dr. Bernd Page (Universität Hamburg),
Prof. Dr. Winfried Lamersdorf (Universität Hamburg)
und Priv.-Doz. Dr. Thomas Schulze (Universität Magdeburg)

Hamburg, den 3. Dezember 2003 Prof. Dr. Hans Siegfried Stiehl (Dekan)

Zusammenfassung

Im Bereich Modellbildung und Simulation ist die Erstellung von Modellen üblicherweise mit einem hohen Entwicklungsaufwand verbunden. Es ist daher sinnvoll, vorhandene und geprüfte Teile eines komplexen Simulationsmodells (sog. Modellkomponenten) auch in weiteren Simulationsstudien erneut einzusetzen. Die Wiederverwendung von Modellkomponenten über die Grenzen eines einzelnen Simulationssystems hinweg wird jedoch kaum praktiziert, da die Kopplung verschiedener Simulatoren oft eine große technische Hürde darstellt. Bisherige Ansätze wie web-basierte Simulation und verschiedene plattformübergreifende Systeme versuchten zwar, die technischen Randbedingungen für eine Kopplung vorhandener Komponenten zu verbessern, konnten sich aber nicht durchsetzen, weil sie stark abhängig waren von den Werkzeugen eines bestimmten Herstellers. Den aktuellen Stand der Entwicklung auf dem Gebiet der verteilten Simulation verkörpert die Simulationsinfrastruktur „High Level Architecture“ (HLA). Doch selbst hier ist noch keine Interoperabilität verschiedener Realisierungen gegeben. Weiterhin sind die HLA-Mechanismen zur Selbstbeschreibung von Teilmodellen, zur Codierung benutzerdefinierter Daten und zur Durchführung kurzer Simulationsläufe nicht ausgereift. Deshalb ist ein neuer Ansatz notwendig, der die herstellerunabhängige und plattformübergreifende Simulation in verteilten Systemen unterstützt und somit die technischen Hindernisse auf dem Weg zur komponentenorientierten Simulation beseitigt.

Diese Arbeit stellt einen solchen Ansatz namens CoSim (Component Based Distributed Simulation) auf Basis der Kommunikationsinfrastruktur CORBA vor. Dazu entstand eine neue Art von Schnittstellenbeschreibungen für Modellkomponenten, die den Einsatz benutzerdefinierter Datentypen erlaubt. Um den Austausch von Simulationsdaten zu regeln, wurde ein Konzept für Zugangspunkte von Modellkomponenten aus Zeiglers Formalismus „Discrete Event System Specification“ (DEVS) übernommen. Zusätzlich berücksichtigt CoSim verschiedene Phasen der Modellnutzung und ermöglicht feingranulare Synchronisationsverfahren zwischen Modellkomponenten. Die Konzeption von Werkzeugen zum Aufbau von Modellbeschreibungen, zur Komponentenerstellung, Komponentenkopplung und zur Experimentdurchführung, die auf den Schnittstellenbeschreibungen aufbauen, vervollständigt den Gesamtentwurf. Der Ansatz dieser Arbeit ist ein modellunabhängiger Beitrag zur verteilten Simulation. Lösungen für spezielle Simulationsanwendungen (z.B. Simulation von Rechnernetzwerken oder Klimaberechnungen) zeigen eine deutlich bessere Performanz, lassen sich jedoch nur schwer für andere Szenarien adaptieren.

Um die Tragfähigkeit des CoSim-Ansatzes zu demonstrieren, wurde ein Beispielmodell realisiert, das die wesentlichen Leistungen das Gesamtsystems auch tatsächlich abfordert. So sind die Teile des Beispielmodells auf unterschiedlichen Plattformen in verschiedenen Programmiersprachen implementiert. Weiterhin verlangt die Synchronisation der Teilmodelle Maßnahmen zur Vermeidung von Verklemmungen. Dazu wurden alle konzipierten Werkzeuge so tief prototypisch realisiert, dass der Modellbildungs- und Simulationsprozess vollständig unterstützt wird. Damit sind Modellkomponenten als Objekte in einer verteilten Umgebung zugänglich und der Wiederverwendung dieser Objekte stehen keine technischen Hindernisse mehr im Wege.

Abstract

Creation of models for simulation usually entails considerable development efforts. Therefore, it is reasonable to reuse existing valid parts of complex simulation models (model components) in other simulation studies. But reuse is hardly ever seen across the boundaries of a single simulation system since the coupling of different simulators often incorporates large technical obstacles. Past approaches like web-based simulation and some cross-platform systems tried to improve the technical settings for the coupling of model components. None of these approaches were widely accepted as they are tied to products of particular vendors. The „High Level Architecture“ (HLA) is said to be state of the art in the area of distributed simulation, but does not achieve full interoperabilty between different implementations. Furthermore, HLA's mechanisms for describing model parts, coding model specific data and executing multiple short simulation runs are not yet mature. A new approach is needed to support vendor independent, cross-platform simulation in distributed systems, so that the main technical obstacles towards component oriented simulation can be overcome.

This dissertation presents such an approach named CoSim (Component Based Distributed Simulation), which uses CORBA as the underlying communication infrastructure. A new kind of model interface descriptions permits the usage of model specific data structures. To control the exchange of simulation data the port concept of Zeigler's formalism „Discrete Event System Specification“ (DEVS) was adopted. In addition, CoSim regards different phases of model usage and enables model components to execute fine grained synchronisation algorithms for causality assurance. The concept is rounded out by tools for creating model components and their descriptions, for coupling model components, and for executing simulation runs. This approach is a model independent contribution to distributed simulation. Solutions for special application domains of simulation (e.g. internetworking of hosts, or climate calculations) will show a much better performance, but can be hardly adapted to other simulation domains.

To demonstrate CoSim's feasibility a model example was implemented, which claims most of the systems capabilities. The model parts are implemented on different platforms in different programming languages and demand a synchronisation scheme that avoids deadlocks. All tools are realized as deep vertical prototypes, so that the whole cycle of modeling and simulation is supported. Thus, model components become accessible as objects in a distributed environment and technical obstacles for the reuse of these objects are overcome.

Danksagung

Mein Dank gilt zunächst Prof. Dr. Bernd Page, der mir am Fachbereich Informatik der Universität Hamburg im Arbeitsbereich Angewandte und Sozialorientierte Informatik eine Arbeitsatmosphäre ermöglichte, die viel Freiraum für die Umsetzung neuer Ideen bot. Seine Unterstützung im Projekt TIDE erlaubte mir die Einarbeitung aktuelle Anwendungsgebiete der verteilten Systemtechnik. Besonders für seine Betreuung dieses Promotionsvorhabens und seine fachlichen Anregungen danke ich ihm herzlich.

Prof. Dr. Winfried Lamersdorf aus dem Arbeitsbereich verteilte Systeme und Informationssysteme verdanke ich die Hinführung zur verteilten Systemtechnik, zunächst im Rahmen des Projekts TIDE und später in verschiedenen Veranstaltungen seines Arbeitsbereichs, dessen Mitarbeiter mich gerne zu Fachgesprächen eingeladen haben.

Priv.-Doz. Dr. Thomas Schulze aus dem Institut für Technische und Betriebliche Informationssysteme an der Otto-von-Guericke-Universität Magdeburg danke ich für die regelmäßige Ausrichtung des HLA-Forums, dessen Gast ich mehrfach sein durfte. Die Vorträge und Diskussionen dort trugen wesentlich zur Aufklärung offener Fragen im HLA-Ansatz bei.

Ich danke Dr. Steffen Straßburger als ehemaligem Mitglied der Instituts für Simulation und Grafik an der Otto-von-Guericke-Universität Magdeburg für die Bereitstellung der Quellcodes und Erläuterungen zu einem Abfüllmodell, das als Grundlage für das in dieser Arbeit durchgängig verwendete Beispielmodell diente.

Ganz herzlich möchte ich mich bei meinen Kollegen Björn Gehlsen, Nicolas Knaak, Matthias Mayer, Ruth Meyer und Volker Wohlgemuth für die freundschaftliche Zusammenarbeit bedanken.

Thomas Schöllhammer realisierte als Diplomand den größten Teil der Experimentierumgebung aus der vorliegenden CoSim-Umgebung und unterstützte mich mit wichtigen Anregungen zum Gesamtkonzept des CoSim-Ansatzes. Dafür gebührt ihm mein Dank und meine Anerkennung.

Thomas Kaß unterstützte mich wesentlich bei der Endredaktion dieser Arbeit. Für seine fachkundigen Kommentare danke ich ihm herzlich.

Nicht zuletzt möchte ich mich bei meiner Mutter und meinem Onkel Jürgen für die Unterstützung auf meinem Lebensweg insbesondere während der Anfertigung dieser Arbeit bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ALSP	Aggregate Level Simulation Protocol
API	Application Programming Interface
CCM	CORBA Component Model
CDR	Common Data Representation
CGI	Common Gateway Interface
CIDL	Component Interface Definition Language
CLI	Common Language Interface
CLR	Common Language Runtime
CLOS	Common LISP Object System
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CoSim	Component-Based Distributed Simulation
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DESS	Differential Equation System Specification
DEVS	Discrete Event System Specification
DGL	Differentialgleichung
DIS	Distributed Interactive Simulation
DNS	Domain Name Service
DoD	Department of Defense
DTSS	Discrete Time System Specification
ECMA	European Computer Manufacturer's Association
EJB	Enterprise Java Beans
FOM	Federation Object Model
FTP	File Transfer Protocol
GIS	Geographic Information System
GMSL	Graphical MOBILE Scripting Language
GUI	Graphical User Interface
HLA	High Level Architecture
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IEEE	Institute of Electrical & Electronics Engineers
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
IP	Internet Protocol
JDBC	Java Data Base Connectivity

LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
MIME	Multipurpose Internet Mail Extension
MIDL	Microsoft Interface Definition Language
MOBILE	Model Base for an Integrative View of Logistics and Environment
MOM	Message Oriented Middleware
MPI	Message Passing Interface
OLE	Object Linking & Embedding
OMG	Object Management Group
OMT	Object Model Template
ORB	Object Request Broker
PADS	Parallel and Distributed Simulation
POA	Portable Object Adapter
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RTI	Runtime Infrastructure
SOAP	Simple Object Access Protocol
SOM	Simulation Object Model
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
TIDE	Tools for an Integrative View of Distributed Environmental Data
UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator
WSDL	WebServices Description Language
WWW	World Wide Web
XML	Extensible Markup Language

1 Einleitung

1.1 Zielsetzung

Simulation als Methodik kann für die Entscheidungsfindung bei komplexen Fragestellungen oder Systemzusammenhängen eine wichtige Unterstützung leisten. Durch geeignete Werkzeuge wird die Anwendung der Simulation entsprechend vereinfacht, so dass ein Dienstleister, der mit der Durchführung einer Simulationsstudie beauftragt ist, von programmiertechnischen Details bei der Realisierung eines Modellentwurfs ganz oder teilweise befreit wird. Er kann sich mehr auf die Gegebenheiten im Anwendungsgebiet zum Zwecke der Modellbildung sowie auf eine methodische Auswertung der Simulationsergebnisse konzentrieren.

Werkzeuge, die den Prozess der Modellbildung und Simulation vollständig begleiten, wurden von verschiedenen Herstellern bereits bis zur Marktreife entwickelt. Jedoch bieten diese Simulatoren selten die Möglichkeit, Modelle oder Modellteile aus anderen Simulatoren wiederzuverwenden. Häufig werden unterschiedliche Sprachen zur Beschreibung von Modellen benutzt, und eine Konvertierung in andere Sprachen ist nicht vorgesehen. In der Regel sind auch keine Schnittstellen in den Simulatoren vorgesehen, die einen komfortablen Zugriff aus externen Anwendungen heraus auf ein laufendes Simulationsexperiment ermöglichen. Bestehende Simulationssysteme wie EXTEND (www.imaginethatinc.com) oder eM-Plant (www.emplant.de) benutzen zwar immer häufiger einen bausteinorientierten Modellierungsansatz, aber dort sind die erstellten Komponenten nur innerhalb des ursprünglichen Simulationssystems einsetzbar.

Die Verwendung in anderen Simulationsanwendungen ist jedoch aus Sicht eines Modellierers wünschenswert. Üblicherweise ist der Aufwand zur Erstellung von Modellen oder Modellteilen sehr hoch. Einen großen Teil der Arbeit nimmt dabei die Gültigkeitsprüfung ein, da Simulationsexperimente verlässliche Aussagen über den Untersuchungsgegenstand liefern sollen. Eben dieser Aufwand ist nicht erneut zu leisten, wenn bereits gut dokumentierte Modellkomponenten vorliegen und in einer Simulationsstudie wiederverwendet werden können.

Die Verknüpfung von einfacheren Modellkomponenten zu komplexeren Modellen ist nicht unmittelbar auf eine Verminderung der Berechnungszeiten von Simulationsexperimenten ausgelegt. Durch die Kopplung soll in erster Linie der Modellbildungszyklus vereinfacht werden, da bestimmte Modellteile bereits in geprüfter Form vorliegen. Es ist im Gegenteil zu erwarten, dass die Performanz eines komplexen Modells, in dessen Verhaltensberechnung verschiedene Simulatoren einzubeziehen sind, durch zusätzliche Datenübertragungen und Umwandlungsschritte leidet, verglichen mit einer monolithischen Realisierung. Angesichts der stetigen Entwicklung der Rechenkapazitäten lässt sich dieses Hindernis jedoch leichter überwinden als die neuerliche Entwicklung gültiger Modellteile.

In dieser Arbeit wird daher die softwaretechnische Frage behandelt, auf welche Weise sich komplexe Computermodelle aus einfacheren Komponenten

zusammensetzen lassen, wenn diese Komponenten aus den Werkzeugen unterschiedlicher Hersteller stammen. Damit unmittelbar verbunden ist die Suche nach einer geeigneten technischen Plattform zur Bereitstellung und Einbettung von Modellkomponenten. Nicht behandelt wird die Aufarbeitung domänenspezifischen Wissens, etwa in Form von Referenzmodellen oder industriellen Standards. Diese Aufgabe müssen Modellersteller bzw. Modellnutzer übernehmen.

Da sich das Feld der heutigen Simulationsumgebungen sehr heterogen bezüglich Plattformen und Programmier- bzw. Simulationssprachen darstellt, muss ein Ansatz, der sich zur Integration verschiedener Modellkomponenten eignet, viele Grenzen überwinden, wie beispielsweise Plattformen, Programmiersprachen, Netzwerke, Datencodierungen und Modellierungsverfahren. Nur wenn für möglichst vielfältige Rahmenbedingungen eine geeignete technische Unterstützung bereitsteht, kann die Einbindung vorhandener Simulationsumgebungen in eine umfassende Architektur gelingen.

Einen zunächst viel versprechenden Ansatz stellt die *High Level Architecture* (HLA) dar. Diese ursprünglich im militärischen Rahmen entwickelte Menge von Regeln und Schnittstellen wurde jüngst in verschiedenen Standardisierungsgremien (IEEE, OMG) begutachtet und angenommen. HLA lässt jedoch einige technische Details hinsichtlich der Schnittstellenbeschreibung von Modellteilen und der zu verwendenden Schemata zur Datenübertragung und Synchronisation offen. Diese unzureichende Spezifikation hat sich bereits in der Inkompatibilität verschiedener Realisierungen zur HLA-Unterstützung niedergeschlagen, so dass das ursprüngliche Ziel der herstellerunabhängigen Bereitstellung von Modellkomponenten nicht erfüllt werden konnte. In den folgenden Kapiteln werden deshalb andere Ansätze zur Integration von Simulationsanwendungen untersucht und entwickelt. Einen wichtigen Beitrag dazu werden die Komponentenbeschreibungen leisten, die aus der *Discrete Event System Specification* (DEVS) hervorgehen.

1.2 Zu lösende Probleme

Nachfolgend seien kurz die Probleme aufgezeigt, die bei der Erstellung von Modellkopplungen, d.h. bei der Zusammensetzung komplexer Modelle aus einzelnen Modellteilen, auftreten können und die durch geeignete Werkzeuge zu unterstützen sind.

Eine Plattform zur Nutzung von Modellkomponenten, die unterschiedlichen Simulationsumgebungen entstammen, wird mit einem Verweismechanismus auf Modellkomponenten arbeiten, damit nicht sämtliche benutzten Simulatoren auf dem Rechnersystem eines Endanwenders lokal installiert sein müssen, sondern als Dienste von möglicherweise entfernten Zielsystemen einsetzbar sind. Typischerweise wird man also ein heterogenes, verteiltes System als Grundplattform verwenden. Die damit verbundene Nebenläufigkeit einzelner Simulationsprozesse muss jedoch durch Synchronisationsmechanismen gesteuert und dadurch teilweise wieder eingeschränkt werden, damit die Ursache- und Wirkungsbeziehungen innerhalb eines Gesamtmodells zeitlich korrekt ablaufen. Die Untersuchung geeigneter Synchronisationsverfahren für die verteilte Simulation wird deshalb in nachfolgenden Kapiteln thematisiert. Sie bereitet den Einsatz innerhalb eines Gesamtkonzeptes vor.

Um überhaupt Modellkopplungen zu ermöglichen, müssen zunächst einzelne

Modellkomponenten existieren. Deshalb ist ein Rahmensystem sinnvoll, innerhalb dessen sich Modellkomponenten unterschiedlichster Gattung bereitstellen lassen. Die Menge der Operationen, die auf allen Modellkomponenten zulässig ist, definiert einen Typ „Modellkomponente“. Zu diesen Operationen werden zumindest Mechanismen zum Erzeugen und Löschen von Simulationsläufen sowie unterschiedliche Mittel der Datenkommunikation gehören. Da die Realisierung der Operationen je nach eingesetztem Simulator sehr unterschiedlich ausfallen kann, stellen Adapter die Brücke zwischen Rahmensystem und Simulator her. Ein Teilziel wird daher sein, zumindest ein entsprechendes Rahmensystem zuzüglich eines geeigneten Adapters zu entwerfen und zu realisieren.

Ein Modellierer muss die vorhandenen Modellkomponenten natürlich auch auffinden können. Dabei ist eine Klassifikation der Modellkomponenten äußerst hilfreich. Hier bieten sich hierarchische Verzeichnisstrukturen an, die Verweise auf vorliegende Modelle liefern. Auch eine Suchmöglichkeit nach konkreten Modelleigenschaften ist sinnvoll. Ein Modellverweis sollte genaue Dokumentationen über die gefundene Modellkomponente liefern, um die Eignung der Modellkomponente für das übergeordnete Modellierungsszenario zu prüfen. Diese Prüfung wird in der Regel nicht vollständig automatisiert werden können, da sich nicht sämtliche Eigenschaften der Modellkomponente formal beschreiben lassen. Zumindest jedoch beschreibt eine Modellkomponente formal die Schnittstellen, über die ein Datenaustausch möglich ist. Dazu gehören die verwendeten Datentypen und Kommunikationsverfahren, damit die tatsächliche Kommunikation zwischen den Modellkomponenten unabhängig von den speziellen Gegebenheiten ablaufen kann. Sogar modellspezifische Datentypen, die erst nach Erstellung der simulationsunterstützenden Werkzeuge definiert werden, sollten hier zulässig sein, denn erst die Verwendung geeigneter Datentypen ermöglicht ein hohes Abstraktionsniveau bei der Modellierung.

Ein wichtiges Werkzeug für den Simulationsanwender wird ein Kopplungswerkzeug sein, dem die zu koppelnden Modellkomponenten sowie die gewünschten Kommunikationspfade vorgelegt werden. Das Kopplungswerkzeug übernimmt dann die Aufgabe, diese Spezifikation automatisch abarbeiten kann. Weiterhin ist eine allgemeine Experimentierumgebung vorzusehen, um ausführbare, komplette Modelle zu parametrisieren und Simulationsläufe anzustoßen.

1.3 Behandelte Fragen und Thesen

In den folgenden Kapiteln werden die folgenden Fragestellungen behandelt:

1. Wie können Modellkomponenten in einem verteilten System herstellerunabhängig bereitgestellt werden?
2. Wie lassen sich komplexe Simulationsmodelle aus einfacheren Komponenten zusammensetzen?
3. Welche technischen Plattformen sind für eine Komponentenbereitstellung und Modellkopplung geeignet?
4. Lassen sich Werkzeuge realisieren, die eine modellunabhängige Unterstützung bieten?

Zur Beantwortung dieser Fragen werden folgende Thesen aufgestellt.

1. Es ist eine Architektur notwendig, die flexible Schnittstellen enthält, um Modellkomponenten zu beschreiben, und nach Bedarf Referenzen auf ausführbare Modellrealisierungen erzeugt. Der in Kapitel 6 beschriebene Ansatz CoSim definiert eine entsprechende Architektur.

2. In einer Modellkomponente werden typisierte Zugangspunkte explizit ausgewiesen. Ein komplexes Modell entsteht durch Festlegung der Kommunikationswege zwischen diesen Zugangspunkten. Dabei sind Mechanismen vorzusehen, die die Synchronisation der Modellkomponenten untereinander ermöglichen.

3. Auf der Basis von CORBA oder WebServices lässt sich unabhängig von Produkten bestimmter Hersteller oder von Programmiersprachen die vorgeschlagene Architektur umsetzen.

4. Die drei Rahmenwerke Cage (Abschnitt 7.1), CoCo (Abschnitt 7.1) und ExpU (Abschnitt 7.4) unterstützen einen Simulationsdienstleister unabhängig vom vorliegenden Simulationsmodell.

1.4 Beteiligte Informatikdisziplinen

Bei der Kopplung von Modellkomponenten lassen sich Prinzipien aus verschiedenen Informatikdsiziplinen gewinnbringend einsetzen. Modellbildung und Simulation liefern einen methodischen Rahmen, aus dem sich die grundsätzlichen Anforderungen an ein Unterstützungssystem ableiten lassen. Die Grundlagen für eine solide Infrastruktur zur Kommunikation der eingesetzten Simulationssysteme untereinander liefert die verteilte Systemtechnik. Aus der Softwaretechnik ergeben sich schließlich noch Leitlinien für die Strukturierung eines Unterstützungssystems im Großen und der einzelnen Werkzeuge im Kleinen.

1.5 Eigener Zugang zur Fragestellung

Der Zugang zum Fragenkomplex der Modellkopplung in verteilten Systemen ergibt sich für mich fast natürlich aus vorangegangenen Arbeiten. Anfangs beschäftigte ich mich in meiner Diplomarbeit (Kriebisch 1996) mit dem hierarchischen Aufbau von Modellen. In Vorgängerarbeiten (Häuslein 1993) und (Freese und Seidel 1994) wurden dabei bereits verschiedene Modellierungsmethodiken unterstützt. Zur Implementation war der Einsatz verschiedener Programmierparadigmen notwendig, denn das verwendete System „Macintosh Common Lisp“ beherbergte nicht nur den funktionalen Ansatz der Programmiersprache Lisp, sondern auch einen mächtigen Mechanismus zur objektorientierten Programmierung (CLOS, Common Lisp Object System). Das entstandene Werkzeug Dynamis III habe ich dann später im Rahmen des Forschungsprojektes MOBILE (Hilty u. a. 1998) um eine Realisierung der grafischen Beschreibungssprache für Simulationsexperimente (GMSL) erweitert. Die verteilte Ausführung der beschriebenen Simulationsexperimente wurde im Projektrahmen jedoch nicht vollständig umgesetzt.

Im Forschungsprojekt TIDE (Bachmann u. a. 1999) hatte ich dann Gelegenheit, mich intensiv in die Realisierung verteilter Umweltanwendungen einzuarbeiten. Der in dieser Arbeit vorgestellte Ansatz CoSim zur komponentenorien-

tierten, verteilten Simulation greift also frühe Ideen des MOBILE-Projektes wieder auf und vervollständigt die technische Ebene, basierend auf den Erfahrungen im TIDE-Projekt. Den praktischen Umgang mit der verteilten Simulationsarchitektur HLA erlernte ich dann im Rahmen einer Technikstudie (Bachmann 2001), als es darum ging, die HLA-Schnittstellen für eine CORBA-Umgebung, wie sie im TIDE-Projekt verwendet wurde, komfortabel nutzbar zu machen. Einige Unzulänglichkeiten des HLA-Ansatzes ließen sich bei dieser Studie leicht erkennen.

1.6 Vorgehensweise und Aufbau der Arbeit

Um die im Abschnitt 1.3 aufgeworfenen Fragen zu beantworten, werden zunächst verschiedene Lösungsansätze der verteilten Simulation vorgestellt. Hier ist vorrangig der HLA-Ansatz zu diskutieren, da dieser Ansatz den weitesten Fortschritt auf diesem Gebiet darstellt. Nachdem die Defizite bisheriger Ansätze herausgearbeitet sind, soll ein Forderungskatalog den Rahmen für weitere Entwicklungen liefern. Zentrales Thema dieser Arbeit wird dann der Aufbau eines neuen Konzeptes für die komponentenorientierte Simulation sein. Die prototypische Realisierung von unterstützenden Werkzeugen wird einen Querschnitt durch wesentliche Teilprobleme und deren Lösungen bei der Erstellung, Kopplung und Auswertung von Modellkomponenten angehen. Um die Tragfähigkeit des entwickelten Ansatzes nachzuweisen, wird ein komplexes Modell beispielhaft mit Hilfe der neuen Werkzeuge umgesetzt. Dabei wird auch aufgezeigt, wie die verschiedenen Werkzeuge ineinander greifen, ohne modellspezifische Anpassungen vornehmen zu müssen.

Der Aufbau der Arbeit gliedert sich wie folgt in Kapitel: Das zweite Kapitel führt in die Grundbegriffe von Simulation, verteilter System- und Komponententechnik ein. Ein Beispielmodell zur Fassbefüllung, das in weiteren Kapiteln zur Veranschaulichung von Problemen und Lösungen der Simulationstechnik dient, beschreibt Kapitel 3. Daran schließt sich ein Kapitel an, das den bisherigen Stand der Technik auf dem Gebiet der verteilten, komponentenorientierten Simulation aufarbeitet. Eine Bewertung der bisherigen Ansätze und die Rechtfertigung für einen neuen Ansatz findet sich in Kapitel 5. Es folgt die konzeptionelle Beschreibung des neuen Ansatzes namens CoSim. Mit CoSim entstanden einige Werkzeuge, deren prinzipielle Funktionsweise in Kapitel 7 thematisiert wird. Kapitel 8 geht dann auf ausgewählte Probleme und Lösungen bei der Realisierung dieser Werkzeuge ein. Das Beispielmodell aus Kapitel 3 konnte mit Hilfe der neuen Werkzeuge vollständig umgesetzt werden. Einzelheiten der Umsetzung und damit Beispiele zur Werkzeugnutzung gibt Kapitel 9. Kapitel 10 fasst abschließend die erreichten Ergebnisse zusammen und nennt offene Fragen, die sich aus der vorliegenden Arbeit ergeben.

2 Grundlagen und Begriffe

In der verteilten, komponentenorientierten Simulation begegnen sich drei Informatikdisziplinen: die verteilte Systemtechnik, die Softwaretechnik und die Simulation. Zu jedem Gebiet folgen in diesem Abschnitt daher nun Grundlagen und Begriffe, die für die verteilte, komponentenorientierte Simulation wichtig sind.

2.1 Modellbildung und Simulation

Modellbildung und Simulation beschäftigen sich mit der Abbildung von realen oder geplanten Systemen auf Modelle, um Aussagen über die Eigenschaften, Abhängigkeiten und Wirkungensbezüge im ursprünglichen System zu gewinnen (Taylor 2000). Kenntnisse auf dem jeweiligen Anwendungsgebiet des betrachteten Systems sind dabei meist unerlässlich. Da die Modelle oft in einer Form vorliegen sollen, die eine rechnergestützte Auswertung des Modellverhaltens ermöglichen, lässt sich die Simulation als anwendungsnaher Zweig der Informatik auffassen.

Dieser Abschnitt wird nun die wichtigsten Grundbegriffe der Modellbildung und Simulation darstellen, so wie sie in weiteren Kapiteln dieser Arbeit verwendet werden. Eine ausführliche Gegenüberstellung unterschiedlicher Definitionen aus diesem Gebiet findet sich beispielsweise in (Häuslein 1993).

2.1.1 Systeme

Als *System* bezeichnet man ganz allgemein eine Menge von Objekten, die zueinander in Beziehung stehen. Der Systembegriff taucht deshalb in den verschiedensten Zusammenhängen auf. Man spricht beispielsweise von Produktionssystemen, Logistiksystemen, Spielsystemen oder auch von Computersystemen. Es ist üblich, ein System nicht an sich zu betrachten, sondern hinsichtlich eines gewählten Zweckes (wie in Forrester 1972), wodurch die Auswahl der Systemelemente und die Systemgrenze bestimmt werden. Systemelemente sind dabei die als unteilbar betrachteten Teile des Systems. Die Systemgrenze trennt zwischen den Bestandteilen des Systems und seiner Umwelt.

Die Systemelemente bekommen Attribute zugeschrieben, die die besonderen Eigenschaften der einzelnen Elemente herausstellen. Die Gesamtheit aller Attributwerte zu einem Zeitpunkt wird als *Zustand* des Systems bezeichnet. Sind die Attributwerte aller Systemelemente über den gesamten Betrachtungszeitraum hin unveränderlich, so liegt ein statisches System vor. Dynamische Systeme hingegen erlauben eine Änderung des Zustands. Die Menge aller möglichen Zustandsfolgen bezeichnet man als das *Verhalten* des Systems.

Die wesentlichen Aufgaben, die in Zusammenhang mit Systemen auftreten, benennt (Zeigler u. a. 2000, S. 12). Die *Systemanalyse* versucht Einblick in das

Verhalten eines Systems zu gewinnen, in dem die Struktur des Systems untersucht wird, also die Systemelemente und ihre Beziehungen zueinander. Versucht man umgekehrt die Struktur aus dem Verhalten zu rekonstruieren, so wird das als *Systeminferenz* bezeichnet. Beim *Systementwurf* liegt noch kein Gesamtsystem vor. Stattdessen werden verschiedene Zusammenstellungen von vorhandenen Teilen betrachtet, um ein Systemverhalten zu erzeugen, das dem Systemzweck möglichst nahe kommt.

2.1.2 Modelle

Ein Modell stellt das Ergebnis einer vereinfachenden Abbildung eines Systems dar. Von unwesentlichen Einzelheiten des Systems wird dabei abstrahiert und Gruppen von Systemelementen werden zusammengefasst. Oft wird das beobachtete Systemverhalten nur idealisiert abgebildet. Der Modellzweck gibt auch hier die Art und den Grad der Vereinfachungen an. Die Begriffe Modellzustand, Modellverhalten und Modellgrenze lassen sich analog zu den Systembegriffen definieren. Ein einzelnes Attribut, das bei der Berechnung des Modellverhaltens außerhalb der Modellgrenze abgelesen oder verändert werden kann, bezeichnet man auch als *Modellgröße*.

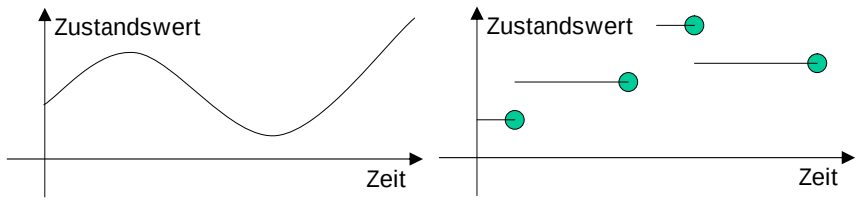
Die Modellierung kann auf unterschiedlichste Arten erfolgen. Deshalb ist eine Klassifikation von Modellen nach diversen Kriterien möglich. Verschiedene Schemata zur Modellklassifikation findet man z.B. in (Page 1991, Abschnitt 1.2) und (Lechler 1996). Materielle Modelle dienen in der Industrie als Vorlage für später zu realisierende Güter (z.B. im Fahrzeugbau). Sie sind meist eine maßstabsgetreue Verkleinerung oder weniger detailgetreue Nachbildung, an dem sich die gewünschten Untersuchungen (z.B. Luftwiderstand) durchführen lassen. Diese Modelle werden hier nicht weiter behandelt, ebensowenig wie grafische Nachbildungen. Stattdessen stehen Modelle im Mittelpunkt, die sich rechnerisch bearbeiten lassen.

Ein wichtiges Unterscheidungsmerkmal ist die Art und Weise, in der eine rechnerische Lösung erreicht wird. Analytische Modelle lassen sich in einem Schritt durch eine geschlossene Formel berechnen. Unter Simulationsmodellen versteht man solche Modelle, die eine Schritt für Schritt Lösung voraussetzen. Im Abschnitt 3.3 ist ein Teilmodell zur Befüllung von Fässern beschrieben, dessen analytische Lösung in Anhang D hergeleitet wird, aus Gründen der Anschaulichkeit jedoch als Simulationsmodell realisiert ist (Abschnitt 9.3). Ähnlich wie Systeme teilt man auch Modelle in dynamische oder statische Modelle ein, je nachdem, ob eine Zustandsänderung erfolgen kann oder nicht. Ein *Simulator* ist ein Werkzeug, das eine dynamische Modellbeschreibung interpretiert und das beschriebene Modellverhalten erzeugt, also Zustandsänderungen im Simulationsmodell herbeiführt (Zeigler u. a. 2000, Abschn. 2.1.4).

Eng verknüpft mit den Zustandsänderungen sind die Zeitpunkte, zu denen die Änderungen eintreten. Bei der Modellbildung muss man verschiedene Zeitbegriffe unterscheiden (Fujimoto 2000). *Physikalische Zeit* bezieht sich auf den Verlauf der Zeit im Realsystem. *Simulationszeit* ist die Abstraktion, die durch ein Modell vorgenommen wird. Dabei ist die Simulationszeit eine lineare Abbildung der physikalischen Zeit. Damit ist sichergestellt, dass Zeitintervalle in den Zeitsystemen ebenfalls korrespondieren. Als *Wanduhrzeit* bezeichnet man die Zeit, die innerhalb eines Simulationsprogramms gilt. Sie wird üblicherweise durch das Betriebssystem von einer Hardwarekomponente abgelesen. Wird die

Simulationszeit von der Wanduhrzeit bestimmt, so spricht man von Simulation in Realzeit. Üblicherweise ist in der Simulation die Simulationszeit gemeint, falls nicht explizit andere Angaben gemacht werden.

Die Darstellung des Zeitverlaufs in dynamischen Modellen ist ein weiteres wichtiges Unterscheidungsmerkmal. Ein kontinuierlicher Verlauf der Zustandsänderungen ist in Abbildung 2.1(a) dargestellt. Hier liegen meist Differentialgleichungen oder Differenzengleichungen zu Grunde. In zeitdiskreten Modellen werden Zustandsänderungen nur zu Einzelzeitpunkten angegeben (Abbildung 2.1(b)); in der dazwischen liegenden Simulationszeit sind die Zustände als konstant anzusehen, oder sie werden überhaupt nicht betrachtet. In der zeitdiskreten Simulation haben sich zudem unterschiedliche Modellierungsstile herausgebildet. In (Page 1991, Abschnitt 2.2) sind die etablierten Sichtweisen Ereignisorientierung, Prozessorientierung und Transaktionsorientierung erläutert. Eine Modellrealisierung im prozessorientierten Stil findet sich auch im Abschnitt 9.2. Einige Einzelheiten zu dieser Sichtweise finden sich dort.



(a) kontinuierlich (b) diskret

Abbildung 2.1: Zustandsänderungen

Unabhängig von der Art der Zustandsübergänge können Zufallseinflüsse vorhanden sein. Modelle mit solchen Einflüssen heißen stochastisch, sonst nennt man sie deterministisch. Häufig anzutreffen sind statische Modelle, die zufallsabhängige Einflüsse besitzen (sog. Monte-Carlo-Modelle). Sie sind ebenfalls zu den Simulationsmodellen zu zählen.

2.1.3 Experimente

Die Beschreibung des Modellverhaltens hat in der Regel einige Freiheitsgrade, damit das Verhalten auch unter verschiedenen Randbedingungen berechenbar bleibt. Um eine Randbedingung bei der Berechnung einfließen zu lassen, wird eine Modellgröße vor Beginn der Berechnung mit einem Wert belegt. Diese Modellgröße nennt man *Parameter*. Die Berechnung des Modellverhaltens unter einer gegebenen Menge von Parameterbelegungen bezeichnet man als *Simulationslauf*. Die *Eingabegrößen* eines Modells sind diejenigen Modellgrößen, deren Wertebelegungen während eines Simulationslaufs Einfluss auf das Modellverhalten nehmen. Modellgrößen, an denen sich das Modellverhalten ablesen lässt, sind *Ausgabegrößen*. Die Werte von *Ergebnisgrößen* ergeben sich durch die Anwendung beliebiger Auswertungen auf tatsächlich beobachtete Ausgabegrößen.

Diese Charakterisierung der Modellgrößen folgt im Wesentlichen (Zeigler 1984), der unter dem Begriff *experimental frame* eine formale Trennung zwischen Modell und Experiment einführte. Führt man diese Trennung konsequent weiter, so lassen sich Experimente auffassen als eine Beschreibung für eine Menge

durchzuführender Simulationsläufe. Innerhalb eines Experiments können durchaus Simulationsläufe mit unterschiedlichen Modellen auftreten. Dies ist besonders nützlich, wenn innerhalb einer Simulationsstudie konkurrierende Modelle untersucht werden sollen, die unterschiedliches Modellverhalten erzeugen.

2.1.4 Modellbildungszyklus

Eng verknüpft mit dem Begriff der Simulation ist der Begriff der Modellbildung. Während die Simulation die Durchführung von Simulationsexperimenten in den Vordergrund stellt, befasst sich die Modellbildung mit dem Übergang von einem System hin zu einem Modell und zu einer berechenbaren Beschreibung des Modellverhaltens. Beide Methoden lassen sich innerhalb eines Prozesses miteinander koppeln. Da dieser Prozess Rückkopplungen enthält, spricht man in diesem Zusammenhang vom *Modellbildungszyklus*.

Abbildung 2.2 ist eine stark vereinfachte Darstellung der Aktivitäten innerhalb des Modellbildungszyklus. Sie reicht aber aus, um diejenigen Schritte zu erläutern, die für die weiteren Kapitel dieser Arbeit relevant sein werden. Eine wesentlich ausführlichere Behandlung des Prozesses, der mit dem Verfahren der Modellbildung und Simulation verknüpft ist, findet sich in (Häuslein 1993) und (Page 1991).

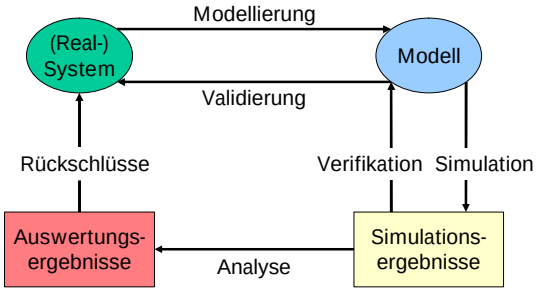


Abbildung 2.2: Modellbildungszyklus

Ausgehend von einem realen oder fiktiven System erfolgt die Modellierung nach einer dem Ziel der Simulationsstudie angemessenen Methode. Ziel dieses Schrittes ist ein Modell, dessen Verhalten in nachfolgenden Simulationsläufen berechnet wird. Ein Simulationslauf für sich liefert zunächst nur einfache Simulationsergebnisse. Deshalb schließt sich in der Regel eine umfangreichere Auswertungsphase an, die die Einzelergebnisse zusammenführt und in Beziehung setzt zu den benutzten Parametrisierungen. Die Auswertung sollte dann Rückschlüsse auf die Vorgänge im ursprünglichen System erlauben.

Damit diese Schlüsse verlässlich sein können, müssen alle vorangegangen Schritte einer Gültigkeitsprüfung unterzogen werden. Die Auswertung ermittelt in der Regel selbst geeignete Kenngrößen, die eine Bewertung des Modellverhaltens erlauben und mit bereits vorhandenen Daten des ursprünglichen System abgeglichen werden. Die Prüfung der Berechnungen des Simulationsmodells bezeichnet man als Verifikation. Im Rahmen der konzeptionellen Validierung soll bestätigt werden, dass das erstellte Modell das behandelte System angemessen repräsentiert. Stellen sich Fehler bei der Durchführung eines Schrittes im

Prozess der Modellbildung oder Simulation heraus, so sind entsprechende Veränderungen in vorangegangenen Schritten durchzuführen.

Der Einsatz von Modellbildung und Simulation kann nach (Page 1991) verschiedene Gründe haben. Ist das zu untersuchende System zu komplex, um es in allen Einzelheiten zu behandeln, so wird es durch die Abstraktions- und Idealisierungsschritte bei der Modellbildung überhaupt erst handhabbar. In manchen Systemen lassen sich die tatsächlichen Vorgänge auch nur schwierig beobachten, weil sie zu schnell oder zu langsam ablaufen. Der Rechnereinsatz kann dann den Ablauf der Zeit im Simulationsmodell steuern. Verursachen Experimente an realen Systemen nicht akzeptable Kosten oder Schäden, so bietet die Simulation eine kostengünstige oder überhaupt erst sinnvolle Möglichkeit zur zielgerichteten Untersuchung dieser Systeme. Planungshandlungen haben fiktive Systeme oder fiktive Randbedingungen zum Gegenstand. Um im Vorfeld bereits Probleme zu erkennen und zu beseitigen, ist eine Bewertung auf der Basis simulierter Daten die einzige Möglichkeit zur Entscheidungsunterstützung.

2.2 Simulation und verteilte Systeme

Ein verteiltes System ist charakterisiert durch (Lamersdorf 1994, Abschnitt 3.1)

- unabhängige, meist heterogene Subsysteme (Prozessoren, Speicher, u.a.)
- Möglichkeiten zur Kommunikation zwischen den Subsystemen
- Fähigkeiten zur Weiterarbeit bei Ausfall von Teilkomponenten
- einen minimalen, gemeinsamen Systemzustand (zumindest um andere vorhandene Komponenten zu erkennen).

Für diese Arbeit werden ausschließlich Systeme von Bedeutung sein, die mehrere Einheiten zur Berechnung des Verhaltens eines Gesamtmodells zur Verfügung stellen, sog. *logische Prozesse* (vgl. Abschnitt 4.1). Systeme, die ausschließlich verteilten Speicher zur Verfügung stellen, sind nicht zu berücksichtigen.

2.2.1 Parallelsimulation

Oftmals nehmen die Berechnungen innerhalb einer Simulationsstudie einen Umfang an, der die Kapazität eines einzigen Prozessors sprengt (Nicol u. a. 1997). Durch verteilte Systeme lässt sich die anfallende Rechenlast auf mehrere Prozessoren aufteilen und somit eine Steigerung der Berechnungsgeschwindigkeit erzielen. Diese Art der Simulation wird in (Fujimoto 2000) als *Parallelsimulation* bezeichnet, wenn spezielle Hardware auf der Basis eines gemeinsamen Speichers verwendet wird. Um Simulationsmodelle in massiv parallelen Systemen, die eine große Anzahl von Prozessoren auf der Basis gleichartiger Hardware bereitstellen, unabhängig vom Vorhandensein gemeinsamen Speichers zu realisieren, verwendet man spezielle Nachrichtenmechanismen wie beispielsweise MPI (MPI-Forum 1994).

2.2.2 Verteilte Simulation

In die *verteilte Simulation* sind mehrere Rechensysteme involviert, die Nachrichten über ein Netzwerk austauschen. Bei der verteilten Simulation ist zu

unterscheiden, ob ein einzelner Simulationslauf noch vollständig auf einem Rechensystem abläuft, oder ob sogar die Berechnung des Modellverhaltens innerhalb eines Simulationslaufs durch mehrere Rechensysteme vorgenommen wird. Parallelsimulation hingegen dient fast immer der Aufteilung eines Simulationslaufs auf die einzelnen Prozessoren. Eine Mischform aus paralleler und verteilter Simulation sowie insbesondere die Maßnahmen zur Koordination der einzelnen logischen Prozesse diskutieren Liu und Nicol (2001).

Ist bereits ein einzelner Simulationslauf für eine Recheneinheit zu umfangreich, so muss man Teile eines Gesamtmodells auf andere Recheneinheiten auslagern. Hier ist jedoch bereits bei der Modellierung eine Modellaufteilung vorzunehmen, was vom Modellentwickler nach (Praehofer u. a. 2000, S. 90) grundlegende Kenntnisse der verteilten Simulation voraussetzt. In der Regel sind für die Modellteilung Konstrukte zur Koordination der einzelnen Modellteile einzufügen, also steigt der Aufwand zur Modellerstellung. Zwischen den einzelnen Modellteilen erfolgt während eines Simulationslaufs ein ständiger Datenaustausch, der je nach verwendetem Kommunikationsverfahren der Recheneinheiten untereinander viel Zeit in Anspruch nehmen kann, was sich durch starke Performanzverluste nachteilig auswirkt. Durch mobilen Code, wie es für die Programmiersprache Java in (Boger 2001) diskutiert wird, kann ein Anwendungssystem diese Aufteilung dynamisch vornehmen. Für die Simulation ließen sich damit die aktuellen Kommunikationsbeziehungen innerhalb des Gesamtmodells ausnutzen, um den notwendigen Datenaustausch zu verringern.

Wird ein Simulationslauf auf einem einzigen Rechensystem durchgeführt, so besteht immer noch die Möglichkeit, mehrere voneinander unabhängige Simulationsläufe innerhalb eines Simulationsexperiments parallel zu berechnen. Eine zentrale Kontrollinstanz sorgt dann für die Zuordnung von Simulationsläufen zu Recheneinheiten und führt die erzielten Ergebnisse zusammen. Im Falle von Optimierungsaufgaben sind beispielsweise geeignete Parameterkombinationen eines Simulationsmodells aufzufinden, die eine benutzerdefinierte Zielfunktion maximieren. Durch die in (Gehlsen und Page 2001) verwendeten Optimierungsstrategien wird eine parallele Ausführung von Simulationsläufen bestmöglichst ausgenutzt.

2.2.3 Interoperable Simulation

Verteilte Systeme leisten aber auch einen Beitrag für andere Zielsetzungen. So ist es häufig notwendig, *Interoperabilität* herzustellen, d.h. den Einsatz verschiedener Hardware- und Softwareplattformen zu überbrücken. Dann kommen virtuelle Maschinen (wie z.B. die Java Virtual Machine, vgl. Lindholm und Yellin 1999) oder Komponentenrahmenwerke (z.B. Microsofts COM, vgl. Box 1998) zum Einsatz. Eine noch allgemeinere Struktur zum Aufbau virtueller Organisationen beschreiben Foster u. a. (2001) mit dem *Grid Computing*. Die von einem Grid verwalteten Ressourcen[1] sind natürlich auch für die Simulation einsetzbar (Gouache und Priol 2000). Der in dieser Arbeit verfolgte Ansatz zur Überwindung von Simulatorgrenzen lässt sich als verteilte, komponentenorientierte Simulation bezeichnen. Komponenten und die damit erhoffte Wiederverwendung stehen im Mittelpunkt des Abschnitts 2.3.

[1] „*CPU cycles*: Use them now, or lose them forever“, David B. Lamkins, `news:comp.lang.lisp.mcl`

2.2.4 Vergleich der Verteilungszwecke

Eine verteilte Durchführung der Simulation kann also unterschiedlichste Hintergründe haben. In Abbildung 2.3 sind die Verteilungszwecke noch einmal anschaulich zusammengefasst. Man erkennt dort Rechenknoten, die durch Workstation-Symbole dargestellt sind. Jeder Rechenknoten hat spezielle Eigenschaften hinsichtlich des verwendeten Modellierungs- und Simulationssystems, des installierten Betriebssystems und der darunterliegenden Hardware.

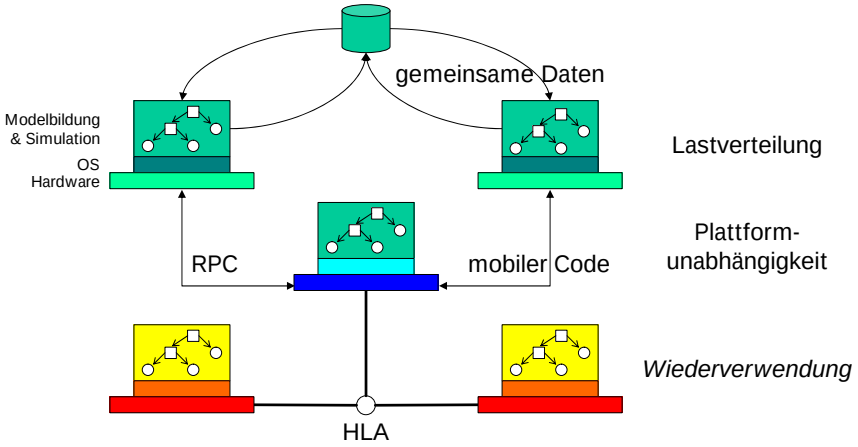


Abbildung 2.3: Verteilungszwecke

Massive Lastverteilung erfolgt meist auf einheitlicher Hardware und verwendet gemeinsamen Speicher. Plattformunabhängige Ansätze der verteilten Simulation fallen unter den Begriff *web-basierte Simulation* und werden in Abschnitt 4.4 ausführlicher behandelt. Dazu wird aber immer noch ein einheitliches Simulationssystem oder ein Rahmenwerk einer bestimmten Programmiersprache verwendet.

Als Zielsetzung der vorliegenden Arbeit wurde in Abschnitt 1.1 jedoch die Wiederverwendung von Simulationskomponenten angestrebt, die sich über verschiedene Simulationssysteme auf heterogenen Plattformen erstreckt. Um dieses Ziel zu erreichen, sind umfassende Architekturen notwendig wie beispielsweise die in 4.2 vorgestellte „High Level Architecture“ oder der in Kapitel 6 vorgestellte Ansatz „CoSim“. Arbeiten, die Parallelisierung vorwiegend aus Performanzgründen betreiben, sollen hier nicht weiter betrachtet werden.

2.3 Komponenten und Wiederverwendung

Die Begriffe *Komponente* und *Wiederverwendung* sind eng miteinander verknüpft. Seit Mitte der 90er Jahre haben sie auch einen festen Platz in der Simulation. Was können die damit verbundenen softwaretechnischen Konzepte also für die Simulation leisten?

2.3.1 Softwaretechnische Sichtweise

Für den Begriff der Software-Komponente hat sich bisher noch keine gefestigte Definition herausgebildet. Auf den kleinsten gemeinsamen Nenner bringt es (Szyperski 2002, Abschn. 1.1):

> One thing can be stated with certainty: components are for composition.

Bei der Komposition setzt man existierende Bausteine zu neuen Gebilden zusammen, die einen Mehrwert gegenüber isolierten Bausteinen ergeben sollen. Mit dem Einsatz von Komponenten geht also immer auch eine *Wiederverwendung* existierender Bausteine einher. Dabei lassen sich verschiedene Arten der Wiederverwendung unterscheiden. Eine *Blackbox*-Abstraktion verhindert den Einblick in die Implementation hinter einer Schnittstelle, eine *Whitebox* hingegen erlaubt den Eingriff in tieferliegende Strukturen. Sind wiederum nur Beobachtungsmöglichkeiten der Implementation vorhanden, so spricht man auch von *Glassboxes*. Wiederverwendung nach dem Blackbox-Prinzip erlaubt es dem Benutzer nicht, sich auf bestimmte Details der Implementation zu verlassen. Wird dieses Prinzip durchbrochen, so liegt eine Whitebox-Wiederverwendung vor, und die benutzten Bausteine können in der Regel nicht gegen andere Bausteine ausgetauscht werden (Szyperski 2002, Abschn. 4.1.5).

Sind zwei Komponenten gekoppelt, so kann man sich die Kopplung als einen Vertrag über die mögliche Kommunikation und die auszutauschenden Daten vorstellen, den beide Komponenten einhalten müssen. Die *Schnittstelle* zwischen den Komponenten definiert dabei die syntaktische Ebene der Kommunikation, d.h. welche Aufrufe überhaupt erlaubt sind und welche Funktionssignatur diese Aufrufe haben. Die Bedeutung eines Aufrufs lässt sich teilweise durch Vor- und Nachbedingungen angeben, meist reichen die formalen Beschreibungsmittel dazu aber nicht aus. Deshalb gehört auch die natürlichsprachliche Beschreibung aller Schnittstellen-Operationen zum Vertrag zwischen den Komponenten.

In Abgrenzung zu Objekten aus der objektorientierten Programmierung spricht man jedoch nur dann von Komponenten, wenn sie einen gewissen Grad an Eigenständigkeit verkörpern. Griffel (1998, S. 31) verbindet diese Eigenschaft direkt mit der Granularität, d.h. mit dem Ausmaß notwendiger Beziehungen zu anderen Komponenten. Die Eigenständigkeit einer Komponente lässt sich auch in anderen Bereichen beobachten, etwa bei der Entwicklung, der Beschaffung oder bei der Installation (Szyperski 2002, Abschn. 1.1).

Da eine Komponente nicht für sich alleine eingesetzt wird, sind zwei Schnittstellen notwendig, um eine Komponente zu beschreiben. Eine Schnittstelle gibt die Funktionalität an, die die Komponente bereitstellt. Diese Funktionalität bezeichnet man auch als *Dienst*. Die zweite Schnittstelle bestimmt, auf welche externen Dienste eine Komponente zurückgreift.

Weiterhin gehört zur Komponentenbeschreibung auch die Angabe des anvisierten Komponentenrahmenwerks, in dem eine Komponente zum Einsatz kommen soll. Ein *Rahmenwerk* im objektorientierten Sinne ist eine Menge kooperierender Klassen, die eine allgemeine Lösung für ähnliche Aufgabenstellungen bereitstellt und ein Ablaufschema vorgibt (Oestereich 1998, Glossar). Bezogen auf die Komponenten stellt das Rahmenwerk die Umgebung dar, in der die Komponenten installiert werden können und bietet Möglichkeiten zur Konfiguration und Kopplung der Komponenten. Weiterhin enthält es die *Infrastruktur*

für die Kommunikation, also den Unterbau, der die grundlegenden Mechanismen zur Verfügung stellt, um Nachrichten zwischen den Komponenten zu versenden. Auch querschnittliche Dienste, die für alle Komponenten einsetzbar sind, können sich im Komponentenrahmenwerk befinden.

Der Einsatz einer Komponente in einem Rahmenwerk unterscheidet sie auch deutlich von *Funktionsbibliotheken*, also einer Sammlung von direkt verwendbarem Code, wie etwa eine Bibliothek für numerische Algorithmen. Diese Funktionen sind nach (Oses u. a. 2002, S. 252) keine Komponenten, weil sie nicht Bestandteil einer Architektur sind. Eine *Architektur* spezifiziert die grundlegende Struktur eines Systems (Oestereich 1998, Glossar), in diesem Fall also Regeln für die Form der Funktionen.

2.3.2 Komponentenrahmenwerke

Zur Zeit gibt es drei Hauptrichtungen für allgemeine Komponentenrahmenwerke:

- Die Object Management Group (OMG), eine Gruppe von mehr als 800 Werkzeugherstellern, definiert CORBA-basierte Standards aus der Perspektive von Unternehmensanwendungen.
- Ausgehend von PC-Systemen entwickelte Microsoft COM-basierte Standards.
- Die Java-basierten Standards stammen von Sun und entwickelten sich aus einer Internet-Sichtweise heraus.

Mit dem neuen CORBA 3 Standard (OMG 2002e) ist ein Komponentenmodell für CORBA entstanden (OMG 2002a), das in seiner frühen Form bereits in (Siegel 2000) existierte. Die wesentlichen Merkmale der Kommunikationsinfrastruktur CORBA sind Unabhängigkeit von Plattform, Programmiersprache und Werkzeughersteller (vgl. Abschnitt 6.1). Dieses CORBA Component Model (CCM) soll hier stellvertretend auch für andere Komponentenrahmenwerke kurz dargestellt werden. Es ist nämlich als Obermenge des Sun-Rahmenwerkes (Enterprise Java Beans, EJB) gedacht und ähnelt Microsofts Rahmenwerk (Component Object Model, COM) zumindest in Bezug auf die mehrfachen Schnittstellen einer Komponente. Weiterhin lassen sich im CCM auch verschiedene Eigenschaften wiederfinden, die (Praehofer und Reisinger 2000) für den Einsatz in der Simulation beschreiben:

- Komponenten lösen aktiv Ereignisse aus, die von anderen Komponenten interpretiert werden können.
- Komponenten lassen sich persistent ablegen.
- Mit einem interaktiven Kopplungswerkzeug lassen sich Komponenten zu größeren Einheiten zusammensetzen.
- Die Eigenschaften einer Komponente lassen sich interaktiv einstellen (z.B. innerhalb einer Experimentierumgebung 7.4).

Für die Simulation bietet das CCM nützliche Dienste an. Dazu gehören verschiedenartige Möglichkeiten des Zugangs zum Zustand und zum Verhalten

einer CORBA Komponente. Die unterschiedlichen Zugänge erlauben eine Unterteilung in Eingabe- und Ausgabegrößen eines Simulationsmodells. Ein Client kann zwischen diesen Zugängen navigieren oder direkt Operationen auf einer Komponente über das `Equivalent Interface` aufrufen.

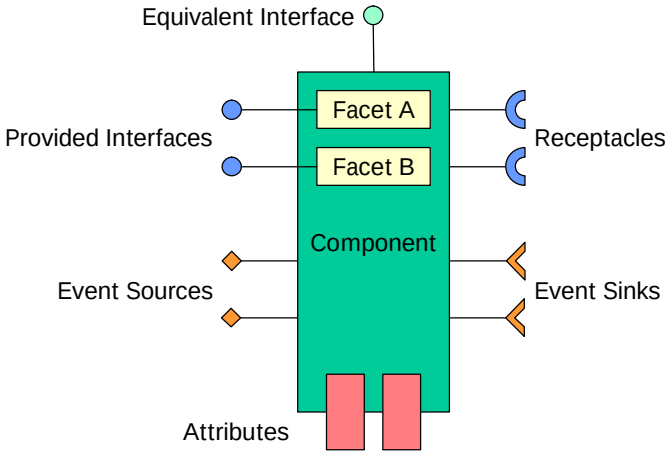


Abbildung 2.4: CORBA Component, nach (Szyperski 2002, S. 251)

Eine CORBA Komponente enthält zunächst verschiedene Objekte, die als `Facetten` bezeichnet werden. Jede Facette erlaubt einem Client, die Methoden der zugehörigen Schnittstelle aufzurufen. Das Gegenstück einer Facette wird als `Receptacle` bezeichnet und ermöglicht der Komponente, selbsttätig Methoden auf Rückrufobjekten beim Client aufzurufen. Wird nur eine einfache Datenübertragung benötigt, bei der keine Parametrisierungen bzw. Rückgabewerte auftreten, so stellen `Attributes` entsprechende Schreib- bzw. Leseoperationen bereit. Die Methodenaufrufe für Attribute blockieren, bis die Methode beim Empfänger vollständig abgearbeitet wurde. Nicht blockierende Datenübertragung kann mit Hilfe von `Event Sources` bzw. `Event Sinks` stattfinden.

Die Nutzung einer CORBA-Komponente lässt sich in zwei Phasen aufteilen. In der ersten Phase erzeugt ein Client eine neue Ausprägung der Komponente und konfiguriert diese Ausprägung. Nachdem die Konfiguration abgeschlossen ist, folgt die operationale Phase. Setzt man ein Simulationsmodell aus Komponenten zusammen, so muss auch ein Simulationslauf innerhalb eines Simulationsexperiments zunächst parametrisiert werden, was einer Konfiguration der verwendeten Komponenten entspricht. In der operationalen Phase wird dann das Modellverhalten berechnet, und die Durchführung von Zustandsübergängen wird entsprechende Schnittstellen der Komponenten benutzen.

Die Rahmenbedingungen für den Aufbau einer Komponente lassen sich mit Hilfe der Component Interface Definition Language (CIDL) genauer beschreiben. Diese Beschreibung erlaubt dann der Komponentenumgebung (dem sog. Application Server), verschiedene Teile für den Zugriff auf die Komponente wie beispielsweise Persistenz oder Transaktionsfähigkeit automatisch zu implementieren.

2.3.3 Komponenten in der Simulation

Unter Komponenten (*component systems*) in der Simulation verstehen Zeigler u. a. (2000, S. 11) Teile eines Gesamtsystems, die zusammenwirken, um das Ausgabeverhalten des Gesamtsystems zu erzeugen. Die einzelnen Komponenten und ihre Kopplung werden dort durch den Formalismus DEVS (vgl. Abschnitt 4.3) beschrieben. Ein Beispiel für die Komposition von Simulationsmodellen führt (Miller u. a. 2001) in der Hafensimulation mit Schiffsbewegungen an. Die Planung der Schiffsbewegungen erfolgt als diskretes Modell und die tatsächlichen Bewegungen hängen mit einem kontinuierlichen Wetter- und Wassermodell zusammen. Hier werden Untermodelle benutzt, um den Detaillierungsgrad zu erhöhen.

Bei unterschiedlichen Simulationsprojekten treten oft ähnliche Modellierungsprobleme auf, jedoch werden die Simulationsmodelle bisher meist vollständig neu realisiert (Praehofer und Reisinger 2000; Heim 1997). Daher fordern Praehofer und Reisinger explizit die Wiederverwendung von früher realisierten Modellkomponenten und den Aufbau von Modellbibliotheken, um den Aufwand einer neuen Modellrealisierung so gering wie möglich zu halten. Die Wiederverwendung kann dabei einzelne parametrisierbare Basiskomponenten, bestehende Subsysteme oder ganze Simulationsanwendungen umfassen. Die einzelnen Komponenten werden dann hierarchisch zusammengesetzt. Der hierarchische Aufbau von Simulationen kann nach (Oses u. a. 2002) helfen, mit der zunehmenden Komplexität von Simulationsmodellen besser umzugehen.

Eine Whitebox-Wiederverwendung in der Simulation setzt voraus, dass die Realisierung tieferer Modellteile bei hierarchischem Modellaufbau aus den höheren Modellteilen heraus zugänglich ist. Damit ist nicht nur die Verhaltensbeschreibung der tieferen Modellteile gemeint, sondern auch der Zugriff auf die Umgebung, in der die Modelle tatsächlich ausgeführt werden. Hierdurch ergibt sich der Vorteil, dass vorhandene Modellteile bei Bedarf leichter an die Gegebenheiten einer speziellen Simulationsstudie anpassbar sind. Man muss jedoch die Offenheit des zugehörigen Komponentenrahmenwerks aufgeben, da das Rahmenwerk konkrete Schnittstellen zu den beteiligten Ausführungsumgebungen vorzusehen hat. Nur wenn die Anzahl unterstützter Umgebungen festgelegt ist, kann auch die Schnittstelle des Rahmenwerks vollständig spezifiziert werden. Ein entsprechendes geschlossenes Simulationssystem (MOOSE: Cubert und Fishwick 1997, 1998a), das verschiedenartige Verhaltensbeschreibungen in einer verteilten Umgebung unterstützt, stellt der Abschnitt 4.4.4 kurz vor.

Die Wiederverwendung von Simulationskomponenten wird später in Kapitel 5 wieder aufgegriffen. Dort findet dann auch eine ausführlichere Diskussion über den Nutzen und die Randbedingungen für eine erfolgreiche Wiederverwendung statt. Eine wichtige Bedingung stellen die Referenzmodelle dar, also eine intensive Aufarbeitung des domänenspezifischen Wissens in Form von Begriffsbildungen und gemeinsamen (Daten-)Strukturen. Da sich diese Arbeit nur mit der Überwindung der technischen Hindernisse bei der Wiederverwendung von Komponenten beschäftigen wird, bleibt es die Aufgabe der Benutzer von Komponentenrahmenwerken, geeignete Aufarbeitungen vorzunehmen.

3 Modellbeispiel Fassbefüllung

Anhand des folgenden Beispielmodells werden die Probleme der verteilten, komponentenorientierten Simulation aufgezeigt. Die darauffolgenden Kapitel beschäftigen sich dann mit möglichen Lösungen. Das Beispiel nimmt das in (Lantzsch u. a. 1999) besprochene Originalmodell zum Vorbild. Dieses Originalmodell diente zur Validierung einer Modellkopplung mit Hilfe der in Abschnitt 4.2 besprochenen Technik HLA. Das im Folgenden besprochene Beispiel ist weitgehend fiktiv, es wurde jedoch versucht, Zahlenangaben den vorhandenen realen Wirtschaftsdaten anzunähern. Da das Modell lediglich der Veranschaulichung von Modellkomponenten und deren Datenaustausch dient, kann auf eine exakte Nachbildung der realen Vorgänge verzichtet werden.

3.1 Gesamtmodell

Zentraler Gegenstand des Beispielmodells sind zu befüllende Fässer. Das Originalmodell betrachtete Fässer, die in einem chemischen Industriebetrieb gefüllt wurden. Hier soll es um Bierfässer gehen, da Mitarbeiter einer großen Hamburger Brauerei auf Nachfrage freundlicherweise geeignete Rahmendaten zur Verfügung stellen konnten (von Polier 2002).

In einer Abfüllanlage werden leere Bierfässer aufgefüllt, dabei stehen mehrere Abfüllstationen zur Verfügung. Ein Transportunternehmen versorgt seine Kunden per LKW mit den gefüllten Fässern. Die Kunden des Transportunternehmens sind Zwischenhändler, die ihren Bedarf an zu liefernden Fässern auf die Verbrauchszahlen der betreuten Gaststätten abstimmen. Die Zwischenhändler wiederum geben angefallenes Leergut bei jeder Lieferung an den Abfüllbetrieb zurück. Eine Beladungskomponente verbindet die beiden Modellteile Befüllung und Transport. Hierbei wählt ein Disponent geeignete Strategien für die Leergutbefüllung und für die Auslieferung von Fässern nach Kundenpriorität. Es existieren also insgesamt drei Modellteile (Befüllung, Beladung, Transport) und ein gekoppeltes Gesamtmodell.

Abbildung 3.1 zeigt eine Gesamtansicht des gekoppelten Modells. Im Rahmen von Simulationsexperimenten soll das zusammengesetzte Modell dann Aufschluss über mögliche Engpässe im Befüllungs- oder Transportprozess geben, indem Kenngrößen wie Maschinenauslastung oder Lieferzeiten ermittelt werden. Die Realisierung des Beispielmodells wird Modellteile enthalten, die mit Hilfe unterschiedlicher Werkzeuge und Programmiersprachen implementiert sind. Die Lauffähigkeit des Gesamtmodells innerhalb der in späteren Kapiteln (6, 7) vorgeschlagenen Architektur zur verteilten, komponentenorientierten Simulation soll damit die Tragfähigkeit eines neuen Konzeptes demonstrieren.

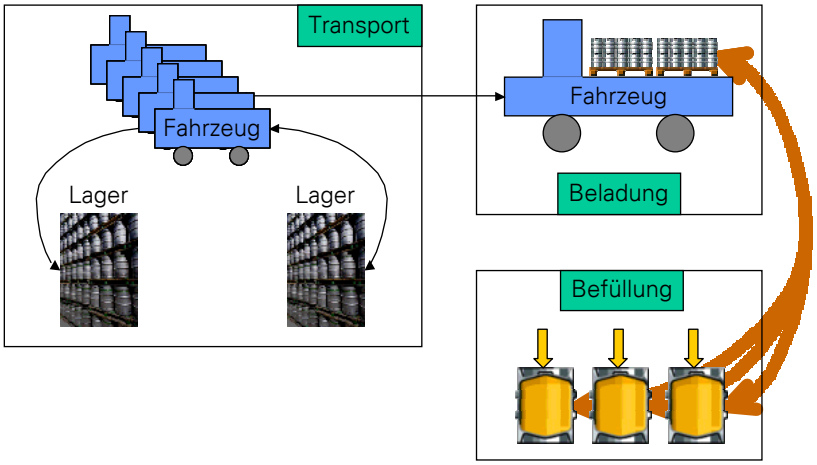


Abbildung 3.1: Gesamtmodell Fassbefüllung

3.2 Bestellung und Transport

Die Transportkomponente sorgt für die Belieferung der Kunden mit Bierfässern. Während eines Simulationslaufs steht eine feste Anzahl von Fahrzeugen zur Verfügung, die jedoch eine unterschiedliche Kapazität aufweisen können. Die Ladung eines Fahrzeugs besteht also aus einer Menge belegter Fass-Stellplätze. Dabei ist es prinzipiell möglich, ein Fahrzeug mit Fässern verschiedener Größen zu beladen. Abbildung 3.2 stellt die verschiedenen Parameter eines Fahrzeugs dar.

In einer einfachen Transportstrategie pendeln die eingesetzten Fahrzeuge zwischen Abfüllbetrieb und Kunden. Je Kunde kann die mittlere Fahrtzeit dieser Strecke als normalverteilte Zufallsvariable angenommen werden. Fahrzeuge sind dabei nicht an feste Kunden gebunden. Ein Fahrzeug kehrt nach erfolgter Lieferung mit dem Leergut, das beim Kunden zum Zeitpunkt der Lieferung vorhanden war, direkt zum Abfüllbetrieb zurück und wartet dort auf Bestellungen.

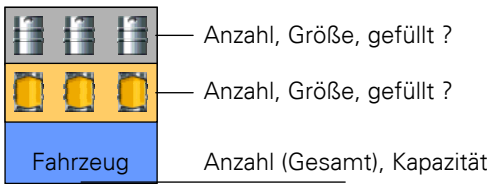


Abbildung 3.2: Fahrzeug im Teilmodell Transport

Das Transportmodell berechnet weiterhin das Lagerungs- und Bestellverhalten der einzelnen Kunden. Dabei wird eine exponentialverteilte Zeit zwischen je zwei Verbrauchsereignissen verwendet. Jeder Kunde bestellt nach der gleichen Lagerhaltungsstrategie neue Fässer. Unterschreitet der aktuelle Lagerbestand durch ein Verbrauchsereignis eine bestimmte Untergrenze, so löst das eine Bestellung beim Abfüllbetrieb aus. Die Bestellmenge berechnet sich aus der Fehlmenge zwischen aktuellem Bestand zum Bestellzeitpunkt und der maximalen

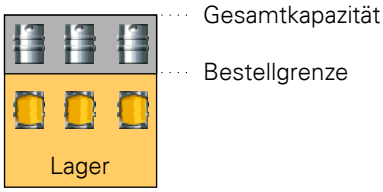


Abbildung 3.3: Lager im Teilmodell Transport

Lagerkapazität (Abbildung 3.3), wobei der mögliche Verbrauch zwischen Bestellzeitpunkt und Lieferzeitpunkt unberücksichtigt bleibt. Die Lagerkapazität und die Bestellgrenze variieren je Kunde.

Kompliziertere Transportstrategien, bei denen komplette Touren für einzelne Fahrzeuge geplant werden, oder verschiedene Bestellstrategien könnten innerhalb der Transportkomponente auch ihren Platz finden. Sie wurden jedoch nicht realisiert, da keine detailgetreue Nachbildung der realen Vorgänge notwendig ist.

3.3 Befüllung

Das Gesamtmodell enthält mehrere Abfüllstationen, die alle die gleiche Struktur und die gleiche Verhaltensbeschreibung aufweisen. Deshalb werden entsprechend viele Exemplare dieser Modellkomponente im Gesamtmodell auftreten. Jedes Modellexemplar einer Abfüllstation wird ggf. mit unterschiedlichen Parametern versorgt, um unterschiedliche Leistungskenngrößen, also den Fülldruck, abzubilden.

Der Füllvorgang ist durch eine Differentialgleichung beschrieben. Es handelt sich also um einen zeitkontinuierlichen Modellteil. Der Fülldruck ist nach oben beschränkt durch die Leistungsfähigkeit der benutzten Füllvorrichtung. Im Originalmodell ist unter Fülldruck die Füllmenge je Zeiteinheit zu verstehen, was üblicherweise als Durchfluss bezeichnet wird. Um nahe an den Bezeichnungen des Originalmodells zu bleiben, wird auch in dieser Arbeit der Begriff Fülldruck anstatt des Begriffs Durchfluss benutzt.

Der Fülldruck soll mit der erreichten Füllhöhe abnehmen, damit gegen Ende des Füllvorgangs ein hoher Fülldruck das Fass nicht überlaufen lässt. Ein minimaler Fülldruck darf auch nicht unterschritten werden, sonst dauert die Befüllung länger als notwendig. So variiert also der Fülldruck zwischen einem maximalen und einem minimalen Wert. Der Einfachheit halber sei eine lineare Abhängigkeit von der Füllhöhe angenommen. Abbildung 3.4 zeigt beispielhaft die Veränderung des Befüllungsdrucks.

Die numerische Lösung von Differentialgleichungen erfolgt durch schrittweise Näherung mit Hilfe von Integrationsverfahren. Eine Besprechung verschiedener Verfahren findet man beispielsweise in (Rechenberg 1972). Durch die lineare Abhängigkeit des Fülldrucks von der Füllhöhe ergibt sich jedoch eine lineare Differentialgleichung erster Ordnung. Damit lässt sich eine geschlossene Formel für die Füllhöhe angeben. Die Formel mit entsprechender Kurve findet sich in Abbildung 3.5. Die allgemeine Lösung kann einschlägigen mathematischen Handbüchern wie beispielsweise (Bronstein und Semendjajew 1991, S. 420) entnommen werden. Eine Herleitung mit Parametern für das Befüllungsmodell ist

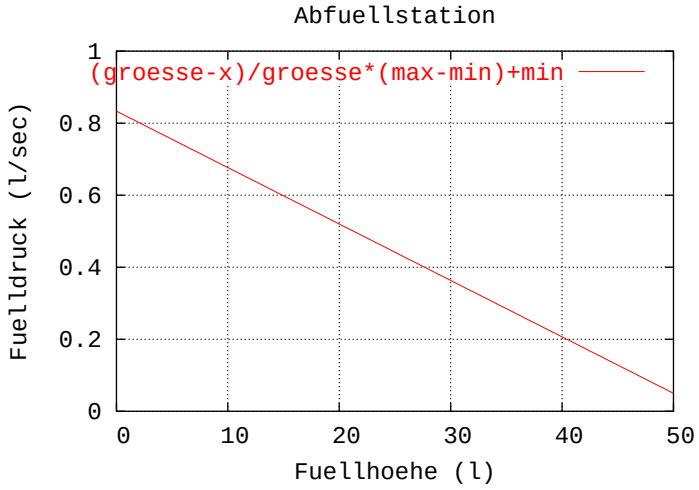


Abbildung 3.4: Veränderung des Befüllungsdrucks (50 Liter Fass)

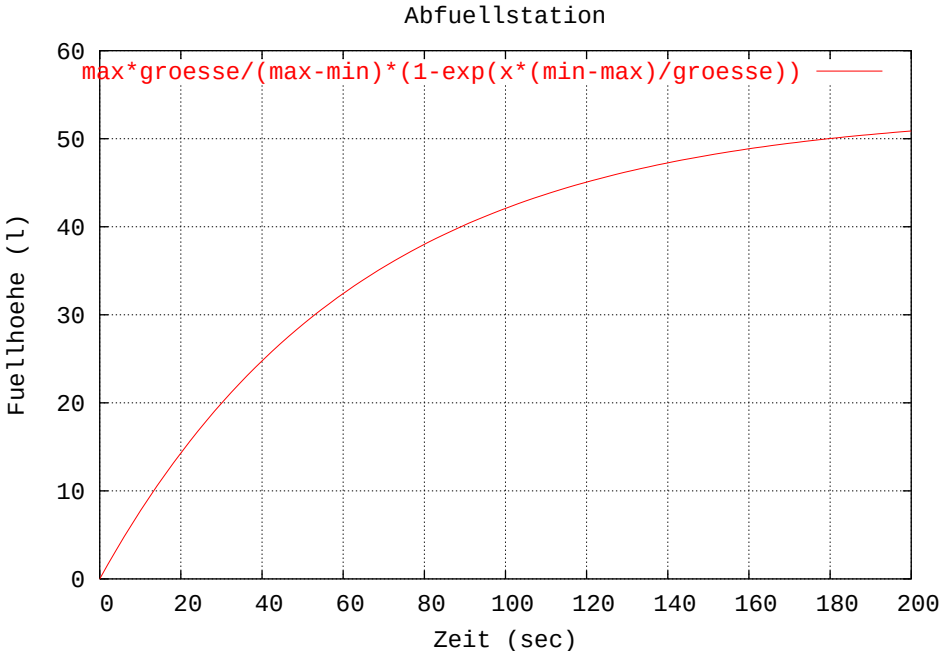


Abbildung 3.5: Veränderung der Füllhöhe (50 Liter Fass)

in Anhang D angegeben. Diese geschlossene Formel kann bei der Verifikation für die Realisierung dieses Modellteils (Abschnitt 9.3) eingesetzt werden.

3.4 Beladung

Die Beladungskomponente verbindet eine Transportkomponente mit mehreren Exemplaren von Abfüllkomponenten. Dazu kann die Beladungskomponente als

ein Zwischenlager auf dem Gelände des Abfüllbetriebes angesehen werden. Hier reihen sich die zurückgegebenen Leerfässer entsprechend ihrer Ankunftszeit auf, und die bereits gefüllten Fässer sind getrennt nach Fassgrößen gelagert. Abbildung 3.6 stellt dieses Zwischenlager mit einzelnen Fässern verschiedener Größen dar. Dabei ist die Anzahl der vorhandenen Abfüllkomponenten beliebig, aber während eines Simulationslaufs festgelegt.

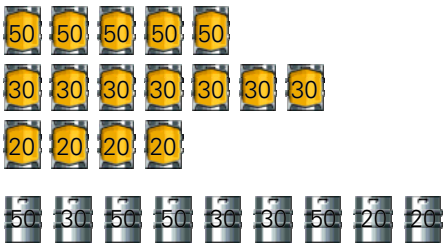


Abbildung 3.6: Teilmodell Beladung

Bei hinreichend großen Fehlmengen im Lagerbestand eines Kunden erfolgt eine Bestellung von Fässern durch die Transportkomponente. In einer einfachen Lieferungsstrategie prüft der Disponent in der Beladungskomponente, ob genügend befüllte Fässer entsprechender Größen vorhanden sind, und ob ein Fahrzeug zum Transport bereit steht. Sobald beide Bedingungen erfüllt sind, erhält die Transportkomponente eine Nachricht, um die bestellten Fässer zum Kunden zu transportieren.

Solange noch Leerfässer vorhanden sind, erhalten die Abfüllstationen reihum je ein Fass (Round-Robin-Verfahren). Nach erfolgter Befüllung gelangt das Fass wieder zur Beladungskomponente und wird dem Zwischenlager mit der zugehörigen Fassgröße hinzugefügt. Nach jeder Fassbefüllung, nach jeder Rückkehr eines Fahrzeugs und bei jeder Bestellung ist erneut zu prüfen, ob Lieferungen ausführbar sind und somit die Transportkomponente zu benachrichtigen ist.

4 Lösungen für die verteilte, komponentenorientierte Simulation

Zu den zentralen Problemen der verteilten, komponentenorientierten Simulation zählen die Kommunikation zwischen den einzelnen Komponenten und die Synchronisation einzelner logischer Prozesse. Da für die Synchronisation schon viele ausgereifte Algorithmen existieren, wird ein eigener Abschnitt einen Überblick über die bestehenden Verfahren und deren Vorteile bzw. Defizite geben. Die vorhandenen Kopplungsansätze zwischen den Komponenten sind jedoch sehr heterogen, entsprechend viele Kommunikationswege finden sich. Eine häufig diskutierte Basis bietet HLA. Sie wird in Abschnitt 4.2 ausführlich behandelt. Auf einer formalen Grundlage basiert der DEVS-Ansatz, der das Konzept von Input- und Output-Ports vorstellt. Dieses Konzept wird in Abschnitt 4.3 vorgestellt und in Abschnitt 6.4 verwendet. Weitere Ansätze entstammen dem Bereich der webbasierten Simulation. Dort finden sich vorwiegend Realisierungen in Java und mit JavaBeans wieder. Herausgehoben werden sollen zudem noch zwei Realisierungen aus dem universitären Umfeld. MOOSE verspricht Plattformunabhängigkeit durch ein eigenes Austauschformat und MOBILE bietet eine explizite Kopplungsbeschreibung sowohl grafisch als auch textuell an.

4.1 Synchronisationsmechanismen

Ein verteiltes Simulationsmodell kann man auffassen als eine Menge nebenläufiger, logischer Prozesse, die untereinander Nachrichten mit Zeitstempeln austauschen. Aus den empfangenen Nachrichten und dem eigenen lokalen Zustand berechnet jeder logische Prozess eine Folge von Simulationsereignissen, die zu lokalen Zustandsveränderungen und zum Versenden weiterer Nachrichten führen können. Der Zeitstempel einer versendeten Nachricht gibt dabei den jeweiligen Simulationszeitpunkt an, zu dem die Nachricht ihre Wirkung entfalten soll. Abbildung 4.1 zeigt die drei Prozesse Bestellung, Beladung und Befüllung, die im Beispielmodell aus Kapitel 3 nebenläufig agieren.

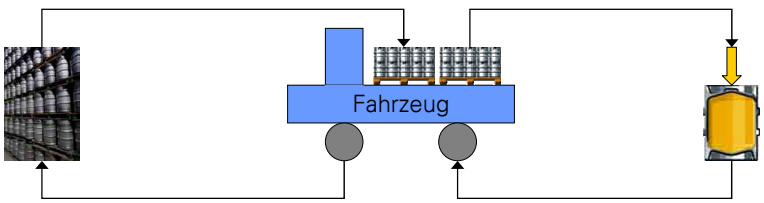


Abbildung 4.1: Logische Prozesse

Im Prinzip ließen sich nun die logischen Prozesse direkt den vorhandenen

Prozessoren eines parallelen oder verteilten Rechensystems zuordnen, und jeder Prozess berechnete seine individuellen Ereignisse. Man könnte zunächst annehmen, dass eine wesentliche Aufgabe der verteilten Simulation eine bestmögliche Zuordnung von Prozessoren zu logischen Prozessen wäre. Dem hält (Mehl 1994) entgegen:

> Das Dilemma bei verteilter Simulation ist, dass ein statisches Mapping von logischen Prozessen auf Prozessoren in der Regel nicht während der ganzen Simulation optimal ist. Eine dynamische Strategie kommt allerdings meist auch nicht in Frage, da das dazu nötige Migrieren von logischen Prozessen sehr zeitaufwändig ist.

Eine Migration ist häufig auch gar nicht möglich, weil der auszuführende Code an eine bestimmte Ausführungsumgebung gebunden ist, die nicht notwendigerweise an jedem beteiligten Rechenknoten verfügbar ist.

4.1.1 Das Synchronisationsproblem

Für die verteilte Simulation reicht es nicht aus, wenn jeder logische Prozess nur seine individuellen Ereignisse berechnet. Die einzelnen Prozesse müssen sich koordinieren, da jeder Prozess die Ereignisse in der Reihenfolge der Zeitstempel abarbeiten muss und dabei nicht nur die lokalen Ereignisse zu berücksichtigen hat, sondern auch die Ereignisse, die von Nachrichten der anderen Prozesse herrühren. Wird diese Abarbeitungsreihenfolge nicht berücksichtigt, so können spät eintreffende Ereignisse Auswirkungen haben, die bereits bearbeitete Ereignisse betreffen. Berechnungsfehler, die aus solchen Situationen entstehen, heißen *Kausalitätsfehler*. Das Problem, eine korrekte Abarbeitungsreihenfolge zu gewährleisten, heißt *Synchronisationsproblem*.

Abbildung 4.2 zeigt eine Situation, in der ein Berechnungsfehler aufgetreten ist. Der Bestell-Prozess sandte zum Zeitpunkt 10 eine Bestellung an den Belade-Prozess. Die Beladung konnte zum Zeitpunkt 15 erfolgen, nachdem der Befüllungsprozess die benötigten Fässer bearbeitet hatte. Zum Zeitpunkt 20 wird dann die Ankunft des ausliefernden Fahrzeugs im Bestell-Prozess eingeplant. Fragt nun der Bestell-Prozess den Lagerzustand im Simulationszeitpunkt 25 ab, bevor die Nachricht mit der Fahrzeugabfahrt bearbeitet wurde, so führt dies zu einer zusätzlichen, fehlerhaften Bestellung.

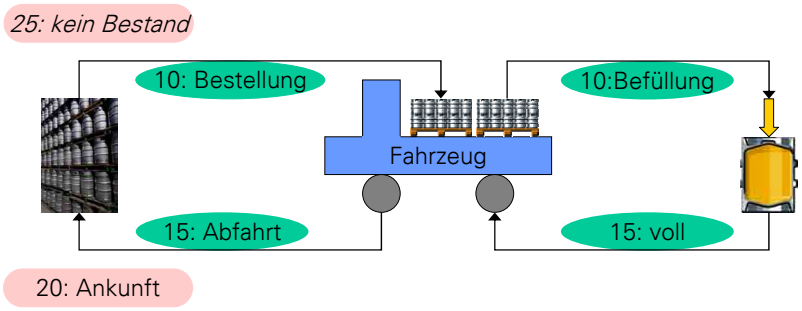


Abbildung 4.2: Kausalitätsfehler

Das Synchronisationsproblem bezieht sich immer auf die modellierte Zeit innerhalb eines Simulationsmodells. Der Abgleich von internen, durch Hardware

unterstützten Uhren, nennt sich nach (Coulouris u. a. 1994, Kap. 10) zwar auch Synchronisationsproblem, soll aber hier nicht weiter betrachtet werden. Die Vermeidung von Kausalitätsfehlern (Kausalitätsprinzip) in der Simulation ähnelt dem Leser-Schreiber-Problem aus (Jessen und Valk 1987, Abschn. 2.4.1) sehr. Bei der Simulation wird jedoch der Graph der Abhängigkeiten erst im Verlauf der Abarbeitung bestimmt und ist stark vom tatsächlichen Datenaustausch abhängig (Praehofer und Reisinger 2000). Es ist also nicht von vornherein bekannt, welche Nachricht welche Zustandsvariable ändern wird, und welche Ausgaben ein Modellteil durch Empfang einer Nachricht erzeugen wird. Deshalb wurden verschiedene spezielle Algorithmen für die Synchronisation in Simulationssystemen entwickelt.

4.1.2 Prinzipien der Synchronisationsverfahren

Einen guten Überblick der verschiedenen Synchronisationsverfahren gibt (Fujimoto 2000). Soweit nicht anders vermerkt, stammen die folgenden Beschreibungen aus diesem Werk. *Optimistische* Synchronisationsverfahren erlauben zunächst eine Abarbeitungsreihenfolge, die nicht alle externen Nachrichten berücksichtigen muss. Die Feststellung von Kausalitätsfehlern führt dann zum Zurücksetzen der beteiligten Prozesse in einen fehlerfreien Berechnungszustand. Jeder Prozess muss also vorangegangene Zustände so speichern, dass ein Zurücksetzen möglich ist. *Konservative* Synchronisationsverfahren vermeiden die Verarbeitung von Ereignissen außerhalb einer gültigen Abarbeitungsreihenfolge gänzlich.

Damit neben den zeitgestempelten Nachrichten keine anderen Abhängigkeiten berücksichtigt werden müssen, schließt man die Verwendung von gemeinsamem Speicher der Prozesse aus. Jeder Prozess ist also für einen bestimmten Teil des Gesamtzustandes verantwortlich. Eine hinreichende Bedingung zur Wahrung der Kausalität im Gesamtsystem ergibt sich dann durch das *Lokalitätsprinzip*. Nach dem Lokalitätsprinzip berechnet jeder logische Prozess für sich allein betrachtet seine Ereignisse in der Reihenfolge aufsteigender Zeitstempel. Dadurch begeht kein Prozess einen Kausalitätsfehler. Das Gesamtergebnis entspricht genau dann dem Ergebnis einer sequentiellen Abarbeitung aller Ereignisse, wenn Ereignisse mit gleichem Zeitstempel im verteilten wie im sequentiellen Fall gleich angeordnet werden.

Ein Synchronisationsverfahren muss also entscheiden, welche Ereignisse nebenläufig bearbeitet werden dürfen oder welche voneinander abhängig sind. Um ein Netz aus Abhängigkeiten zu berechnen und den Ereignissen ohne Abhängigkeiten ihre Abarbeitung zu signalisieren, sind in der Regel zusätzlich zu den Zeitstempeln weitere Informationen notwendig. Die verschiedenen Verfahren unterscheiden sich dadurch, welche zusätzlichen Informationen sie erzeugen oder von den beteiligten Prozessen abfragen.

4.1.3 Konservative Verfahren

Die üblichen konservativen Synchronisationsverfahren gehen davon aus, dass jeder Prozess die Nachrichten an andere Prozesse in der Reihenfolge aufsteigender Zeitstempel sendet. Weiterhin nimmt man an, dass die Nachrichten in der gleichen Reihenfolge beim Empfänger ankommen, in der sie versendet wurden, und dass keine Nachrichten verloren gehen. Führt nun jeder Prozess Buch über die

eingegangenen Nachrichten, so gilt die Bearbeitung derjenigen Ereignisse als sicher, deren Zeitstempel so weit zurückliegt, dass alle anderen Prozesse bereits Nachrichten mit mindestens gleich großem Zeitstempel gesendet haben. Damit kann keine frühere Nachricht mehr empfangen werden.

Sobald aber alle Nachrichten eines Prozesses bearbeitet wurden, muss der bearbeitende Prozess mit weiteren Bearbeitungsschritten warten, bis weitere Nachrichten des erstgenannten Prozesses vorliegen. Blockieren mehrere Prozesse so, dass sie zyklisch aufeinander warten, liegt eine *Verklemmung* vor. Ein konservatives Synchronisationsverfahren muss entweder dafür sorgen, dass keine Verklemmungen auftreten, oder es muss auftretende Verklemmungen erkennen und beseitigen.

Zur Vermeidung von Verklemmungen senden die Prozesse zusätzliche, zeitgestempelte Nachrichten, in denen sie versprechen, keine weiteren Nachrichten mit kleinerem Zeitstempel zu senden. Diese Nachrichten heißen *Null-Nachrichten* im ursprünglichen Algorithmus von Chandry und Misra (1979) oder *Garantien* (Mehl 1994). Sie lösen keine Ereignisse auf Anwendungsebene aus, sondern dienen nur zur Übermittlung eines Zeitstempels, der den individuellen Blick eines Prozesses in die simulierte Zukunft anzeigt. Das Zeitintervall, in dem keine kleineren Zeitstempel gesendet werden, heißt *Lookahead*.

Die Null-Nachrichten können je nach Bedarf gesendet werden, beispielsweise wenn ein Prozess alle eingegangenen Nachrichten eines Vorgängers abgearbeitet hat und somit blockieren müsste. Eine eingehende Null-Nachricht führt zur Aufhebung der Blockade, wenn der Zeitstempel groß genug ist. Andererseits können die beteiligten Prozesse auch selbsttätig Null-Nachrichten versenden, wenn sie einen entsprechenden Berechnungszustand erreicht haben. Ist das Zeitfenster des Lookahead-Wertes klein, werden unter Umständen viele Null-Nachrichten benötigt, um die Blockade für das nächste Ereignis zu lösen. Die Verlangsamung des Simulationsfortschritts durch entsprechend hohes Kommunikationsaufkommen wird als *Time Creeping* bezeichnet.

Damit der Mechanismus der Null-Nachrichten Verklemmungen tatsächlich vermeidet, dürfen keine zyklischen Abhängigkeiten existieren, in denen die beteiligten Prozesse jeweils einen Lookahead-Wert von 0 besitzen. Genauer gesagt muss es in jedem Zyklus von Abhängigkeiten mindestens einen Prozess geben, dessen Lookahead-Wert ständig größer ist als eine beliebige untere Schranke > 0. Damit schließt man die Konvergenz der Zeitstempel gegen einen Wert aus, der kleiner als das nächste Ereignis ist. Gibt es keinen entsprechenden Zyklus, so erreichen die Zeitstempel der Null-Nachrichten nach endlicher Zeit die Simulationszeit des nächsten geplanten Ereignisses, und dieses Ereignis kann sicher berechnet werden.

Durch die untere Schranke der Lookahead-Werte lassen sich einige Simulationen gar nicht durchführen. Abbildung 4.3 zeigt eine Situation, in der es zu einer Verklemmung kommt. Hier hat der Bestell-Prozess eine Bestellung zum Zeitpunkt 10 abgegeben und kann vom Belade-Prozess sofort beliefert werden. Der Bestell-Prozess wird dem nächsten ankommenden Fahrzeug für seine Rückfahrt noch Leergut mitgeben. Ein Lookahead ist nicht über den Zeitpunkt hinaus möglich, zu dem frühestmöglich noch Lieferungen eintreffen könnten. Dies ist gerade der Zeitpunkt 10. Die Nachricht für das Leergut soll nämlich denselben Zeitstempel tragen wie die früheste, noch nicht bearbeitete Lieferung. Der Belade-Prozess kann aber auch keinen positiven Lookahead angeben, da jede Bestellung direkt bedient wird. Hier ergibt sich also ein Zyklus aus Prozessen, die

keine positiven Lookahead-Werte angeben können. Diese Vorhersehbarkeitsforderung kann nach (Mehl 1994) gerade bei zyklischen Warteschlangenmodellen mit exponentiell verteilten Bedienzeiten in der Regel nicht erfüllt werden. Die Verklemmung kann auch nicht durch Maßnahmen des Synchronisationsverfahrens behoben werden, sondern nur durch zusätzliche Annahmen bei der Modellierung, die einen Zeitfortschritt garantieren. Hier ließe sich beispielsweise eine zeitliche Verzögerung bei Leergutrückgabe einführen.

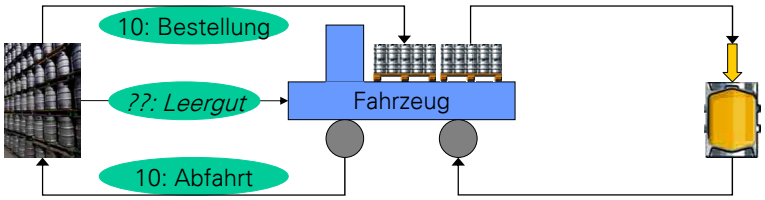


Abbildung 4.3: Verklemmung

Eine andere Möglichkeit, Verklemmungen aus dem Wege zu gehen, stellen (Zeigler u. a. 2000, S. 269) vor. Das Verfahren verlangt eine höhere Granularität bei der Angabe von Lookahead-Werten und steht im Zusammenhang mit der Modellierung nach dem DEVS-Formalismus (Abschnitt 4.3). Ein logischer Prozess empfängt und sendet hierbei Nachrichten für bestimmte Zugangspunkte (Ports, Abschnitt 4.3.2). Jeder Prozess ermittelt nun für jeden Zugangspunkt getrennt die Simulationszeiten, zu denen eine Nachricht ankommen bzw. gesendet werden kann (*Earliest Input Time* bzw. *Earliest Output Time*). Durch diese Trennung lassen sich höhere Lookahead-Werte für einzelne Ausgabeports berechnen, die den gekoppelten Prozessen an deren Eingabeports zur Verfügung gestellt werden. Solange jeder Rückkopplungszyklus im Gesamtmodell mindestens ein zeitverzögerndes Element besitzt, ist die Verklemmungsfreiheit sichergestellt. Diese Forderung ist leichter zu erfüllen als in einem grobgranularen Verfahren, wo der gesamte Prozess die Zeitverzögerung durchführen muss.

Abbildung 4.4 verdeutlicht dieses Verfahren wieder anhand des Abfüllmodells. Das Bestellungsmodell und das Belademodell besitzen nun für die Nachrichtenarten Bestellung, Abfahrt und Leergut jeweils verschiedene Ports. Die erste Nachricht besteht also aus einer Bestellung zum Zeitpunkt 10 auf dem zugehörigen Port. Nachdem das Bestellungsmodell über die unmittelbare Abfahrt des Fahrzeugs benachrichtigt wurde, kann es die Fahrtzeit (hier 10 Zeiteinheiten) ermitteln und dem Belademodell auf dem Leergut-Port die Nullnachricht mit einem Zeitstempel von 20 senden. Daraufhin wird das Belademodell eine erneute Bestellung zum Zeitpunkt 15 direkt beantworten, da die Leergutrücknahme erst später einzuplanen ist.

Die Angabe möglichst großer Lookahead-Werte ist wesentlich für die Performanz eines konservativen Synchronisationsverfahrens, da sie das Time Creeping mildert. Der Einfluss des Lookaheads auf die Performanz kann dramatisch sein. Bei sehr geringen Werten liegt die Geschwindigkeit teilweise deutlich unter derjenigen, die durch sequentielle Abarbeitung der Ereignisse erreicht wird. Da der Lookahead stark vom Modell abhängt, können kleine Änderungen des Modellverhaltens auch deutliche Auswirkungen auf die Ausführungsgeschwindigkeit haben. Um den Blick in die Simulationszukunft so weit wie möglich voranzutreiben, müssen deshalb die Gegebenheiten im Simulationsmodell bestmöglichst

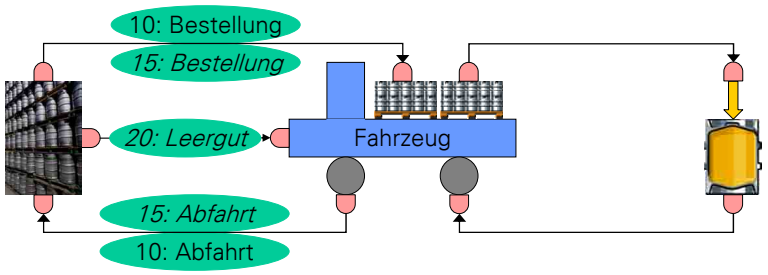


Abbildung 4.4: Erhöhte Granularität durch Ports

ausgenutzt werden. Dabei können physikalische Randbedingungen wie Transportzeiten zwischen den Prozessen oder Reaktionszeiten auf bestimmte Ereignisse einfließen. Gelegentlich können auch Ungenauigkeiten bei der Ermittlung von Ereigniszeitpunkten hingenommen werden, ohne die Simulationsergebnisse zu verfälschen. Wenn die geplanten Ereigniszeitpunkte nicht durch externe Ereignisse verschiebbar sind, so lässt sich auch hieraus ein erhöhter Lookahead-Wert ermitteln.

Um Verklemmungen zu erkennen, setzt man Zähler ein, die die blockierten Prozesse markieren. Zu Beginn eines Simulationslaufs sind alle Prozesse als blockiert markiert. Ein spezieller Steuerungsprozess befreit zunächst alle Prozesse. Nun wird ein Baum aus Abhängigkeiten zwischen den Prozessen erstellt. Sendet ein Prozess A eine Nachricht an den Prozess B, so wird B Nachfolger von A, sofern A nicht bereits im Baum enthalten ist. Jeder einzelne Prozess zählt, wie viele derjenigen Prozesse noch aktiv sind, die von ihm Nachrichten erhalten haben. Sobald ein Prozess blockiert und keiner seiner Nachfolger aktiv ist, signalisiert er seinem Vorgänger den Zustand seiner Inaktivität. Im Falle einer Verklemmung geht diese Meldung bis zum Steuerungsprozess zurück, und der Steuerungsprozess sorgt für die Auflösung der Verklemmung.

Um eine Verklemmung aufzulösen, sind diejenigen Prozesse zu ermitteln, die den kleinsten unbearbeiteten Zeitstempel tragen. Unter der Voraussetzung, dass sich keine Nachrichten mehr auf dem Weg zu einem Empfängerprozess befinden, lässt sich eine Umfrage unter allen Prozessen nach diesem Zeitstempel durchführen. Während dieser Umfrage kann kein Prozess neue Nachrichten erzeugen, da eine Verklemmung vorliegt und alle Prozesse blockiert sind. Um das Kommunikationsaufkommen für die Umfrage gering zu halten, empfiehlt sich ein hierarchisches Vorgehen.

Verklemmungen lassen sich auch durch künstliche Barrierepunkte aufbauen. Ein Barrierepunkt gibt einen Simulationszeitpunkt an, den alle beteiligten Prozesse gemeinsam erreicht haben müssen, bevor sie mit der Bearbeitung von Ereignissen fortfahren dürfen. Durch Barrierepunkte kann ein Simulationslauf in Schritte eingeteilt werden, die aufeinander aufbauen. Während eines Schrittes laufen alle Prozesse unbeeinflusst voneinander ab. Dadurch müssen einzelne Prozesse in der Regel kürzer warten, und die Auslastung der einzelnen Prozessoren wird erhöht. Um sicherzustellen, dass keine Nachrichten mehr auf dem Weg zu einem Empfängerprozess sind, zählen die einzelnen Prozesse sowohl die versendeten als auch die empfangenen Nachrichten. Gleichen sich beide Gesamtsummen über alle Prozesse, so ist jede versendete Nachricht auch empfangen

worden, und keine Nachricht ist mehr unterwegs. Erfolgt die Synchronisation durch Barrierepunkte, können diese Summen-Informationen beim Aufbau von Barrieren gleich mitverschickt werden.

Verfahren mit Barrierepunkten lassen sich auch mit Lookahead-Werten kombinieren. Dabei wird ein globales Minimum der lokalen minimalen nächsten Zeitstempel ermittelt und an alle Prozesse weitergeleitet. Gibt es nur wenige Kommunikationswege zwischen logischen Prozessen, so lohnt es sich, eine vollständige Abstandsmatrix bezüglich des Lookaheads zu erstellen. Somit ergeben sich für die einzelnen logischen Prozesse in der Regel höhere Zeitstempel, deren zugehörige Ereignisse zur Bearbeitung zugelassen sind. Die Berechnung der zulässigen Ereignisse kann allerdings ein hohes Kommunikationsaufkommen erzeugen (quadratisch zur Anzahl der Prozesse).

Sind die Kommunikationswege zwischen den logischen Prozessen nicht im Voraus bekannt, so ist beim Anlegen einer neuen Kommunikationsverbindung sicherzustellen, dass der Empfänger dieser Verbindung vom Sender keine Nachrichten erhält, die in die bereits simulierte Vergangenheit des Empfängers fallen. Solche Situationen könnten sonst zu Kausalitätsfehlern führen. Deshalb wird entweder jeder Empfänger daran gehindert, weiter als den minimalen Lookahead voranzuschreiten, oder der Sender muss vor dem Anlegen einer neuen Verbindung weit genug in Zukunft blicken können.

Da die Simulationsergebnisse in der Regel exakt reproduzierbar sein sollen, ist auch die Reihenfolge von Ereignissen mit gleichem Zeitstempel zu regeln. Ein Zeitstempel wird dabei um zusätzliche Informationen angereichert, die die Reihenfolge auf Grund von Prozessidentifikatoren, Prioritätsregelungen und Empfängerpräferenzen festlegt.

4.1.4 Optimistische Verfahren

Optimistische Synchronisationsverfahren in der Simulation haben Ähnlichkeit mit der optimistischen Bearbeitung von Transaktionen in verteilten Systemen (Coulouris u. a. 1994, S. 423). Der wesentliche Unterschied bei der Simulation entsteht dadurch, dass nicht jede beliebige Reihenfolge für die Bearbeitung der Ereignisse gewählt werden kann, wie es bei Transaktionen erlaubt ist. Es reicht eben nicht aus, irgendeine sequentielle Anordnung der Ereignisse zu fordern, um die Kausalität der Simulationsvorgänge zu gewährleisten. Stattdessen müssen die Zeitstempel der Ereignisse berücksichtigt werden. Weiterhin ähneln optimistische Verfahren in der Simulation entfernt den optimistischen Verfahren zur Abarbeitung von Mikroprozessor-Befehlen. Auch hier können Ereignisse (Prozessorbefehle) betrachtet werden, bevor ein Programmzähler den Befehl erreicht hat. Dabei wird vorher abgeschätzt, ob der Befehl auszuführen ist, oder ob vorher ein Sprungbefehl den Programmzähler verändert.

Das erste und bekannteste Verfahren zur optimistischen Synchronisation in der ereignisdiskreten Simulation ist das Time Warp Verfahren von Jefferson (1985). Es führte die wichtigsten Konzepte aller optimistischen Verfahren ein: Zurücksetzen von Zuständen, Aufheben bereits gesendeter Nachrichten, globale virtuelle Zeit und Bereinigung nicht mehr benötigten Speichers.

Die Restriktionen für das Versenden von Nachrichten sind bei optimistischen nicht so streng wie bei konservativen Verfahren, denn die logischen Prozesse müssen Nachrichten nicht in zeitgestempelter Reihenfolge verschicken, und die Nachrichten brauchen auch nicht in Absendereihenfolge ausgeliefert zu werden.

Dabei kann es passieren, dass ein Prozess schon Ereignisse auf der Basis von Nachrichten mit großem Zeitstempel bearbeitet hat, aber später noch Nachrichten mit kleinerem Zeitstempel eintreffen. Die späten Nachrichten könnten nun Einfluss auf die schon verarbeiteten Ereignisse haben, die der Prozess nicht berücksichtigen konnte. Die Wirkungen der verarbeiteten Ereignisse müssen nun rückgängig gemacht werden, damit keine Kausalitätsfehler entstehen. Dazu speichert jeder lokale Prozess nach dem Senden einer Nachricht die Inhalte seiner Zustandsvariablen. Die Speicherung kann entweder inkrementell oder vollständig vorgenommen werden. Auch die versendeten Nachrichten sind aufzubewahren.

Fällt eine empfangene Nachricht in die Vergangenheit eines logischen Prozesses, so setzt dieser Prozess seine lokale Simulationszeit und die Zustandsvariablen auf den letzten, vor dieser Zeit gespeicherten Zustand zurück und führt seine Berechnungen unter Einbeziehung der späten Nachricht fort. Weiterhin sorgt dieser Prozess dafür, dass die Nachrichten, die nach der späten Nachricht bereits gesendet wurden, bei den entsprechenden Empfängerprozessen ungültig werden. Dazu sendet er jede entsprechende Nachricht noch einmal, jedoch mit einer Markierung für deren Ungültigkeit (anti-message). Empfängt ein logischer Prozess eine als ungültig markierte Nachricht, so muss auch dieser Prozess seinen Zustand zurücksetzen und ggf. weitere Nachrichten per Markierung zurückrufen. Damit keine Zyklen beim Zurückziehen auftreten, werden die Zeitstempel in markierten Nachrichten so erweitert, dass sich auch bei gleichen Zeitstempeln immer noch eine eindeutige Reihenfolge der Nachrichten ermitteln lässt.

Damit die logischen Prozesse ihre lokal gespeicherten Zustände nicht bis zum Ende der Simulation aufbewahren müssen, wird auf Anforderung einzelner Prozesse oder in regelmäßigen Abständen eine globale virtuelle Simulationszeit ermittelt. Dieser Zeitpunkt ist jedoch nur virtuell, weil die lokale Simulationszeit in den Prozessen unterschiedlich weit vorangeschritten sein kann. Er ist global, wenn alle Prozesse diesen Zeitpunkt bereits erreicht haben. Damit braucht kein Prozess Nachrichten vor diesem Zeitpunkt mehr zurückzuziehen. Der Speicherplatz für die Zustände vor diesem Zeitpunkt kann nun freigegeben werden. Eine globale virtuelle Simulationszeit berechnet sich aus dem minimalen Zeitstempel aller unbearbeiteten Nachrichten und aller ungültig markierten Nachrichten. Dabei dürfen sich keine Nachrichten mehr auf dem Weg von einem Sender zu einem Empfänger befinden. Dies kann durch Barrieren ähnlich wie im konservativen Fall passieren. Auch mehrschrittige Algorithmen finden gelegentlich Anwendung, die eine Blockade der einzelnen Prozesse verkürzen sollen.

4.2 HLA

Mitte der 90er Jahre ließ das amerikanische Verteidigungsministerium (DoD) eine technische Basis für Simulationen entwickeln, die nach (Dahmann u. a. 1997) verschiedene vorherige Ansätze zusammenführen sollte (*Aggregate Level Simulation Protocol*; ALSP und *Distributed Interactive Simulation*; DIS). Dafür nennt (Straßburger 2001, S. 3) triftige finanzielle Gründe. Die vorhandenen militärischen Simulatoren mussten in immer umfangreicheren Szenarien eingesetzt werden, aber eine Integration war wegen der fehlenden Interoperabilität mit viel Aufwand verbunden und dementsprechend teuer. Als Ausweg wurde die umfassende „High Level Architecture“ (HLA) entworfen, deren Einsatz für alle neu

entwickelten Simulationen im Bereich des DoD verpflichtend war. Unterstützende Werkzeuge waren bis Oktober 2002 über die Simulationsabteilung DMSO frei erhältlich. Darunter befanden sich eine vollständige Laufzeitumgebung und verschiedene Editoren zur Verwaltung und Dokumentation von Simulationskomponenten. Durch den freien Zugang zu dieser Software beschäftigten sich auch immer mehr zivile Institute mit der Anwendung von HLA. Anwendungen im zivilen Bereich diskutiert (Straßburger 2001) ausführlich. Inzwischen hat HLA auch bei den Standardisierungsorganisationen den Begutachtungsprozess erfolgreich durchlaufen (IEEE 2000a, b, c) und (OMG 2002b). In (Bachmann 2000) findet sich eine detaillierte Beschreibung der Beziehungen zwischen HLA und der OMG-Standardisierung.

HLA wird gern als „Stand der Technik“ auf dem Gebiet der verteilten, komponentenorientierten Simulation beschrieben (Straßburger 2001, Kap. 2). Allein die regelmäßige Anzahl an HLA-Beiträgen zu einer der weltweit größten Simulationskonferenzen (WinterSim: Andradóttir u. a. 1997; Medeirors u. a. 1998; Farrington u. a. 1999; Joines u. a. 2000; Peters u. a. 2001; Yücesan u. a. 2002) belegt das. Deshalb ist in den nachfolgenden Abschnitten die HLA zu Grunde liegende Technik ausführlich beschrieben. Dabei werden sich die Probleme mit dem herstellerunabhängigen Einsatz verdeutlichen.

4.2.1 Überblick

HLA ist eine Basistechnik zur Kommunikation zwischen Komponenten in verteilten Simulationsumgebungen. Diese Komponenten stammen typischerweise aus verschiedenen Simulationsumgebungen. Aus dem militärischen Umfeld stammt die Maßgabe, auch die Einbindung von menschlichen Interaktionen (man-in-the-loop) innerhalb komplexer Übungsszenarien zu ermöglichen. Der Fokus liegt bei HLA auf einer verteilten Semantik und nicht auf einer hochperformanten, parallelen Abarbeitung komplexer Simulationsprobleme. Die HLA-Spezifikation besteht aus drei Hauptteilen:

1. (IEEE 2000a) Regeln zur Kommunikation zwischen Komponenten
2. (IEEE 2000b) Schnittstellen zu verfügbaren Laufzeitdiensten
3. (IEEE 2000c) Notation für Beschreibungen der Komponenten

Die HLA-Regeln besagen im Wesentlichen, dass die gesamte Kommunikation über die Schnittstellen einer Laufzeitumgebung (Runtime Infrastructure, RTI) geführt werden muss, dass alle Komponenten syntaktisch beschrieben sind und dabei die vorgeschriebene Notation verwenden. Sie sollen hier nicht weiter vertieft werden, ebenso wenig wie die konkrete Syntax zur Komponentenbeschreibung. Wichtig für einen Vergleich verschiedener Lösungsansätze sind nur die verschiedenen Dienste, die mit HLA zur Verfügung stehen.

Wesentliche Begriffe der verteilten, komponentenorientierten Simulation mit HLA sind Federate und Federation. Ein *Federate* ist eine Simulationskomponente, die anderen Komponenten während eines Simulationslaufs Daten zur Verfügung stellt. Ein Simulationslauf unter HLA besteht aus dem Aufbau und späterem Abbau einer *Federation*. Während der Durchführung des Simulationslaufs melden sich einzelne Federates bei dieser Federation an und werden Teil der Federation. Auch die Abmeldung von Federates zur Laufzeit ist vorgesehen. Die hierarchische Modellierung komplexer Systeme mit HLA wird hingegen nicht

direkt unterstützt (Davis und Moeller 1999). Erweiterungen, die spezielle Federates zum Datenaustausch über die Grenzen von Federations hinweg vorsehen, diskutieren beispielsweise Ulrikson u. a. (2001) sowie Schulze u. a. (2002).

Federates erzeugen *Objekte*, die verschiedenen *Objektklassen* entstammen können. Eine Objektklasse beschreibt die Struktur der zugehörigen Objekte, d.h. eine Menge von *Attributen*, die zu einem Objekt gehören. Die HLA-Objektklassen enthalten keine Methodensignaturen, deshalb zeigen die Objekte kein eigenständiges Verhalten, sondern dienen nur als Attributbehälter. Die Attribute sind Zugangspunkte für aktuelle Daten innerhalb einer Federation. Um also Daten innerhalb einer Federation zu übertragen, muss das Attribut und das zugehörige Objekt angegeben werden, dem ein Datenwert entspringt. Weiterhin können Federations *Interaktionsklassen* enthalten. Eine Interaktionsklasse besitzt höchstens eine Ausprägung innerhalb der gesamten Federation. Die Interaktionen repräsentieren jedoch keine Funktionsaufrufe im Sinne von Remote Procedure Calls, denn die entsprechenden Schnittstellen adressieren keine Federates und bieten auch keine Rückgabewerte an. Sie sind lediglich ein weiteres Mittel, um die verfügbaren Daten einer Federation zu strukturieren. Die Untereinheiten bei Interaktionen nennen sich *Parameter*, im Vergleich zu den Attributen von Objekten. Sie können als zentrale Behälter angesehen werden, die nicht in Zusammenhang mit Objekten stehen. Alle Attribute und Interaktionen zusammen repräsentieren also die von Federates erkennbaren Modellgrößen (vgl. Abschnitt 2.1.2). In Abbildung 4.5 ist der prinzipielle Aufbau einer Federation mit verschiedensten Zugangspunkten dargestellt. Die Ellipsen und Kreise stehen für die Struktureinheiten Objekt bzw. Interaktion. Die Zugangspunkte Attribut bzw. Parameter erkennt man als kleine Quadrate. Objekte und Interaktionen definieren jeweils eigene Sichtbarkeitsbereiche, so dass Attribute und Parameter mit gleichem Namen in verschiedenen Objekten bzw. Interaktionen vorkommen dürfen. Zur Laufzeit werden zunächst einmal die Objekte anderen Federates einzeln bekannt gemacht. Anschließend lassen sich Veränderungen an den Attribut- oder Parameterwerten an andere interessierte Federates melden.

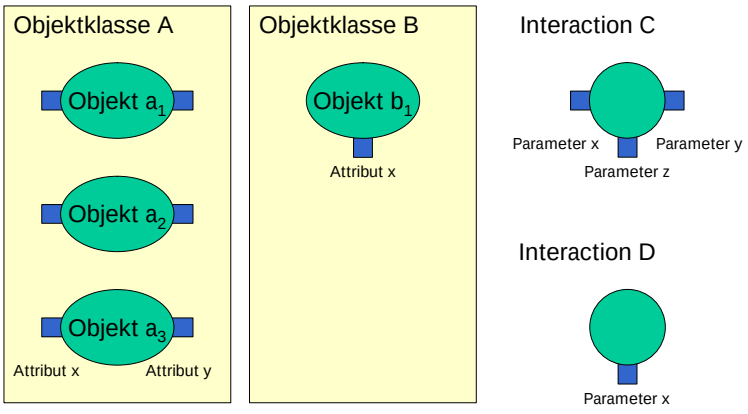


Abbildung 4.5: HLA Zugangspunkte

Eine HLA-Realisierung des Modellbeispiels aus Kapitel 3 könnte etwa eine Objektklasse Fahrzeug mit Attributen für die tatsächliche Beladung und eine

Objektklasse Fass mit Attributen für die Fassgröße und den Füllstand deklarieren. Es wird dann je ein Federate für den Transport- und für den Beladeteil existieren sowie je Füllstation ein weiteres Federate. Das Transport-Federate und das Belade-Federate tauschen dann Veränderungen über den Beladezustand der Fahrzeuge aus, wobei eine feste Anzahl an Fahrzeug-Objekten vom Start der Federation an zu erzeugen ist. Das Belade-Federate kann je Befüllungsvorgang ein neues Fass-Objekt erzeugen und mit dem beauftragten Füll-Federate Veränderungen am Füllstand des Fasses austauschen. Im Gesamtmodell ließe sich der Simulationsfortschritt über eine Interaktion steuern, die die Simulationsuhr repräsentiert und mögliche Benutzereinstellungen an diese Uhr weiterleitet.

Die Dokumentation von Objekt- und Interaktionsklassen innerhalb eines Federates erfolgt durch ein *Simulation Object Model* (SOM). Um eine Federation aufzubauen, ist eine zusammenfassende Beschreibung aller Deklarationen der Federates anzugeben (*Federation Object Model*, FOM). In einem Object Model sind die Namen aller Klassen, Attribute und Parameter abgelegt. Es lassen sich auch neue, zusammengesetzte Datentypen deklarieren. Dabei sind Aufzählungstypen und Record-Strukturen zugelassen. Den Attributen und Parametern sind entweder Basisdatentypen (wie String, Integer, etc.) oder durch ein Object Model neu deklarierte Typen zugewiesen. Die Objekt- und Interaktionsklassen sind hierarchisch aufgebaut, d.h. die Menge zugehöriger Attribute bzw. Parameter wird an untergeordnete Klassen vererbt. Der syntaktische Aufbau eines Object Models wird durch das *Object Model Template* (OMT) vorgeschrieben. Die grafisch- und tabellenorientierte Notation spiegelt den hierarchischen Klassenaufbau wider und lässt sich als textbasierte Variante im XML-Format angeben. Für den Bereich Geodaten beispielsweise versuchen (Schulze u. a. 2002) eine einheitliche Beschreibung auf der Basis von Spezifikationen des Open GIS Consortiums als FOM durchzusetzen. Als wesentliches Defizit der OMT-Beschreibung sieht (Tolk 2000) die fehlende Möglichkeit, Beziehungen wie Aggregation, Assoziation oder Mehrfachvererbung zwischen Objektklassen anzugeben.

Zur Laufzeit stehen den einzelnen Federates verschiedene Dienste zur Verfügung. Diese Dienste werden in den nachfolgenden Abschnitten genauer beschrieben. Dazu gehört u.a. die Verwaltung von

- Federations
- Kommunikationsbeziehungen zwischen Federates
- Datenübertragungen
- Besitzrechten an Attributen
- Synchronisationsschritten für die Simulationszeit
- Einschränkungen bei der Datenübertragung.

Die Kommunikation der Federates mit der Federation und die Kommunikation der Federates untereinander erfolgt ausschließlich durch Funktionsaufrufe und -rückrufe über eine zentrale Komponente (Runtime Infrastructure, RTI). Die RTI hat in den meisten HLA-Realisierungen einen lokalen Repräsentanten innerhalb jedes Federates. Für die Schnittstellen von Federates zur RTI und umgekehrt sind Umsetzungen für die Programmiersprachen Ada, Java und C++ festgelegt. Da HLA auch durch die OMG standardisiert ist, existiert zusätzlich eine Schnittstellendefinition für CORBA-basierte Anwendungen in IDL-Syntax. Damit lassen sich im Prinzip auch Komponenten anbinden, die in anderen Programmiersprachen wie C, COBOL oder Lisp realisiert sind. Das interne Verhalten einer RTI ist jedoch in HLA unspezifiziert. Dadurch sind in der Regel

HLA-Objekte in der Laufzeitumgebung eines bestimmten Herstellers nicht in der Laufzeitumgebung eines anderen Herstellers nutzbar. Diese Situation entspricht in etwa dem Stand der ersten CORBA-Spezifikation. Auch dort war Interoperabilität zwischen unterschiedlichen Object Request Broker Realisierungen nicht möglich. Das Problem wurde aber schnell erkannt und umgehend in nachfolgenden Entwürfen sowie Implementationen behoben, so dass heute der Einsatz unterschiedlicher CORBA-Produkte innerhalb einer Anwendung kein Hindernis mehr ist. Anders sieht es jedoch für HLA aus. Hier gibt es noch grundsätzliche Probleme im Bereich der Synchronisation und des Datenaustausch, die einem herstellerunabhängigen Einsatz entgegenstehen (Abschnitte 4.2.7 und 4.2.6). Abbildung 4.6 fasst noch einmal den Aufbau der HLA-Infrastruktur zusammen. Die kleinen Ellipsen innerhalb eines Federates repräsentieren dabei den herstellerabhängigen Anteil.

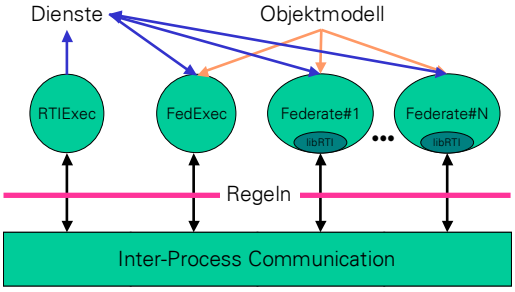


Abbildung 4.6: HLA Infrastruktur, nach (DMSO 2000, S. 2-2)

4.2.2 Federation Management

Die Aktivierung von Simulationen über Federations ist eine Eigenheit, die den militärischen Ursprung der HLA-Spezifikation erkennen lässt. Typischerweise dienen Simulationen mit HLA zur Durchführung komplexer Trainingsszenarien, die über längere Zeiträume andauern und von Mensch-Maschine-Interaktionen geprägt sind. Deshalb wird eine Federation ohne modellspezifische Initialisierungen gestartet. Die einzelnen Teilnehmer können nach und nach in die Simulation einsteigen und sie wieder verlassen. Will man jedoch andersartige Simulationen mit HLA durchführen, die durch viele kurze Simulationsläufe ohne menschliche Interaktionen charakterisiert sind, so fehlt ein Benachrichtigungsmechanismus für Start- und Ende-Ereignisse dieser Simulationsläufe. Hier sollten die Simulationskomponenten üblicherweise direkt nach dem Start eines Simulationslaufs in einen definierten Ausgangszustand gelangen und ihre Arbeit aufnehmen.

Federates existieren erst dann, wenn sie einer Federation beigetreten sind. Es gibt keinen Mechanismus, mit dem man nach Bedarf Federates mit einem definierten Verhalten erzeugen könnte. Auch die erzeugten Federations sind nirgends verzeichnet, so dass Federates für eine aufzubauende Federation nur über benutzerspezifische Regelungen zugreifbar sind. Oses (2002, Abschn. 8.2.3) und (Bachmann 2001) fordern daher zu Recht einen Verzeichnisdienst für die komponentenorientierte Simulation.

Sobald aber eine Federation erzeugt ist, stehen Mechanismen zum Speichern und Wiederherstellen des Gesamtzustandes sowie zur globalen Synchronisation

der Federates zur Verfügung. Die notwendigen modellspezifischen Aktionen zur Speicherung und zur Synchronisation bleiben vollständig den Federates überlassen. HLA stellt lediglich den geordneten Aufruf entsprechender Methoden bereit, der dem 2-Phasen-Commit-Protokoll für Datenbank-Transaktionen (Lamersdorf 1994, Abschnitt 5.3.3) ähnelt. Abbildung 4.7 zeigt die Aufrufe zwischen einem einzelnen Federate und der Federation für die Durchführung eines Speichervorgangs. Dasselbe Schema wird auch zur Wiederherstellung einer Federation oder zur globalen Synchronisation verwendet.

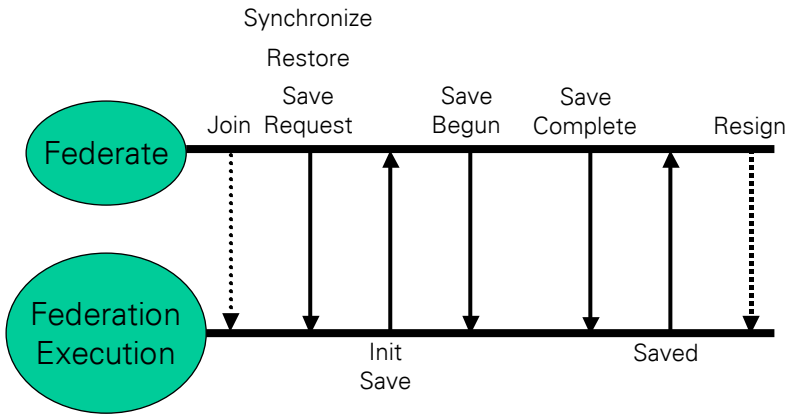


Abbildung 4.7: HLA Federation Aufrufe

Die Schnittstellenbeschreibung zur Erzeugung von Federations (IEEE 2000b, Abschnitt 4.2) sieht explizit die Angabe eines FOMs vor. Damit ist die Laufzeitumgebung in der Lage, im Rahmen des Declaration Managements (Abschnitt 4.2.3) einen Mechanismus zur Nachrichtenweiterleitung zu realisieren, der die Klassenstruktur einer Federation berücksichtigt. Einer RTI-Implementation wird jedoch nicht vorgeschrieben, wie der zugehörige textuelle Parameter `FOM Document Designator` der Schnittstellenfunktion `Create Federation Execution` zu interpretieren ist. Die bekannten Implementationen wie beispielsweise (DMSO 2000) laden eine Datei mit entsprechendem Namen. So ist HLA ohne weitere Vorkehrungen zunächst nur in Netzen mit einem gemeinsamen Dateisystem einfach nutzbar. Sinnvoll wäre jedoch eine Angabe im URL-Schema (wie beispielsweise `ftp://` oder `http://`), um das entfernte Arbeiten mit einer RTI-Umgebung zu vereinfachen. Es ist über die Schnittstellen der Laufzeitumgebung möglich, die Inhalte der FOMs einzusehen, um zur Laufzeit an die Klassenbeschreibungen zu gelangen. Erst dadurch lassen sich mächtige Werkzeuge wie beispielsweise eine allgemeine Experimentierumgebung (Abschnitt 7.4) entwickeln, die auf unterschiedlichen Datentypen operieren können und einen Introspektionsmechanismus benötigen. Für solche Introspektionsaufgaben steht das Management Object Model (Abschnitt 4.2.8) zur Verfügung.

4.2.3 Declaration Management

Damit die einzelnen Federates tatsächlich Daten untereinander austauschen können, ist ein Publish/Subscribe-Mechanismus vorgesehen. Die Publish- und Subscribe-Nachrichten dienen nur zur Vorbereitung des Datenaustauschs und

liefern keine Nutzdaten wie etwa im Beobachter-Muster aus (Gamma u. a. 1996, S. 257ff.). Ein Federate, das bzgl. einer Objekt- oder Interaktionsklasse Daten zur Verfügung stellt, muss sich im Voraus bei der Laufzeitumgebung als entsprechender Datensender mit einem Publish-Aufruf anmelden. Erst danach darf das Federate Objekte erzeugen oder wieder vernichten und Attribute bzw. Parameter verändern. Jeder Werteveränderung kann ein Zeitstempel zugeordnet werden, der in Verbindung mit der Zeitverwaltung (Abschnitt 4.2.6) die Reihenfolge bestimmt, in der die Federates die Veränderung mitgeteilt bekommen.

Um die Veränderungen an den zugehörigen Daten einer Objekt- oder Interaktionsklasse vorzubereiten, führen die interessierten Federates eine Subscribe-Meldung durch. Attribute bzw. Parameter werden nicht direkt von einem Federate durch einen synchronen Methodenaufruf abgefragt. Federates erfahren die Veränderung der Modellgrößen ausschließlich über Rückrufmethoden, die von der Laufzeitumgebung initiiert sind. Eine Laufzeitumgebung muss somit einen Multicast-Mechanismus realisieren, der die Änderungsmitteilungen weiterleitet und dabei die vorliegenden Subscribe-Meldungen berücksichtigt.

Ein Federate kann eine vorherige Publish- oder Subscribe-Meldung jederzeit widerrufen. Die Kommunikationsbeziehungen zwischen einzelnen Federates sind also dynamisch, sie können sich während der Laufzeit einer Federation ändern. Dynamische Beziehungen erlauben jedoch nicht die Verwendung von Synchronisationsverfahren, die Informationen über die möglichen Abstände zeitgestempelter Ereignisse ausnutzen könnten. Viele grafische Werkzeuge zur Kopplung von Komponenten wie CoCo (Abschnitt 7.3) oder WBSS (Oses 2002) nutzen die Möglichkeiten dynamischer Beziehungen gar nicht aus. Sie spezifizieren lediglich feste Kommunikationswege zwischen beteiligten Komponenten.

4.2.4 Object Management

Neue Objekte lassen sich durch Aufruf der Funktion `Register Object Instance` bekannt geben und durch Empfang des Rückrufs `Discover Object Instance` erkennen. Die entsprechende Löschfunktion heißt `Delete Object Instance` bzw. `Remove Object Instance` (Rückruf). Die tatsächliche Meldung der Federates über Veränderungen von Attributen bzw. Parametern findet statt durch Aufrufe der Funktionen `Update Attribute Values` bzw. `Send Interaction` und durch Empfang der Rückrufe `Reflect Attribute Values` bzw. `Receive Interaction`. Während eines Datenaustauschs können mehrere Attribute oder Parameter gemeinsam übertragen werden. Sie lassen sich durch `AttributeHandles` bzw. `ParameterHandles` unterscheiden, die in der Laufzeitumgebung künstlich erzeugt wurden. Handles sind innerhalb einer Federation eindeutig, und die Laufzeitumgebung soll immer die gleichen Handles erzeugen, wenn später die Federation mit gleichem Namen wieder aufgebaut wird. Durch die gemeinsame Übertragung mehrerer Parameter innerhalb einer Interaktionsnachricht ist der Ursprung der Bezeichnung Interaktion vielleicht am besten erkennbar. Interaktionen sollen nach (Straßburger 2001, S. 8) die Beziehungen zwischen Objekten ausdrücken. Dies wird bei Interaktionen möglich, wenn als Parameterdatentyp Verweise auf die beteiligten Objekte angegeben sind und die zusammenhängenden Parameter auch innerhalb eines Interaktionsaufrufs gemeinsam übertragen werden.

Als Datenübertragungseinheit stellt HLA lediglich Byte-Sequenzen zur Verfügung. Die Codierung und Decodierung der einzelnen Datenwerte bleibt voll-

ständig den einzelnen Federates überlassen. Mit der Notation für Klassenbeschreibungen (OMT) in (IEEE 2000c) ist zwar ein Codierungsschema vorgeschlagen worden, dieses Schema ist für Federates jedoch nicht bindend. Bedenkt man, dass HLA bereits seit Veröffentlichung der ersten frei verfügbaren Laufzeitumgebung (Dahmann u. a. 1997) weite Verbreitung gefunden hat, so ist die Interoperabilität zumindest zwischen frühen HLA-Federates nicht automatisch gegeben. Um die Funktionalität eines anderen Federates wirklich nutzen zu können, müssen zunächst außerhalb der HLA-Umgebung Vereinbarungen über die Codierungsrichtlinien getroffen werden. Solange ein Codierungsschema wie OMT nicht bindend ist, bestehen diese Probleme weiterhin. Mit der Einigung auf ein solches Schema wäre es dann auch sinnvoll, Schnittstellen zur Umwandlung zwischen codierten Daten und Datenstrukturen der jeweils verwendeten Programmiersprache zur Verfügung zu stellen. Diese Schnittstellen fehlen im aktuellen IEEE-Standard. Sie können einem Anwendungsentwickler die Arbeit jedoch außerordentlich vereinfachen, wie für CORBA in (Vogel und Duddy 1998, Kap. 2) beschrieben ist.

4.2.5 Ownership Management

Innerhalb einer Federation ist zu jedem Zeitpunkt nur ein Federate berechtigt, die Werteänderungen eines bestimmten Attributs zu melden. Je Attribut kann natürlich ein anderes Federate das Melderecht innehaben, und während der Laufzeit einer Federation lässt sich dieses Recht an andere Federates übertragen. Fasst man ein Federate, das das Melderecht für ein Objektattribut besitzt, als Aufenthaltsort eines Attributs auf, kann man die Attribute innerhalb einer Federation als mobil bezeichnen. Die Interaktionsparameter sind jedoch fest und haben keine Eigentümer, d.h. jedes Federate, das Interaktionen durch das Declaration Management bekannt gegeben hat, darf auch Parameteränderungen versenden.

Die Verwaltung der Rechte erfolgt über die Schnittstellen des Ownership Managements. Abbildung 4.8 zeigt die wesentlichen Anfrageschritte zur Übergabe von Melderechten. Der Simulationszeitpunkt, zu dem eine Übergabe stattfinden soll, lässt sich jedoch nicht per Programmierschnittstelle festlegen. Die Laufzeitumgebung gibt auch keine Garantien, innerhalb welcher Zeiträume eine Übergabe spätestens durchgeführt sein wird. Dadurch kann es passieren, dass die Durchführung der Übertragung in verschiedenen Simulationsläufen zu unterschiedlichen Zeitpunkten stattfindet und eine Wiederholbarkeit der Läufe nicht gegeben ist. Im schlimmsten Fall kann sogar ein Kausalitätsfehler auftreten, wenn die weiteren Simulationsereignisse vom Eigentümer eines Attributs abhängen. Um die Übergabezeitpunkte dennoch zu fixieren, müssten die beteiligten Federates auf modellspezifische programmierte Synchronisationsschritte zurückgreifen, und dazu die Synchronisationspunkte im Federation Management nutzen.

4.2.6 Time Management

Um die Synchronisation der Federates untereinander in Verbindung mit zeitgestempelten Nachrichten durchzuführen, bietet das Time Management eine große Zahl an Schnittstellen an. Hier soll nur auf die prinzipiellen Möglichkeiten der Zeitverwaltung mit HLA eingegangen werden. In diesem Zusammen-

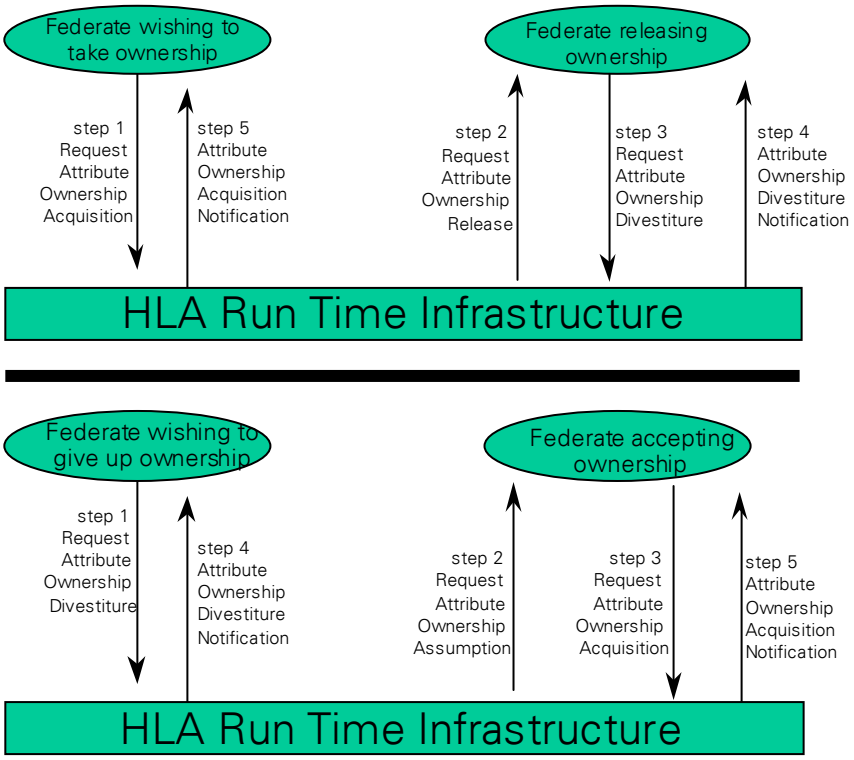


Abbildung 4.8: HLA Ownership, vgl. (DMSO 2000, Fig. 9-3, 9-4)

hang führt HLA zwei wesentliche Merkmale von Federates ein (zeitsteuernd und zeitbeschränkt). Nur zeitsteuernde (*time-regulating*) Federates dürfen überhaupt zeitgestempelte Nachrichten verschicken. Sie müssen über entsprechende Schnittstellen einen aktuellen Lookahead-Wert angeben und versprechen damit, keine Nachrichten mit Zeitstempeln zu versehen, die vor ihrer aktuellen Simulationszeit zuzüglich dieses Lookaheads liegen. Der Empfang zeitgestempelter Nachrichten ist nur den zeitbeschränkten (*time-constrained*) Federates möglich. Federates können beliebig wählen, welche der beiden Merkmale zeitsteuernd und zeitbeschränkt ihnen aktuell zugeschrieben sind. Es sind alle vier Kombinationen erlaubt (Tabelle 4.1). Die Merkmale können sich auch zur Laufzeit beliebig verändern.

zeitbeschränkt	zeitsteuernd	
	+	−
+	?	?
−	?	0

Tabelle 4.1: HLA Zeitkombinationen

Sobald jedoch für ein Federate eines der Merkmale eingestellt ist, muss dieses Federate den Zeitfortschritt bei der Laufzeitumgebung beantragen. Die Fragezeichen in Tabelle 4.1 markieren die entsprechenden Fälle. Der mit „0“ markierte Fall deutet an, dass ein Federate völlig unabhängig von der Zeitverwaltung

agiert, beispielsweise bei reinen Protokollierungsfunktionen. Die Laufzeitumgebung gewährt den Zeitfortschritt nur insoweit, als dass kein zeitbeschränktes Federate Nachrichten mit einem früheren Zeitstempel als die bereits gewährte Zeit erhalten kann. HLA sieht dabei keine zentrale Simulationszeit vor, d.h. die von der Laufzeitumgebung genehmigte Simulationszeit kann je Federate unterschiedlich ausfallen. Zeitbeschränkte Federates, die eine Auslieferung von Nachrichten in strikter Kausalitätsreihenfolge wünschen, erhalten diese also auch in einer solchen Reihenfolge.

Fujimoto und Weatherly (1996) diskutieren die Algorithmen, die für HLA einsetzbar sind. Dort heißt es in Abschnitt 3:

> Time management transparency is important to achieve interoperability. This means the time management mechanism used within each federate is not visible to other federates.

Die Federates sollen also alle zeitgestempelten Simulationsereignisse ohne Berücksichtigung des in der Laufzeitumgebung verwendeten Synchronisationsverfahrens berechnen, um interoperabel zu sein. Mit Interoperabilität ist dort die Fähigkeit aller Federates gemeint, auch in der Laufzeitumgebung eines anderen Herstellers ablaufen zu können. Diese Interoperabilität ist gewährleistet, wenn die Federates nur die vorgesehenen Schnittstellen verwenden und sich an deren Semantik nach (IEEE 2000b) halten. Die HLA-Spezifikation gibt jedoch nicht an, welche Synchronisationsverfahren tatsächlich verwendet werden dürfen. Natürlich wird jeder RTI-Hersteller für sich einen bestimmten Synchronisationsmechanismus implementieren, der in der Regel sowohl korrekt als auch performant ist. Damit ist aber noch längst nicht gewährleistet, dass alle Hersteller den gleichen Algorithmus und die gleichen Nachrichtenformate benutzen. Deshalb besteht bei der Kopplung von Federates, die in Laufzeitumgebungen unterschiedlicher Hersteller leben, aber gemeinsam eine Federation bilden sollen, immer die Gefahr von Inkompatibilität bzgl. des Synchronisationsverfahrens. Solange also keine Aussagen über die möglichen Algorithmen und Formate *zwischen* den Laufzeitumgebungen gemacht werden, bleibt Herstellerunabhängigkeit für HLA unerreichbar.

4.2.7 Data Distribution Management

Für den Datenaustausch zwischen Federates realisiert die Laufzeitumgebung einen Multicast-Mechanismus für Werteänderungen (vgl. Abschnitt 4.2.3). Gibt es aber nun innerhalb einer Federation sehr viele Objekte einer Objektklasse oder erfolgen sehr viele `Send Interaction` Aufrufe einer Interaktionsklasse, so kann die Zahl der übertragenen Einzelnachrichten sehr groß werden. Diese Zahl lässt sich in vielen Fällen nicht wesentlich einschränken, wenn ausschließlich der Publish- und Subscribe-Mechanismus auf der Basis von Objekt- oder Interaktionsklassen benutzt wird. Ohne weitere Möglichkeiten zur Einschränkung würden die meisten Nachrichten, die bei einem Federate eintreffen, häufig gar nicht von Belang sein. Deshalb lassen sich über das Data Distribution Management jeder Nachricht zusätzliche beschreibende Daten mit auf den Weg geben, die der Laufzeitumgebung eine gezieltere Auslieferung der Nachrichten an interessierte Federates ermöglicht. Nicht die Inhalte der eigentlichen Nachrichten werden dabei durch die Laufzeitumgebung interpretiert, sondern nur die beschreibenden Daten. Einen Filtermechanismus auf der Basis von mobilem Code, der auch den

Inhalt von Nachrichten berücksichtigt, stellen (Murphy und Aswegan 1998) vor. Dieser Mechanismus führt jedoch wieder zur Sprachabhängigkeit der Laufzeitumgebung.

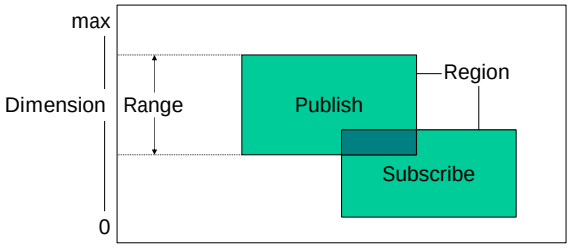


Abbildung 4.9: HLA Data Distribution

Das ursprüngliche Filterkonzept der HLA lässt hingegen die räumliche Einordnung von Nachrichten zu. In der Beschreibung einer Federation (FOM) kann man mehrdimensionale Räume definieren. Eine *Dimension* ist im FOM namentlich gekennzeichnet und steht für ein ganzzahliges Intervall, dessen Untergrenze durch 0 festgelegt ist. Die Obergrenze wird im FOM angegeben. Ein Ausschnitt aus einem definierten Raum heißt *Region* und spezifiziert für jede Dimension ein enthaltenes Intervall (*Range*). Jeder Objekt- und Interaktionsklasse kann nun ein definierter Raum zugeordnet werden. Anschließend lassen sich Nachrichten aus der entsprechenden Klasse durch Regionen charakterisieren. Wird auch die Anmeldung zum Empfang der Nachrichten dieser Klasse mit einer Region versehen, so führt die Laufzeitumgebung zur Bestimmung der tatsächlichen Empfänger aller räumlich eingeordneten Nachrichten der Klasse eine mehrdimensionale Schnittmengenbildung durch. Ein Beispiel für den ebenen Fall mit zwei Dimensionen findet man in Abbildung 4.9. Ein Federate veröffentlicht Nachrichten der Region `Publish` und ein anderes interessiert sich für Nachrichten der Region `Subscribe`. Da die Schnittmenge der Regionen nicht leer ist, werden die Nachrichten zwischen diesen Federates übertragen.

4.2.8 Management Object Model

Um Einblick in die Vorgänge innerhalb der Laufzeitumgebung zu erhalten, lassen sich über die Multicast-Mechanismen des Object Managements auch reflektive Aufgaben wahrnehmen. Das Management Object Model deklariert eine große Zahl an Objekt- und Interaktionsklassen, auf die eine Laufzeitumgebung in festgelegter Weise reagieren muss, d.h. hier interpretiert die Laufzeitumgebung die *Inhalte* der versendeten Nachrichten und verschickt selbsttätig Nachrichten. Diese Klassenbeschreibungen sind zwingender Teil jeder modellspezifischen FOM-Deklaration. Die Anfrage von internen Eigenschaften in der Laufzeitumgebung erfolgt ausschließlich durch asynchrone Kommunikation. Um beispielsweise die Anzahl der versendeten Interaktionen eines anderen Federates zu erfragen, sendet ein Federate zunächst die Interaktion `HLArequestInteractionsSend`. Die Laufzeitumgebung empfängt diese Nachricht und reagiert darauf mit dem Versand der Interaktion `HLAreportInteractionsSend`.

Über das Management Object Model können auch die Objektmodelle der einzelnen Federates bzw. der gesamten Federation ermittelt werden. Hier gilt

jedoch auch die in Abschnitt 4.2.2 genannte Einschränkung, dass lediglich herstellerabhängige Verweise auf den Ablageort der SOM- bzw. FOM-Dateien geliefert werden. Konnte eine Datei erfolgreich eingelesen werden, so muss noch ein lokaler Parsing-Vorgang erfolgen, um die relevanten Daten aus diesem XML-Dokument zu extrahieren.

4.2.9 Ziviler Einsatz

Die Laufzeitumgebung jeder HLA-Realisierung stellt wesentliche Dienste zur Kommunikation und Koordination zwischen den beteiligten Simulationskomponenten bereit. In der aktuellen Form ist jedoch in den Bereichen Synchronisation, Introspektion und Aufbau von Simulationsverbünden eine deutliche Abhängigkeit vom jeweiligen Hersteller der Laufzeitumgebung zu erkennen. Mit der Kommerzialisierung einer ehemals frei verfügbaren Laufzeitumgebung liegt die Vermutung nahe, dass die RTI-Hersteller gar kein Interesse an übergreifender Interoperabilität haben. Könnte man nämlich die Produkte eines Herstellers problemlos durch die eines anderen Herstellers ersetzen, so hätte dies sicherlich einen starken Einfluss auf das z.Zt. hohe Preisniveau von RTI-Software (ca. EUR 6000 für bis zu 5 Java-Federates, (Pitch 2003)).

So findet der Einsatz von HLA zur Unterstützung von heterogenen Simulationsumgebungen im zivilen Bereich nur zögerlich Verbreitung. Hier haben die Industriebetriebe sich eher direkt auf CORBA eingelassen (Tolk 2002). Einige Erweiterungen wie IDL4HLA (Reilly und Williams 2001) oder GERTICO (Herzog u. a. 2001) versuchen den Übergang zwischen HLA und CORBA zu vereinfachen. Sie können aber die Abhängigkeit von einem RTI-Hersteller auch nicht vollständig auflösen.

4.3 DEVS

Die Abkürzung DEVS steht für *Discrete Event System Specification* und ist eine Möglichkeit zur formalen Beschreibung von Systemverhalten durch Ein- und Ausgabemengen sowie Funktionen auf diesen Mengen. Der Formalismus wurde von Zeigler (1976) entwickelt. Eine überarbeitete, ausführliche Beschreibung der theoretischen Basis findet sich in (Zeigler u. a. 2000). Neben der Beschreibung diskreter, ereignisorientierter Systeme werden auch kontinuierliche Systeme und deren Beschreibungen (DESS, *Differential Equation System Specification*) und allgemeine zeitdiskrete Systeme (DTSS, *Discrete Time System Specification*) wie beispielsweise zelluläre Automaten eingebettet.

4.3.1 Klassische Form

DEVS dient dazu, das Ein- und Ausgabeverhalten eines Systems zu beschreiben. Die eigentliche Beschreibung ist also ein Modell im Sinne von Abschnitt 2.1. Die klassische Form einer DEVS ist ein Tupel M = $(X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$, wobei

X die Menge aller möglichen Eingabewerte,

Y die Menge aller möglichen Ausgabewerte,

S die Menge aller möglichen Systemzustände,

$\delta_{int} : S \mapsto S$ eine interne Zustandsübergangsfunktion ist

$\delta_{ext} : Q \times X \mapsto S$ eine externe Zustandsübergangsfunktion ist, mit

$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$

e ist die vergangene Simulationszeit seit dem letzten Zustandsübergang

$\lambda : S \mapsto Y$ die Ausgabefunktion ist, und
$ta : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ eine Funktion ist, die jedem Zustand eine Simulationsdauer zuweist.

Das Systemverhalten lässt sich ausgehend von einem Initialzustand berechnen, indem die interne Zustandsübergangsfunktion δ_{int} angewendet wird, falls zu einem Zeitpunkt kein Eingabewert vorhanden ist. Die Funktion ta (*time advance*) dient dann zum Fortschreiben der Simulationszeit, d.h. der aktuelle Zustand dauert eine bestimmte Zeitspanne an. Die Zeitspanne 0 deutet dabei einen transienten Zustand an, und bei unendlicher Dauer spricht man von einem passiven Zustand. Liegt eine Eingabe vor, so ist die externe Zustandsübergangsfunktion δ_{ext} anzuwenden. Die Systemausgabe ergibt sich immer durch Anwendung der Ausgabefunktion λ.

4.3.2 Ports

Um mehrere Ein- und Ausgänge eines Systems zu betrachten und somit die Modellierung zu vereinfachen, erfolgte die Einführung von Ports. *Input Ports* stimulieren das System an einer bestimmten Stelle und *Output Ports* erlauben das Ablesen einer bestimmten Teilausgabe. Die Eingabemenge X besteht damit aus Paaren von Ports und zugehörigen Werten. Analog können auch die Elemente der Ausgabemenge Y als Paare angegeben werden. Auch (Mügge und Meyer 1996) verwenden Ports, um mehrere Modellein- und -ausgänge zu unterscheiden. In Abschnitt 6.4 wird später das Port-Konzept wieder aufgegriffen und verfeinert werden.

4.3.3 Kopplung

Die Vernetzung von Modellen lässt sich ebenfalls im DEVS-Formalismus darstellen. Dazu baut man ein Tupel $N = (X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, Select)$ auf, wobei

X die Menge aller möglichen Eingabewerte,

Y die Menge aller möglichen Ausgabewerte,

D die Menge aller Komponentennamen,

M_d eine DEVS mit Ports,

$EIC \subseteq \{((N, ip_N)(d, ip_d)) | d \in D, ip_N \in InpPorts, ip_d \in InpPorts_d\}$,

$EOC \subseteq \{((d, op_d)(N, op_N)) | d \in D, op_N \in OutPorts, op_d \in OutPorts_d\}$,

$IC \subseteq \{((a, op_d)(b, ip_d)) | a, b \in D, a \neq b, op_a \in OutPorts_a, ip_b \in InpPorts_b\}$ und

$Select : 2^D \setminus \emptyset \mapsto D$ eine Auswahlfunktion ist.

Die Relation *External Input Coupling* (EIC) gibt dabei die Verknüpfung zwischen Eingaben des vernetzten Modells N und den Eingaben der Komponenten M_d an. Ausgaben der Komponenten M_d, die als Ausgaben des vernetzten Modells zu verwenden sind, gehen in die Relation *External Output Coupling* (EOC) ein, und die Verknüpfungen zwischen den Modellkomponenten beschreibt die Relation *Internal Coupling*. Bei der internen Kopplung erlaubt DEVS keine direkte Rückkopplung einer Modellkomponente mit sich selbst (IC: $a \neq b$). Die Auswahlfunktion (Select) gibt die Modellkomponente an, deren Zustandsübergang zuerst berechnet werden soll, falls es zu einem Simulationszeitpunkt mehrere mögliche Reihenfolgen zur Berechnung geben sollte.

Für die Berechnung des Ausgabeverhaltens von Modellkopplungen wurde ein Rahmenwerk eingeführt, in dem je Spezifikationssystem (DEVS, DESS, DTSS) ein eigener Simulator existiert. Diese Simulatoren gehorchen einem speziellen Nachrichtenprotokoll und werden durch einen *Koordinator* gesteuert. Die Datenübertragung zwischen den einzelnen Komponenten erfolgt bei der Berechnung einer DEVS-Kopplung ausschließlich über den Koordinator, d.h. datenliefernde Komponenten werden vom Koordinator in der durch die Auswahlfunktion bestimmten Reihenfolge mit Eingabedaten versorgt und nach Ausgabedaten befragt. Der Koordinator verwaltet eine Ereignisliste und sorgt somit für die synchronisierte Auslieferung von Ereignissen an die empfangenden Komponenten.

DEVS verlangt also einen strikten Kopplungsaufbau, bei dem das gekoppelte Modell keine Eigenschaften der Kopplungsstruktur selbst zur Berechnung des Ausgabeverhaltens verwendet. Pidd und Castro (1998) verdeutlichen die Folgen dieser Einschränkung an einem Beispiel, das in seinen Grundzügen der Situation im Fässermodel aus Abschnitt 3.4 entspricht. Das Fässermodell enthält eine m-zu-n-Beziehung zwischen der Transportkomponente und den einzelnen Abfüllstationen. So gibt es zwei Ausgänge des Transportmodells (m=2, Bestellung und Rückgabe), die prinzipiell Einfluss auf alle n Abfüllstationen haben. Eine Modellierung mit DEVS erfordert eine Verbindung von jedem Ausgang des Transportsystems zu jedem Eingang einer Abfüllstation, also $m{\cdot}n$ Verbindungen. Durch eine zentrale Weiterleitungskomponente ließe sich zwar die Anzahl der Verbindungen auf $m + n$ reduzieren, allerdings muss diese Weiterleitung für jede (m,n)-Kombination neu definiert werden, weil DEVS keine Auswertung der Kopplungsstruktur zulässt. Deshalb schlagen Pidd und Castro einen anderen Kopplungsansatz vor, der dem gekoppelten Modell erlaubt, die Kopplungsbeziehungen zur Laufzeit auszuwerten. Sie nennen diesen Ansatz *selective external modularity* und präsentieren eine Umsetzung, die nur in der Programmiersprache C++ gültig ist.

Ein gekoppeltes DEVS-Modell ist modular, da der Datenaustausch ausschließlich über die deklarierten Ein- und Ausgänge der Komponenten erfolgt. Nicht modulare Kopplungen, die beispielsweise gemeinsame Variablen benutzen, lassen sich immer umwandeln in modulare Kopplungen, die nur über Ports Daten austauschen (Zeigler u. a. 2000, Abschn. 7.2.4). Damit ist DEVS abgeschlossen bzgl. Modellkopplungen, d.h. eine Kopplung von DEVS ergibt wieder eine DEVS, und eine hierarchische Modellierung von Systemen wird möglich.

4.3.4 Parallelisierung

Um die nebenläufige Berechnung von Modellkomponenten zu ermöglichen, musste bei der Kopplung die Auswahlfunktion der klassischen DEVS-Form eliminiert werden, da sie für die Sequentialisierung aller Berechnungen sorgt. In der parallelen Form ist es einer Modellkomponente erlaubt, alle anstehenden Ausgaben unmittelbar an alle vorgesehenen Empfänger weiterzuleiten. Die empfangenden Komponenten können deshalb in einem Berechnungsschritt mehrere Ausgaben erzeugen und mehrere Eingaben erhalten, was sich in der formalen Notation niederschlägt. Dort werden nämlich Sammlungen von Eingabewerten (engl. *bags* oder *multisets*; also Mengen, die Elemente mehrfach enthalten können) als Parameter für die Zustandsübergangsfunktion δ_{ext} und Sammlungen von Ausgabewerten in der Ausgabefunktion λ verwendet.

Es kann vorkommen, dass eine Modellkomponente in einem Berechnungsschritt sowohl einen internen Zustandsübergang vollziehen soll und gleichzeitig Eingaben vorliegen. In der klassischen Form sorgte die Auswahlfunktion für eine Regelung bezüglich der Berechnungsreihenfolge. Die parallele Form verlangt nun von den Empfängern, dass sie mehrfache Eingaben korrekt behandeln. Daher wurde für diese Situation (in Zeigler u. a. 2000, S. 90, als *confluent* bezeichnet) eine weitere Zustandsübergangsfunktion δ_{con} eingeführt, die die gleiche Signatur wie δ_{ext} besitzt.

Zur Berechnung des Ausgabeverhaltens paralleler DEVS ist ein Synchronisationsverfahren notwendig, um die inhärente Nebenläufigkeit so zu beschränken, dass die Kausalität der versendeten Ereignisse gewährleistet ist. In (Zeigler u. a. 2000, Kap. 11) sind sowohl konservative als auch optimistische Verfahren beschrieben, die in Verbindung mit DEVS eingesetzt werden. Von besonderer Bedeutung ist hier das konservative Verfahren, das Lookahead-Werte je Port berechnet (vgl. Abschnitt 4.1.3 und Abbildung 4.4). Dadurch ist eine feingranulare Auflösung von Verklemmungen möglich, die auch für den CoSim-Ansatz verwendet werden wird (Abschnitt 4.1.3).

Die Datenübertragung zwischen den einzelnen parallelen Komponenten erfolgt bei der Berechnung einer DEVS-Kopplung ausschließlich nach dem Push-Prinzip, d.h. datenliefernde Komponenten sorgen selbsttätig für die Übertragung von Daten zunächst an den Koordinator. Der Koordinator sorgt nur für die Weiterleitung von Ereignissen an die empfangenden Komponenten. Jeder Simulator muss dabei das verwendete Synchronisationsverfahren berücksichtigen.

4.3.5 Beschreibungssprache

Für die Beschreibung von DEVS-Komponenten und deren Kopplungen ist eine eigene Sprache vorgesehen (Zeigler u. a. 2000, Abschn. 17.2). Diese DEVS-Texte lassen sich von einer Ausführungsumgebung direkt interpretieren. Damit ist keine manuelle Codierung für eine spezielle Programmiersprache mehr notwendig. Einen ähnlichen Ansatz zur Kopplungsbeschreibung verfolgten (Mügge und Meyer 1996) im Rahmen des Forschungsprojektes MOBILE. Dort war eine Kopplung auch mit grafischen Mitteln möglich (Hilty u. a. 1998, Abschnitt B.3), wie sie sich bei vielen anderen komponentenorientierten Simulationswerkzeugen inzwischen durchgesetzt hat.

Die DEVS-Beschreibungssprache sieht jedoch nur drei Datentypen vor: Zeichenketten, Ganzzahlen und reellwertige Zahlen. Modellspezifische Datentypen

sind nicht darstellbar. Sie werden nur in Simulationssystemen, die von Modellentwicklern einen Programmieranteil fordern, zur Strukturierung des Datenaustauschs zwischen Modellteilen benötigt. Die DEVS-Sprache dient ausschließlich zur Beschreibung von einzelnen Zustandsübergängen. Hier ist eine Programmierung im Sinne von Steuerung eines Kontrollflusses nicht vorgesehen.

4.3.6 Umsetzungen

Der DEVS-Formalismus selbst ist zunächst unabhängig von bestimmten Anwendungsgebieten und auch von konkreten Programmiersprachen. Eine Nähe zum Programmierparadigma der prozessorientierten Simulation sehen Praehofer u. a. (2000) dadurch gegeben, dass Prozessroutinen die Kontrolle über einen Simulationslauf wieder abgeben, wenn ein Zeitfortschritt erfolgen soll, oder wenn Wartebedingungen erfüllt sind. Ähnlich stößt auch die DEVS-Steuerung bei der Berechnung des Ein-/Ausgabeverhaltens die Zustandsübergänge in verschiedenen Modellkomponenten an. Eine prozessorientierte Modellierung sieht jedoch keine modulare Modellerstellung vor. Hier bietet DEVS mit dem Kopplungsansatz eine Erweiterung.

Um die Modellierung mittels DEVS nun tatsächlich durchzuführen, existieren für verschiedene Programmiersprachen bereits Umsetzungen des DEVS-Formalismus. Dabei eignen sich besonders objektorientierte Sprachen, um eine Klassifizierung der verschiedenen DEVS-Formen direkt zu unterstützen (Zeigler u. a. 2000). So finden sich Realisierungen für das Common Lisp Object System (Sevinc 1991), C++ (Zeigler und Kim 1996) und Java (Sarjoughian 2002).

Auch eine programmiersprachenunabhängige Umsetzung mittels CORBA beschreiben (Zeigler u. a. 1999b) und (Cho 2001, Kap. 4). Dabei stellt (Cho 2001, S. 53) fest:

> Closure under coupling implies that when networking DEVS components, one CORBA interface will suffice for all model classes.

dort ist die Verwendung von Ports in der Schnittstellenbeschreibung vorgesehen, so dass die feingranulare Auflösung von Verklemmungen bei der Synchronisation möglich wird. Um die Implementation der DEVS/CORBA-Schnittstellen zu vereinfachen, wird der Einsatz einer DEVS/HLA-Realisierung (Zeigler u. a. 1999a) und der CORBA-basierten HLA-Realisierung (DMSO 2000) vorgeschlagen. Damit hängt diese DEVS-Umsetzung wieder vom Hersteller der HLA-Laufzeitumgebung ab (vgl. Abschnitt 4.2).

4.4 Komponentenorientierte Simulation und Web-Basierte Techniken

Mit der verstärkten Nutzung von WWW-Diensten seit Mitte der 90er Jahre nahm auch der Begriff „web-basierte Simulation“ auf entsprechenden Fachtagungen (WinterSim, PADS) einen immer größeren Anteil an Veröffentlichungen ein und führte schließlich zu einer eigenen Konferenz (Fishwick u. a. 1998). Unter dem Begriff web-basierte Simulation lassen sich eine Reihe verschiedener Techniken wie CGI, Java und JavaBeans einordnen.

4.4.1 Skriptbasierte Lösungen

Häufig wird unter web-basierter Simulation schon die Ausführung von Simulationsläufen aus einem Browser heraus verstanden (Straßburger und Schulze 2001), wobei ein WWW-Server parametrisierte Anfragen abarbeitet und dazu einen Simulationslauf startet. Ein simpler Mechanismus zum Ausführen von Skripten aus einer WWW-Anfrage heraus ist beispielsweise das Common Gateway Interface (CGI, Tittel u. a. 1996), das einen Prozess erzeugt und externe Anwendungen aufruft. Ziel dieses Vorgehens ist es, dass ein Simulationsendanwender leichten Zugang zu vorhandenen Simulationsmodellen erhält und dabei die gewohnte Arbeitsumgebung (WWW-Browser) nicht verlassen muss. Zudem lassen sich modellangepasste Ein- und Ausgaben wie Formulare, multimediale Präsentationen oder Animationen leichter integrieren. (Miller u. a. 2000b). Die hinter dem WWW-Server stehende Simulationsanwendung wird jedoch nicht offener im Sinne einer verbesserten Koppelbarkeit mit anderen Anwendungen. Prinzipielle Schwierigkeiten mit dem CGI-Einsatz sehen Dorwarth u. a. (1997), da sich viele Simulatoren noch nicht einmal per Skript aufrufen lassen, sondern eine in sich geschlossene Umgebung darstellen.

4.4.2 Java

Die Portierung vieler Simulatoren in die Programmiersprache Java sehen Kuljis und Paul (2000) als weiteren Schwerpunkt der web-basierten Simulation und nennen verschiedene Hintergründe.

- Durch die Verfügbarkeit der Laufzeitumgebung (Java Runtime Environment) für verschiedene Plattformen lassen sich auch die in Java realisierten Simulationsmodelle einfacher verbreiten.
- Simulationsmodelle lassen sich ohne zusätzlichen Übersetzungsaufwand durch den Modellanwender auch als Java-Applet innerhalb eines Browsers ausführen. Komplexe Benutzungsschnittstellen und Animationen sind so noch leichter realisierbar.
- Leichtgewichtige Prozesse (Java-Threads) erlauben eine direkte Umsetzung der Prozessinteraktionen im Rahmen der prozessorientierten Modellierung.

Sie kritisieren jedoch, dass viele der Java-Umsetzungen eher von der Faszination der Technik getrieben werden als von der wirklichen Notwendigkeit neuer Werkzeuge. Der Einsatz einer bestimmten Programmiersprache allein bringt noch keinen Gewinn hinsichtlich Komponentenbauweise und Wiederverwendung. So stellen Page u. a. (1998) auch fest:

> Simulation models have been traditionally monolithic in design. The advent of object-oriented programming has resulted in more elegantly designed monoliths.

4.4.3 JavaBeans

Das Komponentenmodell JavaBeans (Hamilton 1997) erlaubt es, in sich abgeschlossene Einheiten (Beans) zu spezifizieren. Beans können Attribute (Pro-

perties) bereitstellen, die synchron über Lese- und Schreiboperationen zugreifbar sind. Für den asynchronen Zugriff gibt es einen Nachrichtenmechanismus (Events), wobei die Beans ihre Bereitschaft zum Versand oder Empfang entsprechender Nachrichten vorab deklarieren. Die Deklaration von Properties und Events geschieht durch Verwendung spezieller Deskriptor-Objekte oder durch den Reflexionsmechanismus der Programmiersprache Java. Enthält eine Java-Klasse Methoden, die nach einem festgelegten Schema benannt sind, so kann eine Ausführungsumgebung für JavaBeans (BeanBox) diese Klasse als Bean mit entsprechenden Eigenschaften reflektiv erkennen. Die BeanBox stellt automatisch Werkzeuge zur Bearbeitung von Properties und zur Angabe von Kommunikationsbeziehungen bereit. Dazu können datentypabhängige Eingabemasken und Grapheneditoren gehören.

Mit der Einführung von JavaBeans versuchten verschiedene Simulationssysteme (JSIM: Miller u. a. 2000a, b), (SILK: Kilgore und Healy 1998), (COSIMA: Oses 2002), (Praehofer u. a. 2000) oder (Buss 2000), eine baukastenorientierte Modellierung in Java zu erreichen. Sie präsentieren alle einen eigenen Simulationskern zur Ereignisplanung und sind untereinander leider nicht interoperabel.

Der JavaBeans-Ansatz erscheint jedoch viel versprechend, wenn man Beans als logische Prozesse auffasst, wie sie in Abschnitt 4.1 für die verteilte Simulation diskutiert wurden, und die verschiedenen Nachrichtenmechanismen (Properties und Events) benutzt. (Page u. a. 1997) haben bereits gezeigt, wie sich JavaBeans für die Simulation mittels RMI in einer verteilten Umgebung nutzen lassen. Bei einem geeigneten Synchronisationsmechanismus zur Kausalitätssicherung kann man sogar auf eine zentrale Zeitverwaltung verzichten. Der CoSim-Ansatz wird die Synchronisation in Abschnitt 6.6 und die Kopplung von Komponenten über synchrone und asynchrone Nachrichten in Abschnitt 6.5 wieder aufnehmen.

4.4.4 MOOSE

Ein Rahmenwerk zur Entwicklung von verteilten Simulationsmodellen stellt MOOSE (Multimodeling Object-Oriented Simulation Environment) zur Verfügung (Cubert und Fishwick 1997, 1998a). Für die Modellierung mit MOOSE sind mehrere Beschreibungssprachen vorgesehen, die den Einsatz von Teilkomponenten unterschiedlicher Herkunft erlauben. Durch ein entsprechendes Datenaustauschformat verspricht MOOSE somit Plattformunabhängigkeit. Die einsetzbaren Modelltypen decken dabei eine Reihe verschiedener Ansätze ab: Endliche Automaten, Differentialgleichungssysteme, funktionale Blockmodelle und regelbasierte Modelle. Die erstellten Modelle müssen abgeschlossen bezüglich der Modellkopplung sein, d.h. es existieren Regeln, unter denen die Verfeinerung eines Modells zu einem anderen Modelltyp möglich ist und wie die Kopplung verschiedener Modelltypen vollzogen wird.

Im Hintergrund des MOOSE-Systems arbeitet ein Übersetzer, der die Modelldefinitionen in Code für übliche Programmiersprachen (hier C++) transformiert. Der Ausführungsort eines Modells lässt sich somit direkt bestimmen, da auf jedem MOOSE-Wirtsrechner ein Übersetzer arbeitet.

Die Modelldefinitionen werden in Modellverzeichnissen (MOOSE Model Repository, MMR) abgelegt, die auch verteilt sein können (Cubert und Fishwick 1998b). Die Werkzeuge der MOOSE-Umgebung erlauben es nun, die Beschreibung eines entfernten Modells zu laden und sie in einer dem Modelltyp angepassten Umgebung zu bearbeiten. Zur Modellkopplung ist ein grafischer Editor

vorgesehen, der ähnlich wie bei DEVS-Kopplungen (Abschnitt 4.3.2) Ein- und Ausgänge der Teilmodelle verbindet.

4.4.5 Sprachunabhängige Ansätze

Neben den bereits ausführlicher behandelten Ansätzen HLA und DEVS existieren noch einige weitere komponentenorientierte Entwicklungen, die unabhängig von einer bestimmten Programmier- oder Modellierungssprache sind, jedoch weniger Verbreitung gefunden haben. Sie setzen jeweils eine Beschreibungssprache ein, um die Zugangspunkte für Kopplungen in den einzelnen Komponenten zu identifizieren. Meistens wird als Kommunikationsinfrastruktur CORBA verwendet, und die Schnittstellenbeschreibung erfolgt per IDL.

TENT

Die Umgebung TENT (Testbed for Numerical Turbine Simulation) des Deutschen Zentrums für Luft- und Raumfahrt führt Komponenten ein, um eine einheitliche Benutzungsschnittstelle für die Konfiguration von Simulationsläufen auf verschiedenen Parallel- und Hochleistungsrechnern anbieten zu können (Padur und Schreiber 2001). Hier umfasst der Komponentenbegriff nicht nur die unmittelbar an der Berechnung des Modellverhaltens beteiligten Programme, sondern auch Partitionierungswerkzeuge für die parallele Abarbeitung der Modellberechnung, Filter für Ergebnisdaten und Visualisierungswerkzeuge. Ziel ist also die Erfassung der kompletten Abarbeitungsschritte für eine Simulationsstudie, und nicht der Aufbau neuer Modelle aus vorhandenen Modellteilen.

Die Komponenten werden sprachunabhängig gesteuert durch Methodenaufrufe mittels der Kommunikationsinfrastruktur CORBA. Da die zwischen den Komponenten zu übertragenden Datenmengen aber sehr groß werden können, verwendet TENT aus Performanzgründen eine Kombination verschiedener Kommunikationswege außerhalb der CORBA-Umgebung (Sockets, FTP u.a). Lediglich die Identifikation der Datenaustauschpunkte erfolgt durch Datenstrukturen, die per CORBA definiert wurden (Faden u. a. 1999).

CUDOs

Das Produkt pLUG&SIM^{TM} der Firma WindRiver (vormals Integrated Systems, ISI 1999) entstand vor dem Hintergrund der Integration von Software, Hardware und mechanischen Komponenten (*Cosimulation*). Es stellt CORBA-Schnittstellen unter dem Namen CUDOs (Cosimulation Using Distributed Objects) vor, die die Einbindung verschiedener Simulatoren ermöglichen. Für die Simulationswerkzeuge aus der eigenen Produktpalette stehen entsprechende Adapter zur Verfügung. Diese Adapter realisieren sog. Modellfabriken und erlauben es, neue Modellexemplare nach Bedarf zu erzeugen.

CUDOs stellt genau drei Datentypen für die Simulation bereit (float, int, boolean), modellspezifische Datentypen sind nicht verwendbar. Die Beschreibungen sämtlicher Modellobjekte werden als XML-Dokument angegeben und müssen per Dateisystem erreichbar sein. Ein direkter Zugriff auf entfernte CORBA-Modellobjekte ist somit nur eingeschränkt möglich. Zudem erfolgt die Bindung von CORBA-Objekten ORB-spezifisch. Hier wird der ORBIX

`_bind`-Mechanismus eingesetzt (http://www.iona.com). Ein Synchronisationsverfahren für die verteilte Simulation logischer Prozesse ist nicht vorgesehen.

MOBILE

Das Forschungsprojekt MOBILE (Model Base for an Integrative View of Logistics and Environment) zielt auf die Entwicklung eines Systems ab, das die Auswirkungen verkehrsrelevanter Maßnahmen auf die Umwelt abzuschätzen hilft (Hilty u. a. 1998). Dazu wurde ein verteiltes, objektorientiertes Modellbanksystem mit zugehöriger Simulationssteuerung entworfen. Die Integration eines Geografischen Informationssystems (GIS, vgl. Abbildung 4.10) ermöglicht eine raumbezogene Modellierung.

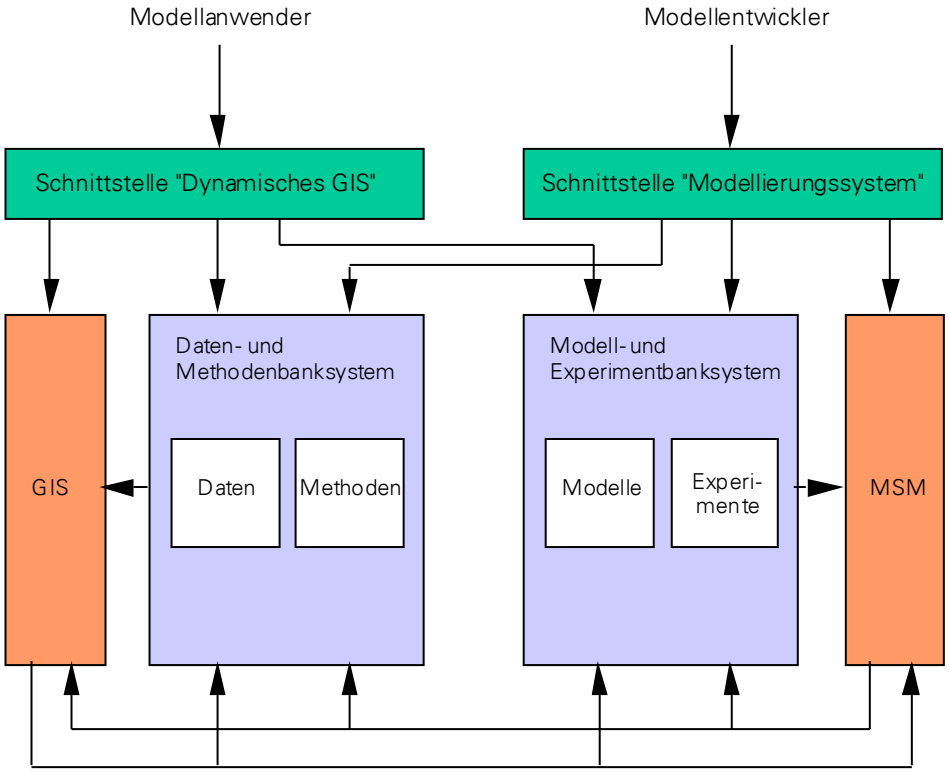


Abbildung 4.10: MOBILE Architektur, aus (Hilty u. a. 1998, S. 79)

Die Beschreibung der Struktur von Modellkopplungen und Simulationsexperimenten kann sowohl textuell in einer Deklarationssprache erfolgen (Mügge und Meyer 1996) als auch grafisch mit Hilfe eines Grapheneditors (Abbildung 4.11). Die Modellkopplung verwendet das aus DEVS bekannte Konzept der Ports (Abschnitt 4.3.2). Durch die Verwendung von Experimentklassen in der Strukturbeschreibung lassen sich komplexe Experimentierszenarien aufbauen, die auch konkurrierende Modelle und Ausführungsschleifen enthalten können.

Der Editor für die grafische Sprache (GMSL, Graphical MOBILE Scripting Language) übersetzt ein erstelltes Diagram in die Textrepräsentation MSL, die dann von einem Kopplungsautomaten (MSM, MOBILE Scripting Machine) in-

terpretiert werden soll. Für die Kommunikation zwischen den Modellteilen, die verteilt vorliegen können, war eine DCE-Realisierung vorgesehen (Becken und Bosselmann 1998). Der Kopplungsautomat wurde jedoch niemals realisiert, so dass die verteilte Ausführung von komplexen Modellen nur konzeptuell blieb.

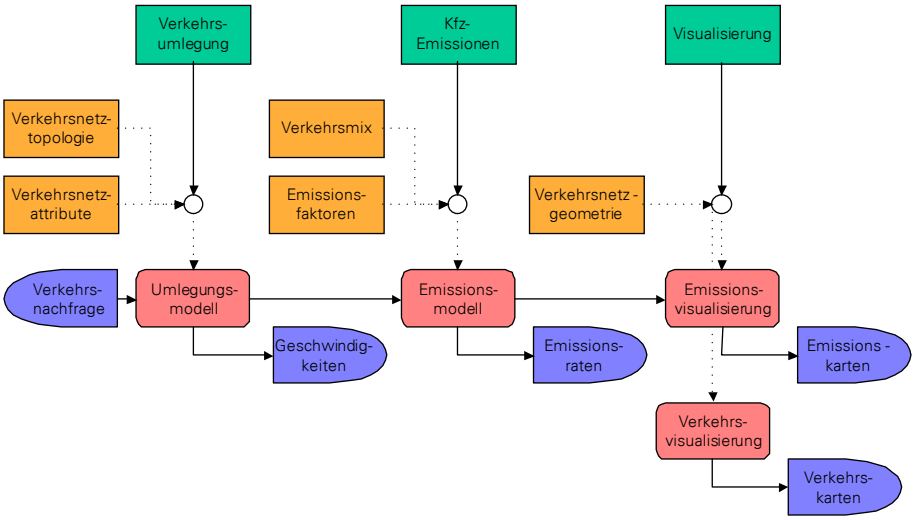


Abbildung 4.11: Experimentspezifikation in GMSL, aus (Hilty u. a. 1998, S. 129)

Weitere CORBA-basierte Ansätze

Einen Ansatz zur Erweiterung der existierenden Integrationsumgebung ENVISION (Environment for integrating simulation models interactively over networks) schlägt (Heim 1997) vor. Hier soll ein Modell als CORBA-Objekthülle repräsentiert und durch einen Agenten unterstützt werden, der die Modellfähigkeiten beschreibt. Um die Modellteile auffinden zu können, ist eine grafische Umgebung geplant, die auf ein LDAP-basiertes Verzeichnis zugreift. Ob dieses Konzept tatsächlich umgesetzt wurde, und welche Möglichkeiten die IDL-Schnittstellen bieten, bleibt jedoch offen.

Einen zentralen Simulationsdienst zur nachrichtenbasierten, ereignisorientierten Simulation präsentiert Shen (2000). Hier ist ein Synchronisationsverfahren direkt mit der Verwaltung einer Ereignisliste gekoppelt. Dieser Ansatz bietet keine Möglichkeit zur hierarchischen Simulation und ist nicht komponentenorientiert.

Chan und Spracklen (1999) schlagen einen agentenbasierten Ansatz zum Auffinden vorhandener Simulationsmodelle vor. Sie verwenden CORBA, um die Agenten miteinander kommunizieren zu lassen. Warum hier nicht der standardisierte CORBA-Namensdienst verwendet wird, bleibt unklar. Auch über die genauen Schnittstellen der Modellkomponenten und deren Mächtigkeit wird keine Aussage gemacht.

Das Rahmenwerk JaCo3 (Gouache und Priol 2000) dient dazu, ausführbaren Simulationscode auf entfernte Rechnerumgebung zu verschieben. Dazu wurden verschiedene Möglichkeiten zum Dateizugriff in IDL formuliert und als CORBA-Objekte realisiert. Weiterhin entstanden IDL-Erweiterungen, um mögliche Par-

allelisierungen zu beschreiben. Die Interoperabilität dieses Ansatzes bewegt sich damit in einem sehr engen Rahmen. Zum einen können nur Rechnerumgebungen eingesetzt werden, auf denen die notwendige Ausführungsumgebung der speziellen Simulationscodes vorhanden ist, zum anderen verlangen die IDL-Erweiterungen spezielle Compiler, um den Code für die Parallelisierungen zu erzeugen.

Mit der Wiederverwendung und Kopplung von Modellteilen aus dem speziellen Anwendungsgebiet der Warteschlangenmodelle für Netzwerke beschäftigen sich Iazeolla und D'Ambrogio (1998). Sie beschreiben die Schnittstelle eines Warteschlangenmodells mittels CORBA-IDL und realisieren Java-basierte Clients mit grafischer Oberfläche für die Experimentdurchführung. Zwischen den beteiligten Simulationsanwendungen werden jedoch die Modellspezifikationen als Texte ausgetauscht, so dass neben der reinen IDL-Spezifikation noch eine bestimmte Modellierungssprache eingesetzt werden muss.

Ebenfalls für die Simulation des Datenverkehrs in Netzwerken dient das Rahmenwerk aus (Cholkar und Koopman 1999). Dort können verschiedene Arten von Knoten innerhalb eines Netzwerkes unterschieden werden. Der Nachrichtenaustausch zwischen den Knoten erfolgt durch IDL-definierte Nachrichtenkanäle und ein eigenes Nachrichtenformat.

5 Bedarf für einen neuen Ansatz

Im letzten Kapitel wurden verschiedene Ansätze zur verteilten, komponentenorientierten Simulation vorgestellt. Hinter jedem entsprechenden Entwurf steht die Zielsetzung der Wiederverwendung, zum Teil jedoch auf unterschiedlichen Ebenen. Wiederverwendung wird dabei als nützlich angesehen, weil sie

- die Kosten für den Aufbau komplexer Modelle senkt,
- den zeitlichen Aufwand zur Modellerstellung senkt,
- die Entwicklung valider Komponenten fördert oder
- die Durchführbarkeit umfangreicher Simulationsstudien erst ermöglicht.

Durch den Einsatz wiederverwendbarer Komponenten kann mehr in andere Teile einer Simulationsstudie investiert werden, wie

- Datenbeschaffung,
- Experimentauswertung oder
- Validierung komplexer Modelle.

Weiterhin wird gezeigt, dass ein Ansatz, der die Wiederverwendung von Modellen und Modellteilen fördert,

- herstellerunabhängig ist,
- sprachunabhängig ist,
- auf einer Blackbox-Wiederverwendung basiert,
- den entfernten Zugang zu Modellkomponenten ermöglicht,
- Mechanismen zur Synchronisation zwischen den Modellkomponenten enthält,
- die Verwendung modellspezifischer Datentypen erlaubt und
- zwischen verschiedenen Nutzungsphasen eines Modells trennt.

Bisherige Ansätze enthalten nur eingeschränkte Möglichkeiten der Wiederverwendung. Das wird eine Bewertung bezüglich dieser Eigenschaften in Abschnitt 5.1 ergeben. Deshalb lohnt es sich, in den folgenden Kapiteln einen neuen Ansatz für die verteilte, komponentenorientierte Simulation zu entwerfen.

5.1 Nutzen von Simulationskomponenten

5.1.1 Kapseln für Systemzusammenhänge

In Simulationskomponenten lassen sich die Erkenntnisse über die Zusammenhänge innerhalb von Teilsystemen zusammenfassen (Miller u. a. 2001). Die intellektuelle Leistung der Modellierung wird so als Software verfügbar gemacht (Heim 1997). Sollten sich diese Erkenntnisse ändern, dann reicht es häufig aus, eine Komponente zu überarbeiten, wenn die Form der Interaktionen zwischen

dem modellierten Teilsystem und seiner Umwelt unverändert geblieben ist. Damit kann der Zugriff auf die Komponente unverändert bleiben. Die Kapselung der Modellierung lässt sich auch gezielt einsetzen, um diese intellektuelle Leistung zu schützen und vor Konkurrenten auf einem Markt für Komponenten zu verbergen (Straßburger 2001).

Komponenten fördern den hierarchischen Aufbau von Modellrealisierungen. Komplexe Modelle lassen sich so leichter beherrschen, und es wird einfacher, auf verschiedenen Hierarchiestufen unterschiedliche Detaillierungsgrade zu verwenden (Oses 2002, Abschn. 2.3.1). Teilweise wird die Ausführung von Simulationsläufen mit umfangreichen Modellrealisierungen sogar durch die Aufteilung auf kleinere Einheiten erst ermöglicht. Steht beispielsweise für das Gesamtmodell auf einem einzelnen Rechner nicht genügend Speicherplatz zu Verfügung, so wird das Modell zerlegt und auf mehreren Rechnern verteilt ausgeführt.

Die Granularität einer Komponente bestimmt mittelbar ihre Einsatzmöglichkeiten. Hier ergibt sich ein Wechselspiel zwischen Funktionalität und Nutzbarkeit (Miller u. a. 2001). Ist eine Komponente groß und bietet viel Funktionalität an, so ist sie in nur wenigen Anwendungfeldern einsetzbar, wird in einem speziellen Gebiet vielleicht aber häufig eingesetzt. Kleinere Komponenten lassen sich hingegen für viele Zwecke einsetzen, bieten jedoch nur einen geringen Funktionsumfang. Die Anzahl konkurrierender Komponenten wird hier zunehmen. Das richtige Maß für die Granularität zu finden, ist in der Simulation auch nicht einfacher als für den allgemeinen Einsatz von Komponenten.

5.1.2 Aufwandssenkung

Generell sinkt natürlich die Programmierlast für eine Modellrealisierung, wenn schon Funktionsbibliotheken oder eben Komponenten zur Verfügung stehen, weil der Aufwand zur Erstellung der tieferen Modellierungsebene nur einmalig anfällt. Die Wiederverwendung selbst verlangt die Angabe der Kopplungsbeziehungen und setzt u.U. noch eine Konfiguration voraus. Dieser Aufwand ist im Vergleich zur Komponentenentwicklung aber zu vernachlässigen (Rabe 1999).

Unter der Voraussetzung, dass die eingesetzten Komponenten gut dokumentiert und gemäß ihrer Spezifikation korrekt realisiert sind, kann sich auch der Aufwand für die Validierung eines Gesamtmodells verringern (Heim 1997). Umgekehrt gerät die Modellvalidierung zu einer sehr aufwändigen Aufgabe, wenn eine Komponente nicht ihrer Spezifikation entspricht.

Häufig wird angenommen, dass zur Verfügung stehende Komponenten auch mit wenig Aufwand änderbar sind (Iazeolla und D'Ambrogio 1998). Nur wenn jedoch der Quellcode einer Komponente vorliegt, die nicht den Einsatzanforderungen genügt, sind kleinere Änderungen durchführbar. Das ist bei der Wiederverwendung nach dem Blackbox-Prinzip nicht der Fall. Auch wenn der Code vorliegt, kann er zu einem Simulationssystem gehören, mit dem der Modellentwickler nicht vertraut ist. Ein entsprechender Einarbeitungsaufwand ist in diesem Fall erheblich (Rabe 1999).

5.1.3 Kostensenkung

Auf einem Markt für Simulationskomponenten kommt der Erstellungsaufwand einer Komponente vielen Nutzern zu Gute, so dass der Preis je Komponente für den einzelnen Nutzer sinkt (Rabe 1999). Mit geringeren Kosten kann auch

die Nachfrage nach Simulationskomponenten steigen, so dass viele Komponenten entstehen und verschiedene Anwendungsbereiche abdecken. Dabei sollten sich dann automatisch diejenigen Modelle durchsetzen, die eine schnelle und effiziente Durchführung gewährleisten (Page u. a. 1998).

Bisher zögern die Hersteller jedoch noch, komponentenorientierte Simulationsmodelle zu unterstützen. Ein Softwarehersteller müsste mit einem Angebot den ersten Schritt machen, aber es lässt sich schlecht abschätzen, welche Einnahmen durch den Komponenteneinsatz zu erwarten sind (Miller u. a. 2001). Zudem hat der Markt für Simulationssysteme noch keine Standardisierungsbemühungen in großem Stil angenommen. Meist sind lediglich Schnittstellen erhältlich für den Datenaustausch mit Datenbanken oder Tabellenkalkulationen. Dadurch ist es für die Modellhersteller unattraktiv, Teilmodelle anzubieten, weil diese Teile für jeden Simulator neu implementiert werden müssen (Rabe 1999). Kunden kaufen auch ungern Produkte, mit denen sie sich an einen Hersteller binden, wenn Ersatz, Erweiterungen oder Ergänzungen benötigt werden.

Benötigt wird also ein Industriestandard, mit dem sich Teilmodelle kombinieren lassen. Dann muss ein Hersteller keine komplette Produktpalette mehr anbieten, sondern kann sich auf sein Kerngebiet konzentrieren. Es ist sogar ein positiver Rückkopplungszyklus zu erwarten (Miller u. a. 2001): Je mehr Teilmodelle für eine standardisierte Technik zur Verfügung stehen, desto eher wird man auch Kopplungen unter dieser Technik sehen. Ein erster Ansatz in diese Richtung ist mit HLA (Abschnitt 4.2) erfolgt, doch hier sind noch Herstellerabhängigkeiten vorhanden.

Auch in der Lizenzierung können neue Wege beschritten werden. Bisher müssen Benutzer, die spezielle Mechanismen einer Software benötigen, ein sehr teures Produkt mit viel nicht benötigter Funktionalität kaufen (Miller u. a. 2000a). So schlagen Miller u. a. (2001) etwa vor, dass Komponenten einer Gesamtlösung für Modellbildung und Simulation nur gekauft werden müssten, sobald ihr Einsatz notwendig wird. Zunächst könnte nur eine Komponente zur Ausführung von entfernten Simulationsläufen existieren. Später könnten Komponenten hinzukommen, die komplexe Experimente, statistische Analysen und Optimierungen ermöglichen. Generell bräuchten den Benutzern nur die tatsächlich anfallenden Nutzungszeiten in Rechnung gestellt zu werden. Auch für den Hersteller könnten sich flexiblere Nutzungsentgelte rechnen, indem günstigere Softwarekosten über die Zeit betrachtet zu wesentlich mehr Nutzungen führen. Möglich ist auch, dass eine Simulationslizenz nur zur Erstellung neuer Komponenten fällig wird (Rabe 1999).

5.1.4 Qualitätssteigerung

Ein generelles Problem bei der Wiederverwendung in der Simulation besteht in der Haltung „Not implemented here“ (Straßburger 2001; Oses u. a. 2002). Da ein Modell gültige Aussagen im Rahmen einer Simulationsstudie machen soll, vertrauen viele Modellersteller fremdem Code nicht. Sie können nämlich nur schwer überprüfen, ob eine fremde Komponente das zugesagte Verhalten auch tatsächlich korrekt implementiert. Liegt der Quellcode einer Komponente nicht vor, so kann die Verhaltensspezifikation nur durch umfangreiche Ein- und Ausgabetests erfolgen, und oft fehlt das Verständnis für die Randbedingungen, die für den erfolgreichen Einsatz einzuhalten sind. Dennoch besteht die Hoffnung, dass sich durch Marktprinzipien Modellkomponenten mit hoher Qualität

durchsetzen (Miller u. a. 2001).

Auch Rabe (1999) führt an, dass gerade auf einem Markt für Simulationskomponenten qualitativ hochwertige und vertrauenswürdige Bausteine entstehen können, denn fehlerhafte oder schlecht dokumentierte Produkte finden in einer Konkurrenzsituation gar keine Abnehmer. Komponenten, die für eine Blackbox-Wiederverwendung gedacht sind, sollten also einer gründlichen Test- und Validierungsphase unterzogen werden.

Es stellt sich auch noch die Frage, wer die Qualität dieser Modelle bewertet und sicherstellt. Page u. a. (1998) fordern deshalb, zunächst einmal geeignete Qualitätskriterien für Simulationskomponenten zu bestimmen. Eine Organisation zur Zertifizierung von Komponenten und den zugehörigen Spezifikation wäre hier denkbar.

5.1.5 Förderung der Kommunikation und Interaktion

Ein wesentlicher Faktor für den Erfolg von Simulationsprojekten sind die Kommunikation und die Interaktion zwischen Simulationsdienstleistern und deren Kunden (Miller u. a. 2001; Robinson und Pidd 1998; Taylor 2000). Führt ein Kunde häufiger Simulationsprojekte durch, so kann die Wiederverwendung von Komponenten zwischen den Projekten helfen, die Begriffsbildung zu festigen und damit die Kommunikation zu erleichtern. Bekannte Modellteile stärken zudem das Vertrauen in die Gültigkeit eines Modellentwurfs. Sind Komponenten von verschiedenen Orten aus nutzbar, so schlagen (Miller u. a. 2001) sowie (Taylor 2000) auch Werkzeuge zur Gruppenarbeit vor, die die Kommunikation zwischen Simulationsdienstleistern und Kunden beschleunigen und verbessern sollen.

5.2 Rahmenbedingungen der Wiederverwendung

Damit eine Simulationsinfrastruktur ihren Nutzen voll entfalten kann, sollte sie möglichst wenige Hindernisse für die Einbettung verschiedenster Modellkomponenten aufstellen. Die folgenden Rahmenbedingungen sind deshalb im Wesentlichen technischer Natur, d.h. eine entsprechende Werkzeugunterstützung sollte von einem Dienstleister, der Simulationsstudien durchführt, lediglich benutzt und nicht erst entwickelt werden.

5.2.1 Herstellerunabhängigkeit

Die Unabhängigkeit von den Softwareprodukten eines bestimmten Herstellers ist wesentlich für eine möglichst weitreichende Wiederverwendung existierender Simulationskomponenten. Müssten nämlich alle Werkzeuge zur Erstellung oder zur Bereitstellung von Simulationskomponenten innerhalb einer Simulationsinfrastruktur ein bestimmtes Produkt benutzen, so besteht die Gefahr von technischen Inkompatibilitäten (z.B. bei Betriebssystemen) oder von überhöhten Preisen. Lässt sich ein Produkt jedoch problemlos gegen ein anderes austauschen, so können Produktverbesserungen viel schneller in den Gesamtprozess einfließen. Voraussetzung für die Herstellerunabhängigkeit sind offene Schnittstellen, die häufig benötigte Dienste klar definieren. Diese Dienste können dann von verschiedenen Herstellern realisiert werden.

Gerade für den Bereich des Datenaustausches sind Abstimmungen über Datenformate sinnvoll. Hier haben herstellerabhängige Lösungen den Nachteil, dass für jedes Paar von zu koppelnden Systemen eine Austauschschnittstelle zu realisieren ist. Ein einheitliches Zwischenformat wie beispielsweise XML vermindert diesen Aufwand erheblich (Rabe 1999). Das Datenformat muss dabei alle Datenbeschreibungen vorhersehen, die im entsprechenden Anwendungsgebiet anfallen.

5.2.2 Sprachunabhängigkeit

Bei der Vielzahl unterschiedlicher Modellierungsprobleme kann eine einzige Modellierungssprache nicht allen Aufgaben angemessen sein. Der Versuch, eine umfassende Modellierungssprache für verteilte Simulationen zu entwerfen, um sie dann auf verschiedene Zielsysteme in eine allgemeine Programmiersprache zu übersetzen, ist regelmäßig fehlgeschlagen (Rabe 1999). Alleine die Definition von Steuerungsregeln und Strategien innerhalb eines Simulationsmodells verlangt nämlich in der Regel die explizite Programmierung und setzt eine entsprechend ausgestattete Programmiersprache voraus. Die Auswahl einer bestimmten Programmiersprache für die Simulation ist nicht wirklich ein gangbarer Weg, auch wenn diese Sprache (wie z.B. Java) als plattformunabhängig gilt: Die vorhandenen Lösungen, die in anderen Sprachen erstellt wurden, lassen sich nicht weiter nutzen. Der Aufwand für eine Portierung ist nur akzeptabel, wenn die Sprache über lange Zeit konstant bleibt und anerkannt wird. Dann wäre aber im Prinzip jede vollständige Programmiersprache auswählbar, die genügend Komfort für den Umgang mit Simulationsproblemen bietet.

Sinnvoller erscheint also die sprachunabhängige Definition von Modellschnittstellen, um die Interoperabilität zwischen existierenden Modellen zu gewährleisten (Straßburger 2001; Cubert und Fishwick 1998a). Durch verschiedene definierte Sprachanbindungen lässt sich dann für neue Modelle die jeweils geeignete Entwicklungs- und Ausführungsumgebung auswählen (Heim 1997).

Mit der Sprachunabhängigkeit muss jedoch die Möglichkeit aufgegeben werden, Simulationscode innerhalb eines verteilten Systems zu migrieren, um beispielsweise eine höhere Berechnungsgeschwindigkeit zu erreichen. In der Regel wird ja die Ausführungsumgebung einer bestimmten Simulationskomponente nicht an demjenigen Rechenknoten vorliegen, der gerade eine hohe Berechnungsgeschwindigkeit verspricht. Auch der Aufwand für die sprachunabhängige Anbindung existierender Simulationswerkzeuge kann erheblich sein und einigen Einschränkungen bzgl. der Möglichkeiten zum Datenaustausch unterliegen (Straßburger 2001). Gelingt jedoch die Anbindung bei einigen Werkzeugen, so kann das ein Anreiz für andere Hersteller sein, ihre Werkzeuge zu öffnen, um bessere Marktchancen zu erlangen.

5.2.3 Blackbox-Prinzip

Die Diskussion um die Verwendung des Blackbox- oder des Whitebox-Prinzips wurde bereits in Abschnitt 2.3.3 geführt. Es bleibt festzuhalten, dass die Wiederverwendung nach dem Whitebox-Prinzip die Möglichkeiten verwendbarer Sprachen sowie Ausführungsumgebungen einschränkt und den Austausch von Komponenten gegen weiterentwickelte Versionen erschwert. Um also die höchstmögliche Interoperabilität zu gewährleisten, sollte eine Simulationsinfrastruktur das Blackbox-Prinzip verwenden, auch wenn hierbei mit Widerständen zu rechnen

ist, wenn die Validierung der gesamten Modellrealisierung als kritisch angesehen wird (Page u. a. 1998).

5.2.4 Entfernter Zugang

Der entfernte Zugang zu Simulationskomponenten im Rahmen eines verteilten Systems wird allein dadurch notwendig, dass verschiedene Simulatoren einbezogen werden sollen, die plattformabhängig sind und ein bestimmtes Betriebssystem verlangen (Fujimoto 2000). Die Simulatoren bleiben dann auf ihrer Plattform bestehen und kommunizieren nur über festgelegte Schnittstellen mit anderen Simulationskomponenten.

Die Verteilung sollte sich auch nicht nur auf lokale Netzwerke beziehen, die z.B. ein gemeinsames Dateisystem betreiben. Man kann heutzutage etwa einen Zugang per TCP/IP voraussetzen, ohne die Zahl der zugänglichen Rechensysteme wesentlich zu reduzieren.

Damit existierende Simulationskomponenten von möglichen Nutzern aufgefunden werden können, werden Verzeichnisse benötigt, die direkt auf nutzbare Komponenten verweisen (Miller u. a. 2001; Tolk 2000). Komfortable Mechanismen zur Suche nach Modellkomponenten sind dabei sehr hilfreich und verlangen eine ausführliche Modellbeschreibung (Cubert und Fishwick 1997).

5.2.5 Synchronisation

Die Notwendigkeit von Synchronisationsmechanismen zur Wahrung der Kausalität bei verteilter Simulation wurde bereits in Abschnitt 4.1 deutlich. Offen bleibt die Frage, welche Synchronisationsverfahren innerhalb einer Simulationsinfrastruktur einsetzbar sein sollen.

Optimistische Verfahren verlangen die Speicherung von Zwischenzuständen, um Rollbacks durchführen zu können, falls Nachrichten gesendet wurden, die die Kausalität verletzen. Bei einigen Simulatoren, die beispielsweise intern Threads oder Koroutinen benutzen, kann es schwierig oder unmöglich sein, diese lokalen Prozesse nach einem Rollback wieder korrekt aufzubauen. Viele Simulatoren sehen deshalb überhaupt keine Rollbacks vor. Sollen diese Simulatoren trotzdem als Komponenten innerhalb eines Gesamtansatzes nutzbar sein?

Ein einfacher Ansatz für eine Simulationsinfrastruktur besteht darin, ein bestimmtes Verfahren mitsamt aller Schnittstellen und Steuerungsnachrichten vorzuschreiben (Miller u. a. 2001). Dieses eine Verfahren verspricht natürlich nicht für alle Simulationsmodelle die optimale Berechnungsgeschwindigkeit, ermöglicht aber die Interoperabilität zwischen den Modellkomponenten.

5.2.6 Modellspezifische Datentypen

Die Werkzeuge innerhalb einer Simulationsinfrastruktur müssen natürlich modellunabhängig arbeiten können, sonst sind für jedes Simulationsprojekt mit besonderen Anforderungen gleichzeitig Änderungen an diesen Werkzeugen notwendig. Die modellspezifischen Definitionen sind also strikt von den Spezifikationen der Simulationsinfrastruktur selbst zu trennen (Straßburger 2001). Hier bietet sich eine spezielle Beschreibungssprache an wie HLA-OMT oder CORBA-IDL.

Andererseits müssen diese modellspezifischen Definitionen innerhalb der Werkzeuge verwendbar sein, da sich die Modellierung mit geeigneten Definitionen vereinfacht. Dazu gehören nicht nur Konstanten und einfache Datentypen wie Zahlen oder Zeichenketten, sondern auch zusammengesetzte Typen, Felder oder Aufzählungen. Durch die Verwendung komplexer Typen vermeidet man, mehrere Aufrufe zur Übergabe zusammenhängender Werte einzusetzen bzw. eine modellspezifische Codierung in Zahlen oder Zeichenketten durchzuführen. Modellunabhängige Werkzeuge müssen sich also darauf einstellen, mit Definitionen umzugehen, die erst zur Laufzeit verfügbar werden, denn mit dem Zugriff auf ein bisher nicht geladenes Teilmodell können neue Typdefinitionen auftauchen.

5.2.7 Benutzerrollen und Benutzungsphasen

In Anlehnung an (Dorwarth u. a. 1997) und (Praehofer u. a. 2000) lassen sich verschiedene Benutzerrollen beim Einsatz von Modellbildung und Simulation unterscheiden.

1. Entwickler einer Simulationsinfrastruktur werden sich üblicherweise in Gremien treffen, um die grundlegende Architektur, notwendige Dienste und die Kommunikationsmechanismen festzulegen. Ihre Ergebnisse liegen u.a. in Form von Schnittstellenspezifikationen vor.

2. Die Entwickler einer Simulationsumgebung setzen die Simulationsinfrastruktur in konkrete Produkte um. Diese Produkte sind üblicherweise Rahmenwerke, mit denen sich Modellkomponenten entwickeln, Komponenten zu größeren Einheiten koppeln oder vollständige Modelle ausführen lassen.

3. Komponentenentwickler besitzen Kenntnisse in speziellen Anwendungsgebieten und erstellen mit Hilfe existierender Rahmenwerke Komponenten, die in diesen Anwendungsgebieten häufig benötigt werden. Weiterhin gehören in diese Benutzergruppe diejenigen Personen, die Komponenten dokumentieren und testen. Als Arbeitsergebnisse entstehen dann die Komponenten selbst, ihre Beschreibungen (ggf. mit speziellen Typdefinitionen) und Einträge in geeignete Verzeichnisse, um die Komponenten aufzufinden.

4. Modellentwickler benutzen Rahmenwerke, um innerhalb einer Simulationsstudie ausführbare Modelle zu erstellen. Sie suchen in Verzeichnissen nach existierenden Komponenten aus dem Anwendungsgebiet der Studie, konfigurieren die Komponenten und koppeln sie. Je nach Architektur der Simulationsinfrastruktur lassen sich die zusammengesetzten Modelle wiederum als Komponenten auffassen, oder sie stellen ausführbaren Code für ein spezielles Rahmenwerk dar.

5. Modellnutzer schließlich führen Simulationsexperimente mit vollständigen Modellen durch. Sie stellen Parameter ein, definieren die zu beobachtenden Laufzeitwerte und analysieren die Simulationsergebnisse. Die Präsentation ihrer Ergebnisse (etwa in Form von Diagrammen oder Animationen) dient als Grundlage weiterer Entscheidungen bzgl. des simulierten Realsystems.

Die Anforderungen der ersten vier Benutzergruppen wurden bereits in den vorherigen Abschnitten behandelt (Schnittstellen, Blackboxes, Tydefinitionen, Verzeichnisse). Aus Sicht der Modellnutzer fehlt noch eine wichtige Forderung, nämlich die Unterstützung der Experimentierphase. Hier sollten sich die Ein- und Ausgaben eines komplexen Modells hinsichtlich des typischen Vorgehens leicht unterscheiden lassen.

Die Experimentierphase verläuft dreischrittig (parametrisieren, berechnen, auswerten, Wittmann 1992) und enthält häufig Wiederholungen dieser Schrittfolge (Wittmann 1993). Ein Rahmenwerk kann einen Modellnutzer in dieser Phase wesentlich besser unterstützen, wenn es in die Lage versetzt wird, die Ein- und Ausgaben eines beliebigen Modells den einzelnen Experimentierschritten automatisch zuzuordnen. Deshalb ist bereits bei der Modellbeschreibung eine entsprechende Unterscheidung der möglichen Interaktionen notwendig.

5.2.8 Referenzmodelle

Die Einigung auf eine Simulationsinfrastruktur, d.h. auf eine spezifische Technik wie beispielsweise HLA, reicht noch nicht aus, um Simulationskomponenten effizient einzusetzen. Wenzel (2000) schlägt deshalb die Einführung von Referenzmodellen vor. Referenzmodelle erlauben eine systematische Aufbereitung eines modellierten Anwendungsgebiets unabhängig von existierenden Simulationswerkzeugen. Dort ist in Abschnitt 4 zu lesen:

> Um eine verteilte Nutzung und Kopplung der Modelle und Komponenten über mehrere Instanzen zu erreichen, muss bezüglich des betrachteten Anwendungsfeldes eine gemeinsame Terminologie und eine einheitliche Strukturierung vorliegen und verwendet werden.
>
> An dieser Stelle greifen die Referenzmodelle: Sie sind die Basis, um semantische Bezüge in einem verteilten Modell herzustellen; sie schaffen eine gemeinsame Begrifflichkeit und erlauben damit eine gezielte Austauschbarkeit und/oder Kopplung von Modellen; sie stellen die Basis effizienten Modellierens dar.

Eine Schichtenarchitektur, die Referenzmodelle enthält, beschreiben Zeigler und Sarjoughian (2000). Auf unterster Ebene steht die Basiskommunikation, auf die ein Kopplungsschema aufbaut. In den Referenzmodellen wird dann eine gemeinsame Semantik beschrieben, die explizit auch das Zeitverhalten (Synchronisation, diskret/kontinuierlich) definiert. Nach (Wenzel 2000) bleiben aber auch bei der Verwendung von Referenzmodellen noch Fragen hinsichtlich geeigneter Vorgehensmodelle zur Validierung und hinsichtlich möglicher Suchfunktionen offen.

Jenseits der technischen Unterstützung durch geeignete Schnittstellen bleibt also auch für die Entwickler von Simulationskomponenten noch einige Standardisierungsarbeit nach. Dieser Prozess kann zwar gefördert werden durch den Aufbau von Verzeichnissen, die Schemata zur Systemrepräsentation innerhalb bestimmter Anwendungsbereiche enthalten, bleibt aber außerhalb der Zielsetzung dieser Arbeit (vgl. Abschnitt 1.1).

5.3 Leistungsvergleich bestehender Ansätze

Nachdem im vorangegangenen Abschnitt die Aspekte für eine erfolgreiche Wiederverwendung diskutiert wurden, erfolgt nun eine zusammenfassende Beschreibung und Bewertung der komponentenorientierten Ansätze aus Kapitel 4 hinsichtlich ihrer Unterstützung der einzelnen Aspekte. Die Bewertungskriterien sind demnach Hersteller- und Sprachunabhängigkeit, Verwendung des Blackbox-Prinzips, Möglichkeiten zum entfernten Zugang, Synchronisationsmechanismen, erlaubte Datentypen und die Unterstützung unterschiedlicher Benutzungsphasen. Abschließend werden dann die Ansätze verglichen und der Stand der Technik auf dem Gebiet der verteilten, komponentenorientierten Simulation zusammengefasst.

HLA

Für verschiedene Programmiersprachen (Java, C++) standardisiert HLA die Schnittstellen zur Laufzeitumgebung direkt. Es existiert auch eine Schnittstellenbeschreibung in CORBA-IDL, aus der sich weitere Sprachabbildungen ableiten lassen (Lisp, COBOL, u.a.). Die Sprachunabhängigkeit ist für HLA also gegeben. Da ausschließlich Schnittstellen und deren Semantik festgelegt sind, können die Simulationskomponenten nur nach dem Blackbox-Prinzip sicher arbeiten.

Mit dem OMT stellt HLA eine Möglichkeit zur Verfügung, modellspezifische Datentypen zu definieren. Seit der Standardisierung durch IEEE (2000c) ist auch ein Codierungsschema für beliebige Datentypen vorhanden, dessen Verwendung noch optional ist.

Ein entfernter Aufbau von Federations ist im Prinzip möglich, indem Rechnername und TCP-Port einer RTI angegeben werden und dort die Aufrufe zur Anbindung lokaler Federates erfolgen. Einige RTI-Implementationen verlangen aber für die Konfiguration die Verwendung eines gemeinsamen Dateisystems, wodurch Komponenten außerhalb eines lokalen Netzwerks ausgeschlossen werden. Mit HLA können Nachrichten auch mit dem UDP-Broadcast-Mechanismus übertragen werden, der nur lokal eingebundene Rechnerknoten erreicht. Im militärischen Bereich existiert bereits ein Verzeichnis von Komponenten (www.msrr.dmso.mil). Dort finden sich die Kontaktadressen der Komponentenbetreiber. Direkte Referenzen auf Komponenten sind nicht möglich, und HLA sieht auch noch keinen technischen Rahmen zur Identifikation einzelner Komponenten vor.

Für die Synchronisation sind im Time-Management spezielle Schnittstellen definiert. Die Synchronisationsverfahren und deren Steuerungsnachrichten innerhalb einer RTI sind nicht exakt festgelegt, so dass jeder Hersteller sein eigenes Verfahren verwenden darf, was zu Inkompatibilitäten führen kann. Deshalb ist HLA herstellerabhängig. Es lässt sich zumindest festhalten, dass der Einsatz von optimistischen Verfahren nicht vorgesehen sein kann, da keine expliziten Rollback-Funktionen definiert wurden. Die Schnittstellen ermöglichen nur eine Synchronisation auf Federate-Ebene und sind für einige Anwendungsfälle zu grob granuliert (vgl. Abschnitt 4.1.3 und Abbildung 4.4).

Die Unterscheidung von Benutzungsphasen für den Experimentiervorgang unterstützt HLA nicht. Es existiert noch nicht einmal der Begriff des Simulationslaufs. Als Ersatz dient der Aufbau einer Federation, die über einen be-

stimmten Zeitraum bestehen bleibt, wobei ein Mechanismus zur Verbreitung von Start- und Ende-Benachrichtigungen der Federation fehlt. Bei der Deklaration von Attributen und Interaktionen wird auch nicht global festgelegt, ob es sich um Ein- oder Ausgabeoperationen handelt. Diese Unterscheidung erfolgt erst dynamisch durch die Anmeldung von Federates.

DEVS

DEVS kann man zunächst einmal als Formalismus ansehen, um die Zustandsübergänge in den einzelnen Simulationskomponenten zu beschreiben. Dieser Ansatz für sich ist sprachunabhängig. In der Umsetzung entstand neben sprachspezifischen Lösungen für Java oder C++ auch eine sprachunabhängige Beschreibung per CORBA-IDL, die im Prinzip die Umsetzung in verschiedene Zielsprachen erlaubt. Die nachfolgende Zusammenfassung bezieht sich auf diese DEVS/CORBA-Variante (Zeigler u. a. 1999b).

Da die Realisierung von DEVS/CORBA auf HLA-Basis erfolgen sollte, wird unnötigerweise eine Abhängigkeit vom Hersteller einer HLA-RTI eingeführt, die sich durch direkte Realisierung der Schnittstellen vermeiden ließe. Das Blackbox-Prinzip ist aufgrund der ausschließlichen Verwendung von Schnittstellenbeschreibungen gegeben.

Die Modellkomponenten sind als eigenständige Einheiten durch ein IDL-Interface deklariert. Man kann also direkt auf entfernte Einzelkomponenten zugreifen, wenn die Objektreferenz bekannt ist. Die strikte Kopplung unter DEVS ermittelt das gekoppelte Modellverhalten ausschließlich aus dem Einzelverhalten der beteiligten Komponenten, was den Aufbau von gekoppelten Modellen in manchen Fällen erschwert (vgl. Abschnitt 4.3.3). Der DEVS-Mechanismus enthält in seinem Kopplungsschema bereits Aussagen zur Synchronisation nebenläufiger Komponenten. Diese wurden auf der Basis von Ports (Abschnitt 4.3.2) in die DEVS/CORBA-Umsetzung übernommen.

Eine DEVS-Port-Beschreibung sieht nur drei Datentypen vor (`string`, `int`, `float`). Modellspezifische Typen sind hier nicht verwendbar. Innerhalb einer Port-Beschreibung besteht weiterhin keine Möglichkeit, Benutzungsphasen zu unterscheiden.

MOOSE

Das MOOSE-System stellt im Wesentlichen eine geschlossene Umgebung dar. Die Verwendung von Komponenten anderer Hersteller ist nicht direkt vorgesehen, aber über eine C++-Schnittstelle können Verknüpfungen mit selbst programmierten Komponenten erfolgen.

Es werden verschiedene Modellierungssprachen verwendet, von denen jede eine spezielle Ausführungsumgebungen benötigt. Als Sprachunabhängigkeit im Sinne allgemeiner Programmiersprachen kann man diesen Ansatz nicht verstehen.

Durch ein eigenes Verwaltungssystem erlaubt MOOSE sowohl den verteilten Zugang zu Simulationskomponenten als auch die gesteuerte parallele Abarbeitung von Simulationsmodellen. Die Synchronisation nebenläufiger Prozesse ist dabei überhaupt nicht vorgesehen.

In den Beschreibungssprachen für Modelle kann ein Benutzer neue Typen

deklarieren und sie vorhandenen Ein- oder Ausgabevariablen zuweisen. Die Variablen lassen aber keine Unterscheidung nach Benutzungsphasen zu.

MOBILE

Das Projekt MOBILE legte DCE als verteilte Ausführung fest. Die DCE-Spezifikation ist für sich herstellerunabhängig, verlangt jedoch C als Programmiersprache. Innerhalb dieser Umgebung ist der direkte verteilte Zugang gegeben.

Mit der Beschreibungssprache MSL sind modellspezifische Typdeklarationen möglich. Dabei lassen sich Ein- und Ausgabevariablen unterscheiden und nach verschiedenen Benutzungsphasen klassifizieren. Synchronisationsmechanismen waren nicht vorgesehen. Es wurden jedoch niemals Adapter für die Skriptsprache MSL realisiert, um den Einsatz in der DCE-Umgebung zu ermöglichen.

JavaBeans

Kopplungen nach dem JavaBeans-Muster sind von Natur aus sprachabhängig (Java). Die Spezifikationen der Sprache, der virtuellen Maschine und der JavaBeans-Kopplungsumgebung sind offen und somit herstellerunabhängig. Ein entfernter Zugang ist über den eingebauten RMI-Mechanismus realisierbar.

JavaBeans selbst sieht keine verteilten Synchronisationsmechanismen zur Kausalitätssicherung vor. Im Rahmen der Umgebung COSIMA (Oses 2002) wurde eine spezielle Abarbeitungsreihenfolge der Nachrichten festgelegt, mit der die Synchronisation zum Teil nachgebildet werden kann.

Die mächtigen Reflexionsmechanismen der Sprache Java machen es dem JavaBeans-Ansatz leicht, beliebige modellspezifische Datentypen zu verarbeiten. Einzelne Variablen können jedoch nicht direkt in Benutzungsphasen unterteilt werden. Lediglich der integrierte Konfigurationsmechanismus für Komponenten kann für die Einstellung von Simulationsparametern benutzt werden.

TENT

Die TENT-Umgebung fällt ein wenig aus dem Rahmen der üblichen komponentenorientierten Simulation, da dort der Komponentenbegriff anders verwendet wird. Komponenten dienen nicht zur Nachbildung komplexen Systemverhaltens, sondern zum Aufbau des Workflows im Rahmen einer Experimentierumgebung. Sie repräsentieren jeweils einzelne Arbeitsschritte zur Durchführung einer Simulationsstudie wie Parametrisierung, Steuerung der verteilten Modellberechnung, statistische Auswertung oder Visualisierung.

Durch den Einsatz von CORBA-Hüllen um die verwendeten Werkzeuge herum erlaubt TENT die hersteller- und sprachunabhängige Wiederverwendung nach dem Blackbox-Prinzip und den entfernten Zugriff auf die verwendeten Werkzeuge. Als Experimentierumgebung unterstützt das TENT-System natürlich sämtliche Benutzungsphasen unmittelbar. Da der Datenaustausch auf Dateibasis betrieben wird, ist eine feinere Strukturierung durch modellspezifische Datenstrukturen gar nicht vorgesehen.

CUDOs

Die CUDOs-Schnittstellen dienen im Wesentlichen zur Kopplung verschiedener Simulatoren des Herstellers ISI (Statemate, MATRIXx). Durch ein Rahmenwerk zur Erstellung von allgemeinen Adaptern ist es aber auch möglich, beliebige andere Werkzeuge einzubinden. Hier erlauben wiederum CORBA-Hüllen prinzipiell die hersteller- und sprachunabhängige Wiederverwendung nach dem Blackbox-Prinzip. Die Beschreibung von Zugängen zu Simulationskomponenten (Ports) unterscheidet nicht nach Benutzungsphasen, und als Datentypen lassen sich lediglich einige eingebaute Typen verwenden.

Einschränkungen beim Einsatz möglicher ORBs ergeben sich durch das Adapter-Rahmenwerk selbst, da hier ein herstellerabhängiger Mechanismus zur Bindung von Objektreferenzen eingesetzt wird (ORBIX `_bind`). Der direkte entfernte Zugang zu vorhandenen Simulationskomponenten wird dadurch erschwert, dass deren Beschreibungen als XML-Dokument im lokalen Dateisystem verfügbar sein müssen.

Stand der verteilten, komponentenorientierten Simulation

Tabelle 5.1 fasst die vorgefundenen Eigenschaften der in Kapitel 4 vorgestellten und in den vorhergehenden Unterabschnitten bewerteten Ansätze zusammen. Die Bewertungsaspekte ergeben sich unmittelbar aus Abschnitt 5.2. Positiv (+) markierte Felder deuten eine vorhandene technische Unterstützung eines Aspektes an, bei Negativeinträgen (-) fehlt diese Unterstützung. Hinter geklammerten Markierungen stehen besondere Annahmen, die in den entsprechenden Unterabschnitten kurz erläutert wurden.

Eigenschaft	HLA	DEVS	MOOSE	MOBILE	JavaBeans	TENT	CUDOs
herstellerunabhängig	−	(+)	−	+	+	+	(+)
sprachunabhängig	+	+	(−)	−	−	+	+
Blackbox-Prinzip	+	(+)	−	+	+	+	+
entfernter Zugang	(+)	+	+	+	+	+	(+)
Synchronisation	+	+	−	−	(−)	−	−
Datentypen	(+)	−	+	+	+	−	−
Benutzungsphasen	−	−	(−)	+	(−)	+	−

Tabelle 5.1: Eigenschaften verschiedener Ansätze

Die Herstellerabhängigkeit von HLA macht den Einsatz in einer heterogenen Umgebung schwierig. DEVS stellt durch die Komponentenbeschreibung mit Ports einen grundlegenden Kopplungsmechanismus bereit, der sich auch in den Systemen MOBILE und CUDOs wiederfindet. Zu einer umfassenden Unterstützung fehlen DEVS noch modellspezifische Datentypen und eine Aufteilung der Ports nach Benutzungsphasen. Von MOOSE als eigenständigem System konnte man kaum Unterstützung für den Einsatz in einer heterogenen Umgebung erwarten. Der MOBILE-Ansatz erfährt durch die Verwendung von DCE gute Unterstützung nur für die Programmiersprache C. Auch mit JavaBeans ist man mit Java auf eine Sprache festgelegt. Das TENT-System ist auf eine heterogene

Umgebung zugeschnitten, bedient jedoch nur Anwendungen für den Modellbenutzer, die ohne Synchronisation bezüglich der Simulationszeit auskommen. In CUDOs muss man auch auf Synchronisationsmechanismen, sowie auf modellspezifische Datentypen verzichten.

Der Inhalt der Tabelle 5.1 macht also deutlich, dass bisher kein vorgestellter Ansatz eine umfassende technische Unterstützung in einer hersteller- und sprachunabhängigen Umgebung bietet, um Simulationskomponenten wiederzuverwenden und komplexe Modelle aus einzelnen Komponenten zusammenzusetzen. Ein solcher Ansatz ist jedoch notwendig, um existierende Modellteile ohne großen systemtechnischen Aufwand zu koppeln.

6 CoSim: Ein CORBA-Ansatz für Simulationskomponenten

Das vorangegangene Kapitel zeigte einige Defizite bisheriger Ansätze für die verteilte, komponentenorientierte Simulation bei der Unterstützung wiederverwendbarer Modellkomponenten auf. Deshalb wird nun unter dem Namen *CoSim* einen neues Konzept vorgestellt, das die in Abschnitt 5.2 diskutierten Rahmenbedingungen erfüllt.

CoSim steht für *Component Based Distributed Simulation*. Dahinter verbirgt sich ein Satz von Schnittstellen zur Beschreibung und Kopplung von Modellkomponenten sowie eine Reihe von Diensten und Werkzeugen, um diese Schnittstellen auszufüllen. Durch die Auswahl einer geeigneten Kommunikationsinfrastruktur (CORBA) ermöglicht CoSim den hersteller- und sprachunabhängigen Zugang zu verteilten Objekten und setzt das Blackbox-Prinzip zur Wiederverwendung um. Die CORBA-Schnittstellen sind einerseits so allgemein gehalten, dass die Verwendung modellspezifischer Datentypen problemlos möglich ist, andererseits leisten diese Schnittstellen direkte Unterstützung für die in der verteilten Simulation wichtigen Synchronisationsmechanismen und für die Unterscheidung von Benutzungsphasen beim Durchführen von Simulationsexperimenten.

6.1 Kommunikationsinfrastruktur

Innerhalb der CoSim-Umgebung muss die Funktionalität der Komponenten per Netzwerk zugänglich sein. Dabei werden Anfragen nach dem Client/Server-Prinzip wie etwa entfernte Prozeduraufrufe notwendig. Als Grundlage soll hier die Protokollkombination TCP/IP dienen, die auf den meisten heutigen Rechensystemen vorhanden ist. Dazu wird die Kommunikation mittels der verbreiteten Techniken Sockets, DCE, RMI, DCOM, CORBA sowie WebServices betrachtet, deren Vor- und Nachteile gegenübergestellt und die Verfügbarkeit für verschiedene Plattformen (Windows, Unix, Mac) und Programmiersprachen ausgewertet. Auch mögliche Mehrwertdienste, die für die jeweiligen Techniken verfügbar sind, fliessen in die Bewertung ein.

6.1.1 Sockets

Die direkte Programmierung über Kommunikationsendpunkte der TCP/IP-Ebene (Sockets) ist auf jeder Plattform möglich. Auch stehen in nahezu jeder Programmiersprache entsprechende Bibliotheken für den Netzwerkzugriff zur Verfügung. Da Sockets lediglich die Übertragung binärer Datenströme erlauben, müssen sämtliche höheren Mechanismen wie Datencodierung, entfernte Prozeduraufrufe und Dienstdefinitionen von Anwendungsprogrammierern vollständig

in Eigenarbeit erstellt werden (Orfali und Harkey 1998, Kap. 10). Dieser immense Aufwand entsteht für jede einzubindende Programmiersprache. Bei der Verwendung von Sockets wäre es jedoch möglich, performante Mechanismen einzubauen, die speziell auf den Einsatz als Simulationsinfrastruktur zugeschnitten sind, wie beispielsweise verschiedene Synchronisationsmechanismen oder die Kompression von Datenströmen.

6.1.2 DCE

Das Distributed Computing Environment (DCE, OSF 1995) ist u.a. für die untersuchten Plattformen Windows, Unix und Macintosh implementiert. Es wird jedoch nur die Programmiersprache C direkt unterstützt. Neben der Realisierung von Remote Procedure Calls (RPC) stellt DCE eine Reihe von Mehrwertdiensten zur Verfügung. So gibt es ein ausgeprägtes Sicherheitskonzept auf der Basis von Authentisierungs- und Verschlüsselungsmechanismen (Kerberos), einen Verzeichnisdienst und ein verteiltes Dateisystem. Auch die Nebenläufigkeit spielt innerhalb eines speziellen Thread-Konzeptes eine große Rolle. Gerade für den Simulationseinsatz könnten die Mechanismen zur Synchronisation der Realzeit hilfreich sein.

6.1.3 RMI

In der Programmiersprache Java ist mit dem Konzept von Remote Method Invocations (RMI) bereits eine Möglichkeit zur verteilten Kommunikation vorhanden (Orfali und Harkey 1998, Kap. 13). Java ist für fast alle Plattformen erhältlich. Mit Hilfe eines URL-basierten Namensschemas (RMI-Registry) lassen sich entfernte Objekte lokalisieren. Die Datencodierung erfolgt automatisch über den Java-internen Serialisierungsmechanismus. Da die Programmiersprache auch die Untersuchung von Klassen über eingebaute Reflexions-Schnittstellen ermöglicht, sind keine externen Typ- und Dienstdefinitionen mehr notwendig. Stattdessen können diese Informationen direkt aus den Klassendefinitionen entnommen werden. Die Verwendung von RMI-Objekten in anderen Programmiersprachen ist jedoch nicht vorgesehen.

6.1.4 CORBA

Mit der Common Object Request Broker Architecture (CORBA) steht ein herstellerunabhängiger Mechanismus zur verteilten Kommunikation bereit (OMG 1999). CORBA existiert in einer Vielzahl von Programmiersprachen- und Plattform-Kombinationen, wobei die Interoperabilität zwischen verschiedenen Realisierungen gewährleistet ist. Die Kernfunktionalität übernimmt ein Object Request Broker (ORB), der Anfragen an Objekte entgegennimmt, weiterleitet und die notwendigen Übertragungsprotokolle (z.B. IIOP) beherrscht. Er stellt keinen zentralen Dienst dar, sondern ein spezielles Transportsystem (Objektbus) und wird auf jeder beteiligten Rechnerumgebung benötigt.

CORBA setzt eine Schnittstellenbeschreibungssprache (IDL) ein, um Datentypen und Zugriffsfunktionen zu deklarieren. Definierte Schnittstellen sind mit Precompiler-Werkzeugen für die benötigte Zielsprache in Quellcode zu übersetzen (vgl. Abbildung 6.1). Die entsprechenden Adapter auf der Client-Seite

(*Stubs*) bzw. auf der Server-Seite (*Skeletons*) sorgen für typsichere Interaktionen und regeln dabei die Datencodierung sowie den Aufruf benutzerdefinierter Funktionen.

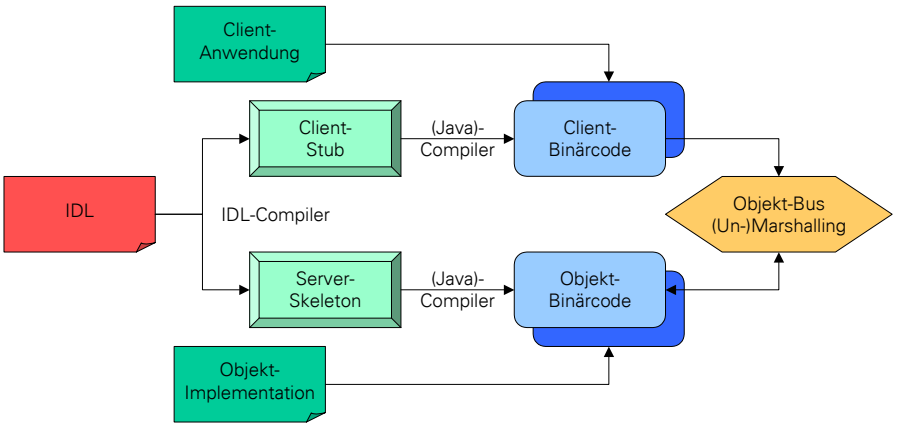


Abbildung 6.1: CORBA Entwicklungsschritte

Mit CORBA sind verschiedene Mehrwertdienste definiert, und für die meisten Dienste existieren inzwischen kommerzielle Realisierungen. So gibt es einen Namensdienst zum Auffinden verteilter Objekte (Naming), einen attributbasierten Suchdienst für verteilte Objekte (Trading), ein Schnittstellenverzeichnis zur Erstellung typunabhängiger Anwendungen auf Introspektionsbasis (Interface Repository) und einen Nachrichtendienst zur asynchronen Kommunikation (Notification). Für bestimmte Anwendungsdomänen lassen sich spezielle Schnittstellenmengen entwerfen. In der Domäne Manufacturing sind beispielsweise die HLA-IDL-Definitionen als Distributed Simulation Facility untergebracht. Diese Definitionen erlauben jedoch nur einen herstellerabhängigen Einsatz, wie in Abschnitt 4.2.6 gezeigt wurde.

6.1.5 WebServices

Die WebServices-Technik verbindet gängige Übertragungsprotokolle wie HTTP oder SMTP/MIME mit einer Datencodierung per XML (Ferris und Farrell 2003). Da die Dienstzugangspunkte (TCP-Ports) für diese Übertragungsprotokolle in der Regel keinen besonderen systemseitigen Kontrollen unterliegen, lassen sich so Restriktionen umgehen, die beispielsweise durch Firewalls auferlegt werden. Ein spezielles Protokoll zur Versendung von Objektnachrichten (SOAP, Simple Object Access Protocol) ermöglicht auch entfernte Methodenaufrufe. Die Typdefinition der übermittelten Objekte erfolgt mit einer XML-basierten Beschreibungssprache (WSDL, WebServices Description Language). Objektanfragen lassen sich so dynamisch zur Laufzeit erstellen und auswerten. Der Verzeichnisdienst UDDI (Universal Description, Discovery and Integration) enthält Informationen über vorhandene WebServices-Objekte in URL-Form.

WebServices sind an keine Plattform oder Programmiersprache gebunden, da lediglich standardisierte, offene Protokolle verwendet werden. Auch an der Anbindung weiterer Infrastrukturen wie beispielsweise EJBs oder rein nachrichtenbasierten Mechanismen (Message Oriented Middleware, MOM) wird inzwi-

schen gearbeitet (Kreger 2003). Die meisten Werkzeuge jedoch existieren bisher für die Programmiersprache Java, die gerade bei der Verwendung von XML eine starke Unterstützung erfährt.

6.1.6 DCOM und .NET

Die Firma Microsoft definiert auf ihren Windows-Plattformen das Distributed Component Object Model (DCOM), das ursprünglich entstanden ist aus der Weiterentwicklung von OLE (Object Linking & Embedding) für die Kommunikation zwischen Anwendungen der Office-Suite. Mit COM+ (ehemals Microsoft Transaction Server) steht eine Reihe von Diensten (Transaktionen, Sicherheit) für diese Komponentenumgebung zur Verfügung, die seit der Einführung von Windows XP auch als WebServices gekapselt werden können (Löwy 2001). DCOM-Unterstützung gibt es für verschiedene objektorientierte Programmiersprachen (C++, Java, Python, u.a.). Dabei ist ein entfernter Methodenaufruf möglich, der aus DCE hervorgegangen ist, und ein explizites Typverzeichnis wurde realisiert. Neue Dienste und Typen sind über eine Beschreibungssprache (MIDL) zu definieren.

Eine aktuelle Microsoft-Entwicklung stellt .NET dar. Diese Technik definiert eine eigene Laufzeitumgebung (Common Language Runtime) und ermöglicht so verschiedenen Programmiersprachen den geregelten Zugriff auf entfernte Objekte und Betriebssystem-Dienste (Thai und Lam 2002). Die Schnittstellen zu dieser Laufzeitumgebung wurden durch die europäische Organisation ECMA als CLI (Common Language Interface) standardisiert, so dass im Prinzip auch Realisierungen auf anderen Plattformen möglich wären. Es bleibt abzuwarten, ob dieser Ansatz sich wirklich durchsetzt und Unterstützung für andere Plattformen geleistet wird.

Die COM+ Laufzeitdienste sind Teil von .NET. Weiterhin wurde mit .NET der DCE-ähnliche Teil zur Kommunikation durch WebServices (HTTP, XML, SOAP) ersetzt. Das Microsoft-Komponentenmodell realisiert Komponenten jeweils auf der Ebene von Binärcodes, wobei für .NET noch eine Zwischenrepräsentation (Bytecode) eingeführt wurde, die zur Laufzeit in Maschinencode übersetzt wird.

6.1.7 Bewertung der Techniken

Die größten Vorzüge für die Anwendungsentwicklung bietet CORBA, weil es eine standardisierte Technik ist, die von verschiedenen Herstellern auf einer großen Zahl von Plattformen und Programmiersprachen unterstützt wird (vgl. Tabelle 6.1). WebServices versprechen eine gleichermaßen offene Umgebung, wurden aber zu Beginn dieser Arbeit noch nicht ausreichend durch Werkzeuge unterstützt.

Aufgrund der Flexibilität von CORBA ist im Bereich der Performanz bei entfernten Methodenaufrufen mit Einbußen zu rechnen. Allein dadurch, dass für den Einsatz beliebiger Datenstrukturen die Typinformation vollständig zu übertragen ist, und dass die eingesetzten Objektreferenzen durch viele Zeichen codiert werden müssen, resultieren lange Nachrichtenblöcke. Da es jedoch beim CoSim-Ansatz in erster Linie um die Wiederverwendbarkeit von Simulationskomponenten geht, in deren Entwicklung viel Aufwand geflossen ist, kann auf

Technik	plattformunabhängig	sprachunabhängig	herstellerunabhängig	Werkzeugunterstützung	Performanz	Bemerkungen
Sockets	+	+	+	−	+	Eigenarbeit (je Sprache)
DCE	+	0	+	+	0	C, Threads, Verzeichnisdienst
RMI	+	−	0	−	0	Java, URL Naming, Reflection
CORBA	+	+	+	+	−	div. Dienste
WebServices	+	+	+	−	−	wenige Werkzeuge
DCOM / .NET	−	+	−	+	0	Microsoft Plattformen

Tabelle 6.1: Vor- und Nachteile betrachteter Kommunikationstechniken
+: gute, −: schlechte, 0: mäßige Unterstützung

eine hohe Durchführungsgeschwindigkeit von Simulationsläufen meist verzichtet werden. Deshalb wurde CORBA als Realisierungsgrundlage für den CoSim-Ansatz ausgewählt. Eigene Tests mit verschiedenen ORB-Produkten (VisiBroker, OpenORB, ORBacus, LispWorks ORB) bestätigten die Reife und Flexibilität dieser Technik. Das in Abschnitt 2.3.2 diskutierte Komponentenmodell für CORBA ging jedoch nicht mit in die Realisierung ein, da zu Beginn der Umsetzung von Werkzeugen noch keine entsprechenden Application Server verfügbar waren.

6.2 CoSim im Überblick

6.2.1 Architektur

Die CoSim-Architektur teilt sich auf in drei horizontale Schichten und drei vertikale Anwendungsgebiete. Dadurch können Benutzungsschnittstelle und Persistenzmechanismen unabhängig von der Anwendungslogik verändert werden Diese Drei-Schichten-Architektur hat sich für den Zugriff auf entfernte Ressourcen bewährt (Orfali und Harkey 1998, S. 27). In Abbildung 6.2 sind die Präsentations-, Anwendungs- und Datenschicht jeweils umrahmt. Die Symbole der Anwendungsgebiete Komponentenentwicklung, Modellkopplung und Experimentdurchführung kennzeichnet eine jeweils gleiche Füllfarbe.

Die Werkzeuge der entsprechenden Anwendungsgebiete unterstützen die Benutzer in ihrer Rolle als Komponentenhersteller, Modellentwickler bzw. Modellnutzer. Die Entwicklerrollen bzgl. Architektur und Rahmenwerken werden durch die Ergebnisse dieser Arbeit abgedeckt. Die erarbeiteten Schnittstellendefinitionen befinden sich im Anhang A. Die Werkzeuge sowie ihre Rahmenwerke sind in den Kapiteln 7 bzw. 8 beschrieben und finden sich im Quellcode beim Open-Source-Provider *SourceForge* unter `http://cosim.sf.net` wieder.

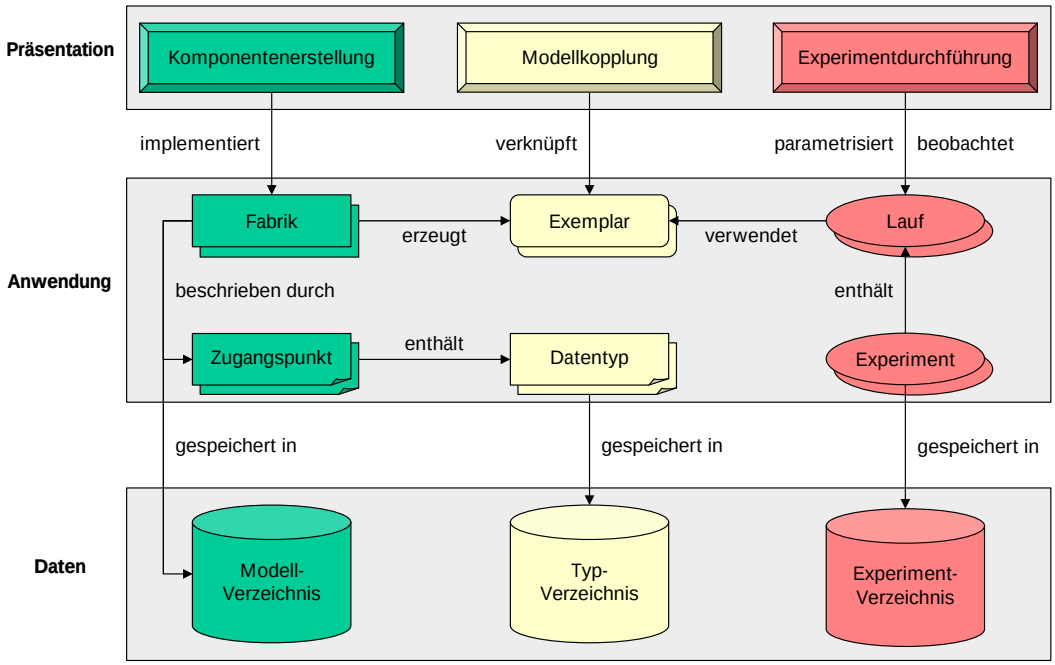


Abbildung 6.2: CoSim-Architektur

6.2.2 Schnittstellen

Eine CoSim-Komponente wird als CORBA-Objektreferenz zur Verfügung gestellt. Somit entfällt für den Modellbenutzer ein lokaler Installationsvorgang je Komponente. Mit dieser Objektreferenz wird ein Mechanismus zur Erzeugung einzelner Modellexemplare verfügbar (Modellfabrik). Innerhalb eines Modellexemplars lassen sich verschiedene Zugangspunkte unterscheiden, die mit den DEVS-Ports aus Abschnitt 4.3.2 vergleichbar sind. Jeder Zugangspunkt stellt wieder eine Objektreferenz dar. Hier wird die Identität einer Objektreferenz im CORBA-Objektmodell also direkt ausgenutzt, um die einzelnen Zugangspunkte zu unterscheiden. Eine Modellgröße im Sinne von Abschnitt 2.1.2 kann auch durch verschiedene Zugangspunkte nutzbar gemacht werden, etwa um sowohl lesend als auch schreibend zugreifen zu können.

Zu jeder CoSim-Komponente gehört eine Beschreibung, die die Eigenschaften verfügbarer Zugangspunkte angibt. Wichtige Elemente dieser Beschreibung sind die Einteilung von Zugangspunkten in Nutzungsphasen sowie die Angaben zum Datentyp und zum Kommunikationsverfahren je Zugangspunkt. Der CoSim-Entwurf sieht nämlich verschiedene Varianten vor, um die Datenübertragung zwischen Komponenten durchzuführen. Abschnitt 6.5 wird diese Varianten ausführlicher betrachten.

Für die Synchronisation von verteilten Berechnungsprozessen innerhalb eines Gesamtmodells stellt CoSim einen einfachen Mechanismus bereit, um ein konservatives Verfahren unter Verwendung von Lookahead-Werten umzusetzen. Die notwendigen Null-Nachrichten werden in Datenstrukturen untergebracht, die auch zum Transport von Nutzdaten zusammen mit beliebigen Kontextwerten dienen (Abschnitt 6.6), wie beispielsweise Zeitstempel oder Raumkoordinaten.

6.2.3 Werkzeuge

Vorhandene CoSim-Komponenten werden üblicherweise in Verzeichnisse eingetragen, auf die ein Modellentwickler zugreifen kann, um Teilmodelle zu komplexen Gesamtmodellen zu koppeln. Verschiedene grafische Benutzungsschnittstellen dazu stellt Abschnitt 7.2 vor. Abschnitt 6.4 erläutert einen Verzeichniszugang, der speziell für die Ablage von Komponentenbeschreibungen entwickelt wurde.

Da die Schnittstelle, die eine CoSim-Komponente anbietet, recht umfangreich ist, sind Werkzeuge sinnvoll, um vorhandene Teilmodelle mit wenig Aufwand als CoSim konforme Komponenten zu verkleiden. Der CoSim Adapter Generator (Abschnitt 7.1) leistet diese Verkleidung für Teilmodelle, die in der Programmiersprache Java realisiert sind.

Die Kopplung von Teilmodellen kann mit dem CoSim Component Connector (Abschnitt 7.3) erfolgen. Dieses Werkzeug besteht aus zwei Teilen. Der erste Teil erlaubt eine grafische Darstellung der verwendeten Teilmodelle und der notwendigen Datenflüsse durch eine Symbolsprache aus Knoten und Kanten. Der zweite Teil enthält einen Interpreter für die spezifizierten Graphen, erzeugt daraus CoSim-Komponenten sowie die zugehörigen Beschreibungen und stößt die Datenübertragungen an. Der CoSim Component Connector sorgt auch dafür, dass verschiedene Kommunikationsmechanismen zwischen den Zugangspunkten einzelner Modellexemplare überbrückt werden.

Die Experimentierumgebung (Abschnitt 7.4) dient zur Durchführung von Simulationsläufen mit CoSim-Komponenten. Sie unterstützt alle Phasen der Experimentdurchführung (Parametrisierung, Laufzeitberechnung, Ausgabeanalyse) und erlaubt die Definition komplexer Experimente, die aus mehreren Simulationsläufen bestehen können.

Ein Ansatz, der die Art der benutzbaren Modellkomponenten nicht einschränkt, muss auch mit den modellspezifischen Datentypen umgehen können, die zum Realisierungszeitpunkt sämtlicher Werkzeuge noch nicht existierten. Eine entsprechende Softwareumgebung wird als *generisch* bezeichnet, da sie von der tatsächlichen Art der Datenstrukturen abstrahieren kann. Sämtliche CoSim-Werkzeuge sind generisch und erlauben die Verwendung aller Daten- und Objekttypen, die mit der Schnittstellenbeschreibungssprache CORBA-IDL definierbar sind.

Da den CoSim-Werkzeugen die Typbeschreibungen aller verwendeten Modellgrößen innerhalb einer Modellkomponente durch ein standardisiertes Typverzeichnis zugänglich sind, können sie bei der Kopplung zu komplexen Modellen eine Typverträglichkeitsprüfung durchführen. Auch bei der Parametrisierung eines ausführbaren Modells im Rahmen eines Simulationsexperiments hilft die Typerkennung, um nur syntaktisch korrekte Datenwerte zu akzeptieren.

6.3 Phasen der Modellnutzung

Der CoSim-Ansatz definiert eine Menge von Nutzungsphasen und entsprechenden Phasenübergängen, die die kontrollierte Durchführung von vielen Simulationsläufen innerhalb eines komplexen Simulationsexperiments erleichtern. Betrachtet man die Durchführung eines Simulationslaufs aus der Benutzersicht, so lassen sich aus den Möglichkeiten der Benutzerintervention verschiedene Nut-

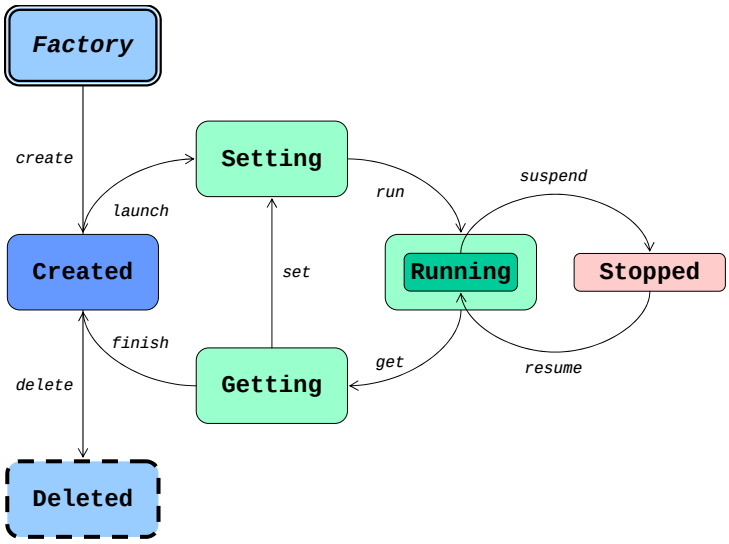


Abbildung 6.3: Nutzungsphasen und Phasenübergänge

zungsphasen ableiten.

6.3.1 Lebenszyklus von Modellexemplaren

Der Lebenszyklus eines Modellexemplars beginnt mit der Erzeugung durch eine dauerhaft existierende Modellfabrik. In der Phase `Created` sind zunächst nur mögliche Voreinstellungen für Parameter ablesbar. Anschließend setzt der Experimentator noch fehlende Parameter (Phase `Setting`). Nachdem das Modellexemplar alle Parameter auf korrekte Initialisierung geprüft hat, erfolgt die schrittweise Durchführung der Verhaltensberechnung im Modell (`Running`). Hierbei lässt sich die Abarbeitung unterbrechen (`Stopped`) und wieder fortsetzen, um verschiedene Modellzustände eingehender zu untersuchen.

Nachdem das Ende der Modellberechnung erreicht ist, werden die Simulationsergebnisse ermittelt (`Getting`). Diese Ergebnisse kann das Modell selbst liefern, oder sie werden durch geeignete externe Berechnungsverfahren erzeugt, beispielsweise durch eingebaute statistische Verfahren einer Experimentierumgebung. Bei externen Verfahren können nur die eingesetzten Parameter und die Folge von beobachtbaren Modellzuständen herangezogen werden. Das Modell selbst kann hingegen intern Variablen verwalten, die dem Benutzer und der Experimentierumgebung nicht im Modellzustand angezeigt werden. Nach Auswertung der Ergebnisgrößen können weitere Durchläufe mit demselben Modellexemplar durchgeführt werden, und ein neuer Zyklus (`Setting`, `Running`, `Getting`) beginnt. Wird ein Modellexemplar nicht mehr benötigt, so erreicht der Lebenszyklus die Endphase `Deleted`, und die benutzen Ressourcen können wieder freigegeben werden.

6.3.2 Phasenübergänge

Die einzelnen Phasen und die vorgesehenen Übergänge veranschaulicht Abbildung 6.3. Kommt es zu Fehlern bei der Verwendung des Modellexemplars, etwa

durch fehlende Ressourcen oder durch Arithmetikfehler, so ist automatisch der Rückfall in die Phase `Created` vorgesehen. Die entsprechenden Phasenübergänge sind hier jedoch der Übersichtlichkeit halber nicht dargestellt.

Die Phasenübergänge werden durch Methodenaufrufe eingeleitet, die im Modellexemplar für die nötigen Vorbereitungen der nächsten Nutzungsphase sorgen. Diese Aufrufe kehren zurück, sobald die Vorbereitungen erfolgt sind. Lediglich die Methode zur Durchführung des eigentlichen Simulationslaufs in der Phase `Running` wartet, bis die Modellberechnungen vollständig abgeschlossen sind. Sämtliche Phasenübergänge werden zusätzlich protokolliert und über einen Nachrichtenkanal zur Verfügung gestellt (*Typed* Notification Service: OMG 2000b, 2001a). So können auch andere Objekte über die Nutzungsphasen eines Modellexemplars informiert werden, für die die Rückkehr der phasenändernden Methodenaufrufe nicht beobachtbar ist. Dies ist beispielsweise dann notwendig, wenn allgemeine Protokollmechanismen über Nutzungsstatistiken realisiert werden sollen, oder um neue Modellexemplare dynamisch zu entdecken und zu koppeln, ähnlich wie beim Object Management in HLA für interaktive Trainingsszenarien (Abschnitt 4.2.4). Die entsprechenden Rückrufmethoden erläutert die Datei `cosim_event.idl` in Anhang A.

Die zentralen Phasen `Setting`, `Running` und `Getting` repräsentieren gleichzeitig die drei Hauptgruppen, in die sich die Zugangspunkte einer Modellkomponente einordnen lassen. Je nach Experimentierphase können also unterschiedliche Zugangspunkte aktiv sein. Damit werden Parameter, Laufzeitgrößen und Ergebnisgrößen unterscheidbar.

6.4 Komponentenbeschreibungen

6.4.1 Zugangspunkte

Eine CoSim-Komponente wird repräsentiert durch ein CORBA-Objekt, das die Funktionalität einer Modellfabrik enthält (vgl. Übersichtsabbildung 6.2). Jede Komponente selbst existiert also nur in einer einzigen Ausprägung und implementiert die Schnittstelle `Model_Factory` (Datei `cosim_factory.idl` in Anhang A). Eine Komponente kann jedoch mehrere unabhängige Modellexemplare erzeugen, die einer dokumentierten Verhaltensbeschreibung folgen. Dabei kommuniziert ein Modellexemplar mit seiner Außenwelt ausschließlich über fest definierte Zugangspunkte. Durch einen Zugangspunkt wird eine Modellgröße im Inneren des Modellexemplars unterschieden und nach außen verfügbar gemacht. Es können aber auch mehrere Zugangspunkte eingesetzt werden, um auf dieselbe Modellgröße innerhalb eines Modellexemplars zuzugreifen, etwa ein Zugangspunkt zum Ablesen eines vorhandenen Wertes und einer zum Setzen eines neuen Wertes.

Jeder Zugangspunkt ist selbst wieder ein CORBA-Objekt (Schnittstelle `Lookup_Connection`, Datei `cosim_lookup.idl`). Alle Zugangspunkte einer Komponente sind in einer Modellbeschreibung zusammengefasst (Schnittstelle `Lookup_Class`). Die Modellbeschreibung ist der Komponente direkt zugeordnet und existiert auch nur in einer einzigen Ausprägung. Da Modellbeschreibungen ebenfalls als CORBA-Objekte verfügbar sind, könnten sie im Prinzip für die Beschreibung verschiedener Komponenten mit gleichen Zugriffsmöglichkeiten, aber unterschiedlichem Verhalten eingesetzt werden. Die Implementierung der

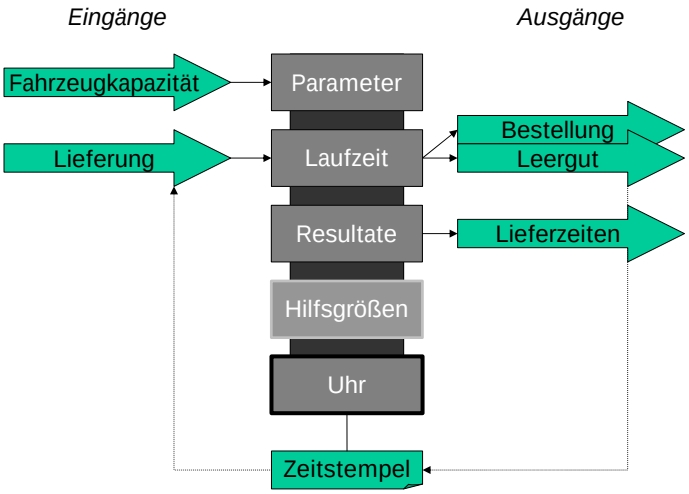


Abbildung 6.4: Zugangspunkte

vorhandenen Werkzeuge sieht diese Möglichkeit jedoch nicht vor. Die Werkzeuge erstellen mit jeder Modellfabrik immer auch eine neue Modellbeschreibung.

Durch die Modellbeschreibung werden die Zugangspunkte eingeteilt hinsichtlich ihrer Verfügbarkeit im Rahmen von Simulationsexperimenten (vgl. Abbildung 6.4). Aus den drei zentralen Nutzungsphasen des Abschnitts 6.3 lassen sich zunächst die drei Kategorien Parametergröße, Laufzeitgröße und Ergebnisgröße ableiten. In der Regel erlauben Parameter nur den schreibenden Zugriff und Ergebnisgrößen nur den lesenden Zugriff. Laufzeitgrößen kommen in beiden Varianten vor. Im Falle gekoppelter Modelle liefern die Ausgangsgrößen eines Modells in der Laufzeitphase die Datenwerte für die Eingangsgrößen eines anderen Modells.

Eine weitere Kategorie ist für die Markierung von Hilfsgrößen vorgesehen. Diese Größen sind nicht direkt zugreifbar, sondern lediglich in Verbindung mit anderen Zugangspunkten. Sie dienen als Schlüsselwörter bei der Angabe von Kontextwerten eines Hauptwertes. Abschnitt 6.6 wird die Diskussion über die Verwendung von Hilfsgrößen wieder aufnehmen.

Die letzte Kategorie aus Abbildung 6.4 erlaubt den Zugriff auf die Simulationszeit. Diese Modellgröße ist üblicherweise in jedem Simulationsmodell anzutreffen und hat eine spezielle Bedeutung. Gerade für die Durchführung von Simulationsexperimenten sind beispielsweise diejenigen Zugangspunkte explizit auszuweisen, anhand derer sich die aktuelle Simulationszeit ablesen bzw. der Simulationsfortschritt interaktiv steuern lässt. Selbst wenn mehrere Zugänge zur Simulationsuhr existieren, so ist doch immer dieselbe Modellgröße eines Modellexemplars betroffen. Die Simulationszeit steht nur in den Phasen `Running` oder `Stopped` zur Verfügung.

Zu einem einzelnen Zugangspunkt gehört die Angabe des Datentyps, dem die übertragenen Datenwerte entsprechen müssen. Weiterhin können Gültigkeitsbereiche und ein Initialwert vorhanden sein. Neben der Zugriffsrichtung (lesend oder schreibend) beschreiben zwei weitere Attribute eines Zugangspunktes die Art der Kommunikation mit der Außenwelt. Zum einen wird festgelegt, welche Seite eine Datenübertragung anstößt, ob das Modellexemplar also selbst aktiv

wird, oder nur passiv Daten entgegennimmt bzw. auf Anfragen hin liefert. Zum anderen gibt der Zugangspunkt an, ob die Datenübertragung synchron oder asynchron erfolgt. Diese letzte Eigenschaft wird im Rahmen der verwendbaren Kommunikationsarten in Abschnitt 6.5 behandelt.

Die vollständige Beschreibung einer CoSim-Komponente sieht zusätzlich zur formalen syntaktischen Beschreibung der Zugangspunkte noch zwei beschreibende Dokumente vor. Das erste Dokument soll die Bedeutung der einzelnen Zugangspunkte erläutern. Das zweite Dokument stellt das Verhalten der Modellexemplare dar. Beide Dokumente sind als URL anzugeben, um sie mit gängigen WWW-Suchmaschinen erfassen zu können. Enthalten die Dokumente Verweise auf die zugehörige Modellfabrik im `corbaname:` Format (vgl. Abschnitt 6.7.1), so ist ein unmittelbarer Zugang zur CoSim-Komponente möglich, wenn eine Experimentier- oder Kopplungsumgebung die Behandlung übernimmt.

6.4.2 Objekthüllen

Mit dem Einsatz von CORBA geht die Gesetzmäßigkeit einher, dass zusammengesetzte IDL-Datenstrukturen (wie z.B. `structure` oder `sequence`) immer als Kopie und Objekte (`interface`) immer als Referenz übertragen werden. In manchen Fällen sollen aber auch zusammengesetzte Datenstrukturen referenziert und wiederverwendet werden, etwa bei der Angabe von Initialwerten für Modellgrößen. Deshalb definiert CoSim Schnittstellen zur Erzeugung von Objekthüllen für zusammengesetzte Datenstrukturen, mit deren Hilfe sich dieses Problem umgehen lässt. Eine entsprechende Implementation speichert diese Objekthüllen in einem Verzeichnis (`Lookup_Repository`, Datei `cosim_lookup.idl`, Anhang A).

Die Zugangspunkte und Komponentenbeschreibungen aus dem vorangegangenen Abschnitt basieren auf einer zusammengesetzten Datenstruktur und sind nur deshalb als CORBA-Objekte einsetzbar, weil sie durch den Lookup-Mechanismus eine Objekthülle erhalten haben. Damit besitzen sie automatisch eine Objektidentität. Diese Identität ist nützlich, um bei Werteanfragen nach Modellgrößen zu unterscheiden. Man vermeidet damit die Vergabe konstruierter Unterscheidungsmerkmale wie z.B. die `AttributeHandles` in HLA (Abschnitt 4.2.4).

Dieser Lookup-Verzeichnisdienst ist in der Lage, beliebige zusammengesetzte IDL-Datenstrukturen persistent abzulegen. Mit dem *Persistent State Service* (OMG 2002d) gibt es zwar eine Spezifikation für die Zusammenarbeit zwischen Datenbanken und IDL-Datenstrukturen, jedoch wird dabei für jede verwendete Struktur eine Persistenzdeklaration benötigt, damit die zugehörigen Werkzeuge die angebundene Datenbank richtig ansteuern. CoSim kann diese Deklarationen jedoch nicht vorab bereitstellen, denn neue Modelle können auch neue Datenstrukturen mit sich bringen. Es ist daher nicht sinnvoll, vom Modellentwickler mit jeder neuen Datenstruktur auch eine Persistenzdeklaration zu fordern, die erst in einem anderen, nicht simulationsspezifischen Werkzeug zur Geltung kommt.

Um eine Objekthülle zu erzeugen, nimmt das Verzeichnis einen strukturierten Namen (`Lookup_Name`), den zu erzeugenden Datentyp sowie die zu verpackenden Daten entgegen. Die erzeugte Objekthülle kann später den Namen, die verpackten Daten und deren Datentyp durch geeignete Methoden reproduzieren (vgl. Abbildung 6.5). Der strukturierte Name besteht aus dem Verzeichnis eines Namensdienstes und einem eindeutigen Identifikator innerhalb dieses Verzeich-

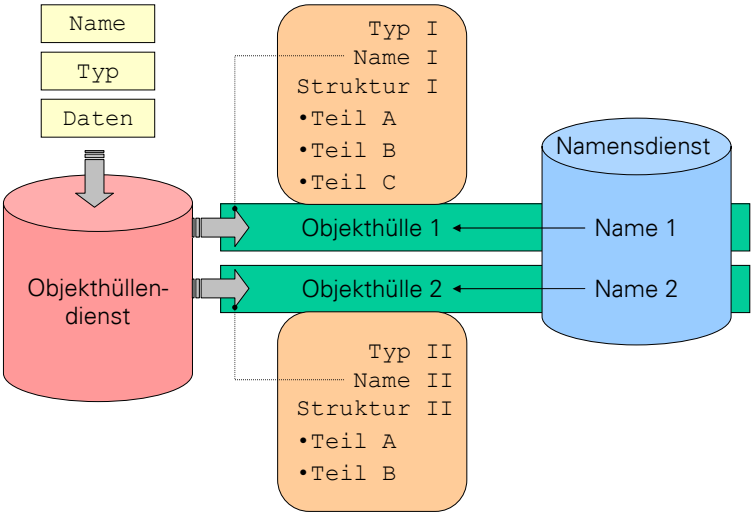


Abbildung 6.5: Erzeugung von Objekthüllen

nisses (vgl. Abschnitt 6.7). Der Identifikator wird benutzt, um die Objekthülle im Verzeichnis zu unterscheiden. Somit sind alle erzeugten Objekthüllen benannte Objekte (`Lookup_Object`), die von sich aus darüber Auskunft geben können, unter welchem strukturierten Namen sie aufzufinden sind. Diese Eigenschaft ist nützlich, um die Objekthüllen leichter wiederzuverwenden, da der strukturierte Name für Menschen erheblich einfacher lesbar ist als die Zahlencodierung einer Objektreferenz (IOR).

6.5 Kommunikationsarten

Um die Kommunikation mit einem Modellexemplar zu beschreiben, müssen je Zugangspunkt drei Eigenschaften festgelegt werden. Die *Kommunikationsrichtung* bestimmt, ob das Modellexemplar Datenwerte empfängt (`IMPORT`) oder versendet (`EXPORT`). Es gibt also immer einen Sender von Daten und einen Empfänger.

Welcher der beiden Kommunikationspartner die Versendung von Daten anstößt, regelt die Eigenschaft *Aktivität*. Ein aktiver Zugangspunkt sorgt selbst für die Lieferung bzw. Anfrage von Datenwerten, ein passiver wartet auf eingehende Lieferungen bzw. Anfragen. Generell erfordert eine Kommunikation, in der der Empfänger aktiv ist, immer zwei Schritte bis zur Ankunft einer Nachricht (Anfrage und Antwort).

Durch die *Art der Kopplung* wird bestimmt, auf welchem Wege Nachrichten zu übertragen sind. Bei der engen Kopplung, oder auch synchronen Datenübertragung, erfolgt die Kommunikation direkt mit dem Kommunikationspartner. Für jede Übertragung erhält dann der aktive Partner unmittelbar eine Quittung vom passiven Partner. Die lose Kopplung oder asynchrone Datenübertragung entkoppelt die Modellkomponente und ihre Nutzer durch einen Nachrichtenkanal. Quittungen erfolgen nur zwischen dem Nachrichtenkanal und dem Empfänger bzw. Sender. Die beteiligten Kommunikationspartner erfahren die

Identitäten auf der Gegenseite nicht.

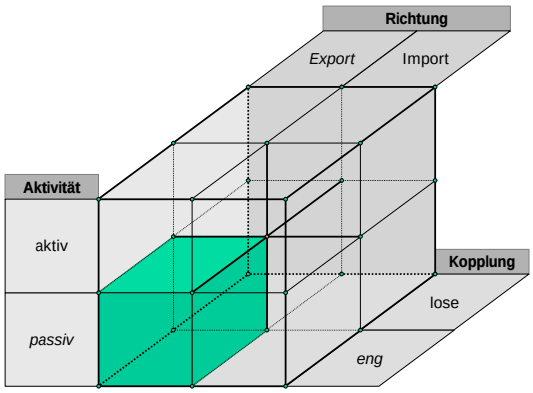


Abbildung 6.6: Orthogonale Eigenschaften der Kommunikation

Abbildung 6.6 verdeutlicht noch einmal die Unabhängigkeit dieser drei Eigenschaften. Es sind also acht Fälle unterscheidbar, wie die Kommunikation über einen Zugangspunkt erfolgen kann. Die einzelnen Aktivitäten eines Zugangspunktes werden in den Abbildungen 6.7 und 6.8 für jeden dieser Fälle dargestellt.

6.5.1 Enge Kopplung

Für den Datenaustausch durch enge Kopplung melden sich die passiven Kommunikationspartner direkt beim aktiven Partner an und erklären so ihre Bereitschaft zum Empfang oder zum Senden von Daten. Ein Zugangspunkt kann dabei auch mehrere Kommunikationspartner bedienen. Der aktive Partner kennt also seine Empfänger bzw. Sender und verwendet die Rückrufschnittstellen `Guided_Value_Receiver` bzw. `Guided_Value_Supplier` (Datei `cosim_datatransfer.idl` in Anhang A), um Daten zu übertragen.

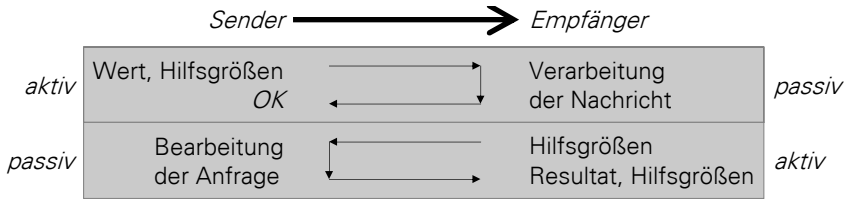


Abbildung 6.7: Synchrone Datenübertragung

Besondere Aufmerksamkeit verdient die enge Kopplung mit aktivem Empfänger und passivem Sender (der untere Fall in Abbildung 6.7). Dies ist die einzige Variante, mit der ein parametrisierter entfernter Methodenaufruf nachgebildet werden kann. Da der Empfänger aktiv ist, kann er bei einer Anfrage nach Daten Parameterwerte für diese Anfrage spezifizieren. Abschnitt 6.6.1 wird beschreiben, wie diese Parameterwerte zu codieren sind. Bei der losen Kopplung über einen Nachrichtenkanal sieht die Spezifikation für diese Anfragen keine Parameterwerte vor.

6.5.2 Lose Kopplung

Für den Datenaustausch durch lose Kopplung melden sich alle Kommunikationspartner bei einem Nachrichtenkanal an. Der Zugangspunkt stellt diesen Nachrichtenkanal bereit und entledigt sich damit der Aufgabe, selbst Anmeldungen entgegenzunehmen und zu verwalten. Zu einem Zeitpunkt dürfen natürlich mehrere Sender und Empfänger gleichzeitig an demselben Kanal angemeldet sein. Der Nachrichtenkanal kann auch dazu dienen, Anfragen bzw. Antworten zeitlich zu bündeln und somit für einen besseren Nachrichtendurchsatz zu sorgen.

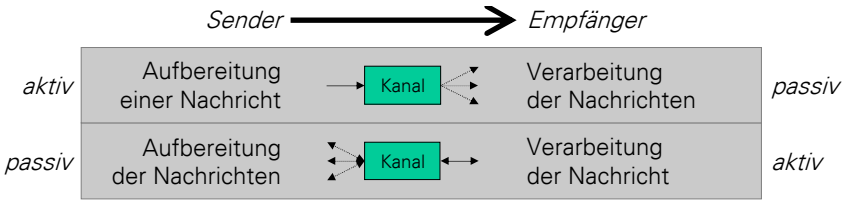


Abbildung 6.8: Asynchrone Datenübertragung

In den CORBA-Spezifikationen (OMG 2000b, 2001a) unterscheidet man die beiden Kommunikationsmodelle `Push` (Abbildung 6.8 oben) und `Pull` (Abbildung 6.8 unten). Dabei können an einem einzigen Kanal Sender und Empfänger ein beliebiges Kommunikationsmodell verfolgen. Der Kanal sorgt dann ggf. für die notwendige Zwischenspeicherung bzw. für regelmäßige Nachfragen verfügbarer Werte (Polling). Die beteiligten Objekte stellt Abbildung 6.9 dar. Die zwischengeschalteten Administrationsobjekte sind dabei für die Einstellung von *Quality of Service* Eigenschaften zuständig wie beispielsweise temporäre Speicherung von Nachrichten, Anzahl erlaubter Sender bzw. Empfänger u.a. Die Kommunikation zwischen den Objekten auf der Client-Seite (außerhalb des Rahmens) und den Dienstobjekten des Nachrichtenkanals (innerhalb des Rahmens) verläuft immer noch mit Hilfe quittierter entfernter Methodenaufrufe (Crane 1997). Die eigentliche lose Kopplung wird erst innerhalb des Kanals selbst vollzogen.

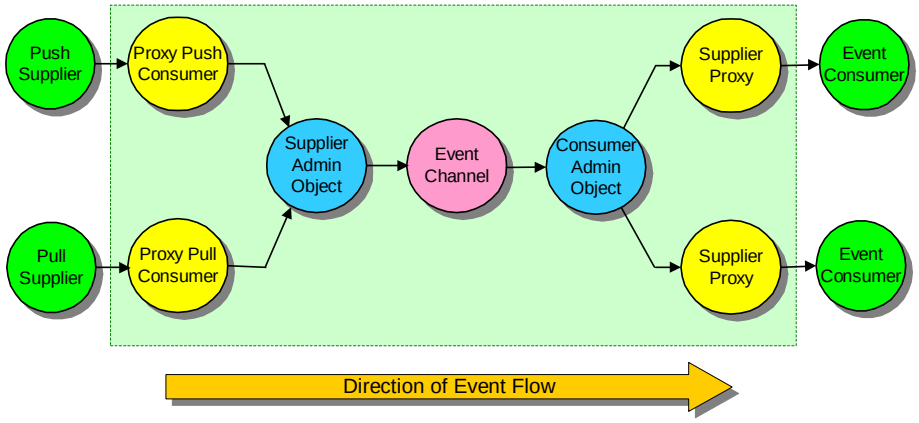


Abbildung 6.9: Event Channel, aus (PrismTech 2002, S. 12)

Wichtig bei der Verwendung eines Nachrichtenkanals ist die rechtzeitige An-

meldung der Empfänger. Es werden nämlich keine Nachrichten an einen Empfänger ausgeliefert, die an den Kanal gesendet wurden, bevor der Empfänger tatsächlich verbunden war. Deshalb sorgen sämtliche CoSim-Werkzeuge dafür, dass die Nachrichtenkanäle so früh wie möglich erzeugt werden, damit sich auch potentielle Empfänger so früh wie möglich anmelden können. Die Anmeldung muss also vor dem Wechsel in diejenige Nutzungsphase abgeschlossen sein, die dem entsprechenden Zugangspunkt zugeordnet ist. Steht ein Verlassen dieser Phase an, so wird der Nachrichtenkanal geschlossen und durch einen neuen Kanal ersetzt. So lässt sich verhindern, dass Nachrichten noch nach dem Phasenübergang bei Empfängern ankommen.

6.5.3 Nachrichtenfilterung

Auf die Daten, die über einen Nachrichtenkanal gesendet werden, lassen sich attributbasierte Filtertechniken anwenden. Diese Filter untersuchen die Inhalte gesendeter Nachrichten, im Gegensatz zu den deklarativen HLA-Filtern (Abschnitt 4.2.7). Um Nachrichtenfilter einzusetzen, müssen die Nachrichten speziell codiert werden, da diese Technik nur einen bestimmten Teil einer Nachricht tatsächlich untersucht (*filterable body*: OMG 2000b, Abschn. 2.2). Dieser Teil besteht aus Name/Wert-Paaren.

Eine geeignete Codierung muss natürlich auch mit modellspezifischen Datentypen umgehen können. Hier bietet sich ein Format an, das an das Speicherformat beliebiger per CORBA-IDL definierbarer Datenwerte in Abschnitt 8.1.2 angelehnt ist. Anstatt jedoch serialisierbare Java-Objekte wie dort zu erzeugen, muss diese Datenstruktur in einzelne Name/Wert-Paare aufgebrochen werden. Da zusammengesetzte IDL-Datentypen in ihrer Typbeschreibung ebenfalls Name/Wert-Paare benutzen, können sie in diesem Fall als Grundlage dienen, um z.B. ähnlich der Java Punktnotation tiefe Schachtelungen abzubilden.

In der aktuellen CoSim-Realisierung wurde diese Art der Codierung für die lose Kopplung jedoch aus Zeitgründen nicht umgesetzt. Stattdessen sind sämtliche Nachrichten in der üblichen CORBA-Any-Form codiert.

6.6 Kontextwerte

Kontextwerte werden bei einer Datenübertragung immer dann angegeben, wenn ein einzelner Wert nicht ausreicht, um ein Simulationsereignis vollständig zu interpretieren. Beispielsweise können Länge und Breite als Hilfsgrößen für Geokoordinaten bei der ortsgenauen Angabe eines Datenwertes auftreten. Hin und wieder werden auch Toleranzgrenzen angegeben, falls sich eine Modellgröße nicht exakt messen lässt. Am häufigsten treten Simulationszeiten auf, d.h. die übergebenen Hauptwerte besitzen einen expliziten Zeitbezug, der durch die Hilfsgröße angegeben ist.

Damit ein Modellentwickler nicht für jeden kontextabhängigen Zugangspunkt einen neuen Datentyp formulieren muss, verwendet CoSim bei der Datenübertragung eine allgemeine Datenstruktur (`Guided_Value`, Datei `cosim_basics.idl`), die neben einem Hauptwert auch beliebige Kontextwerte aufnehmen kann. Zu jedem Kontextwert ist ein Zugangspunkt als Ursprung dieses Wertes anzugeben. Der vollständige Kontext einer Datenübertragung besteht also aus einer Liste von Schlüssel/Wert-Paaren, in der die Zugangspunkte

als Schlüssel auftreten.

Für einige Schlüssel ergibt die direkte Abfrage des entsprechenden Zugangspunktes jedoch keinen Sinn, etwa bei einfachen Indizes für Datenfelder. Diese Zugangspunkte treten nur als Schlüssel auf. Für sie ist die Kategorie der Hilfsgrößen in der Komponentenbeschreibung gedacht.

6.6.1 Datenstrukturen

Um beliebige Datentypen in der CoSim-Umgebung benutzen zu können, werden sämtliche Haupt- und Kontextwerte in CORBA-Any-Datenstrukturen verpackt. Diese Datenstruktur verbindet einen codierten Datenstrom mit zugehörigen Typinformationen. Die Art der Codierung ist Bestandteil der CORBA-Spezifikation (Common Data Representation, CDR: OMG 1999, Abschn. 15.3). Typunabhängige Anwendungen lesen also zunächst die Typinformation aus und wählen erst in Abhängigkeit vom ermittelten Typ die entsprechende Decodierungsfunktion aus. Die notwendigen Codierungs- und Decodierungsfunktionen stellt die CORBA-Infrastruktur schon bereit. Grunddatentypen besitzen eigene Funktionen, und für zusammengesetzte Typen erzeugen IDL-Compiler den entsprechenden Code.

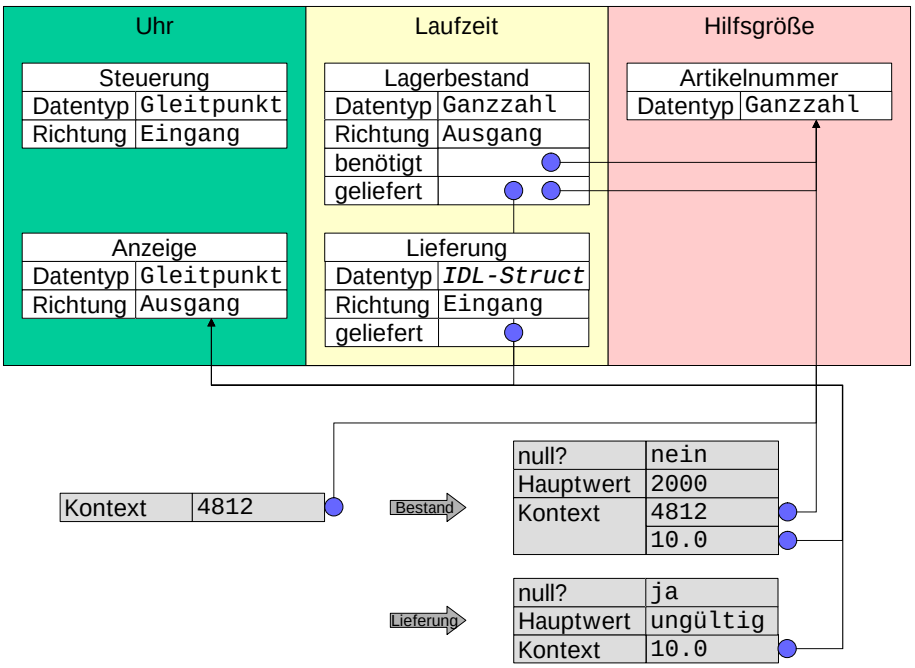


Abbildung 6.10: Datenstrukturen für Kontextwerte

Abbildung 6.10 stellt ein Beispiel für die verwendeten Datenstrukturen im unteren Teil dar. Der obere Teil gibt die Beschreibung einer Lagerhaltungskomponente an. Deutlich sind hier die Verweise auf Zugangspunkte zu erkennen. Der Zugangspunkt „Lagerbestand“ deklariert einen einzigen benötigten Eingabekontext „Artikelnummer“. Die zugehörige Anfrage ist also parametrisiert und wird nach Abschnitt 6.5 im synchronen Verfahren durchgeführt, wobei das Modell-

exemplar in der passiven Rolle auftritt. Das Modellexemplar antwortet auf eine Anfrage mit dem Hauptwert des Lagerbestandes und zwei Kontextwerten (Artikelnummer und aktuelle Simulationszeit), da in der Modellbeschreibung gerade diese beiden Schlüssel im Feld „geliefert“ deklariert wurden. Als Schlüssel in der Kontextliste dürfen Zugangspunkte aus jeder Kategorie deklariert werden, hier also auch die aktuelle Simulationszeit. Die Artikelnummer tritt ausschließlich als Kontextwert auf und ist als Hilfsgröße klassifiziert.

6.6.2 Synchronisation

Das Datenformat enthält weiterhin eine Markierung, ob die transportierten Inhalte eine Null-Nachricht darstellen. Eine solche Nachricht ist in Abbildung 6.10 ganz unten für den Zugangspunkt „Lieferung“ angeführt. Der Hauptwert dieser Nachricht kann ignoriert werden, lediglich der übertragene Kontextwert für die Simulationszeit wird von Bedeutung sein.

Null-Nachrichten sind die Basis für das konservative Synchronisationsverfahren nach (Chandry und Misra 1979) und geben die Lookahead-Werte an (vgl. Abschnitt 4.1.3). CoSim legt dieses Verfahren auf der Basis von Zugangspunkten fest (wie im DEVS-Ansatz mit Ports: Zeigler u. a. 2000, S. 269). Damit ist die Interoperabilität der eingesetzten Komponenten sichergestellt. Ein Ansatz, der verschiedene Verfahren zulässt, müsste zunächst einmal sämtliche Steuerungsnachrichten festlegen, wie etwa die Null-Nachrichten oder Barriere-Nachrichten. Anschließend werden entweder die Komponenten verpflichtet, auch alle festgelegten Verfahren zu unterstützen, oder es findet eine Verhandlung zwischen den Komponenten statt, um sich auf ein bestimmtes Verfahren zu einigen (Tu 2000). Da diese Verhandlungen auch scheitern können, und die Unterstützung mehrerer Synchronisationsverfahren die Komponentenerstellung wesentlich komplizierter macht, wird hier lediglich ein einziges Verfahren verwendet. Es kommt auch nur ein konservatives Verfahren in Frage, da die für optimistische Verfahren notwendige Rollback-Funktionalität selten von den vorhandenen Simulationssystemen angeboten wird (vgl. Abschnitt 5.2.5).

CoSim verlangt von jedem Modellexemplar, je Zugangspunkt nur Nachrichten mit aufsteigenden Zeitstempeln zu versenden, vorausgesetzt der Zugangspunkt deklariert eine Uhrzeit als gelieferte Kontextgröße. Das Synchronisationsverfahren wird also dezentral von den einzelnen Modellexemplaren durchgeführt. Sobald ein Modellexemplar eine Null-Nachricht oder eine gewöhnliche Nachricht mit Zeitstempel empfängt, wird es die anstehenden Ereignisse und weitere mögliche Null-Nachrichten berechnen und versenden. Falls jeder Zyklus von Abhängigkeiten zwischen den Zugangspunkten die Simulationszeit voranschreiten lässt, werden keine Verklemmungen durch dieses Synchronisationsverfahren auftreten können. Das Kopplungswerkzeug aus Abschnitt 7.3 ermittelt mögliche Zyklen und gibt ggf. Warnungen aus.

6.7 Modellsuche

Um Modellkomponenten innerhalb einer Experimentierumgebung oder eines Kopplungswerkzeugs auch tatsächlich nutzen zu können, werden Mechanismen zur Suche nach vorhandenen Modellfabriken benötigt. Mit den CORBA-Diensten Naming Service (OMG 2002c) und Trading Service (OMG 2000c)

existieren bereits zwei standardisierte Verfahren, um Objektreferenzen in Verzeichnissen aufzuführen.

Die formale Beschreibung der Modellkomponenten durch eine Menge von Zugangspunkten bietet einen weiteren Ansatzpunkt. So lassen sich gezielt Konkurrenzmodelle bzw. komplementäre Modelle einordnen. Diese Modelle beziehen sich auf ein gemeinsames Grundsystem und besitzen identische äußere Schnittstellen, also dieselben Zugangspunkte, unterscheiden sich jedoch in den zu Grunde liegenden Modellannahmen bzw. im Detaillierungsgrad. Ein Fabrikverzeichnis stellt diese Suchmöglichkeit nach Komponentenbeschreibungen kurz vor.

6.7.1 Naming

CORBA-Objekte werden eindeutig identifiziert durch eine interoperable Objektreferenz (IOR). Die IOR lässt sich als lange Zahlenreihe darstellen und wird in dieser Form auch von einem Object Request Broker erkannt. Da diese Zahlenreihe für den Menschen nur schlecht zu lesen oder einzugeben ist, werden Objektreferenzen innerhalb eines Namensdienstes verzeichnet. Eine Objektreferenz lässt sich also von diesem Dienst anhand ihres Namens auffinden, und ein leichter zu merkender Name kann anstelle der IOR verwendet werden. Gleichzeitig bietet der Namensdienst auch die Möglichkeit, Mengen von Objektreferenzen zu strukturieren.

Der Namensdienst verwaltet zwei verschiedene Arten von Einträgen, ähnlich wie ein Dateisystem. Für ein Dateisystem sind Dateien die untergeordneten Nutzungseinheiten, für CORBA sind es die Objektreferenzen. Ordner sind die Strukturierungseinheiten eines Dateisystems. Im Sprachgebrauch des CORBA Naming Service heißen diese Einheiten *Kontexte*. Kontexte lassen sich ineinander schachteln und enthalten *Bindungen*, also Zuordnungen von Namen zu Objekten oder wiederum zu Kontexten.

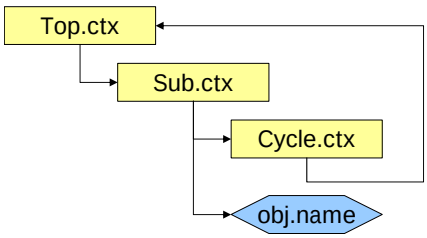


Abbildung 6.11: Strukturen im Namensdienst

Die Strukturen, die durch Verschachtelung von Kontexten erzeugt werden können, lassen sich am besten durch gerichtete Graphen beschreiben (vgl. Abbildung 6.11). Zyklen sind in diesen Graphen auch möglich, und ein zentraler Startknoten ist nicht festgelegt. Im Unterschied zu einem Dateisystem hat der CORBA Naming Service also keine ausgezeichnete Wurzel. Beim Start einer Anwendung dient üblicherweise ein bestimmter Kontext als Ausgangspunkt zur Suche nach benannten Objekten.

Für die Angabe von Objektreferenzen wurde ein spezielles URL-Schema entworfen. Die Objektreferenz `corbaname:iiop:my.host.net:4812/NameService#Top.ctx/Sub.ctx/obj.name` deutet durch das Trennzeichen `#` ein zweischrittiges Vorgehen bei der Benutzung eines Namensdienstes an. Im ersten Schritt muss

der Namensdienst selbst identifiziert werden. Dazu wird im obigen Beispiel der Rechner mit dem (DNS-)Namen `my.host.net` auf TCP-Port `4812` über das Protokoll IIOP kontaktiert und dort die Objektreferenz `NameService` angefordert. Diese Objektreferenz repräsentiert den Startkontext für alle nachfolgenden Anfragen nach Unterverzeichnissen und Objektnamen im zweiten Schritt. Hier soll also in der Kontextstruktur zunächst zum Kontext `Top.ctx` und dann zum Kontext `Sub.ctx` gewechselt werden. Im letzten Kontext ist dann der Objektname `obj.name` zur gesuchten Objektreferenz aufzulösen. Kontext- und Objektnamen bestehen immer aus zwei Teilen, die als `id` und `kind` bezeichnet werden. Der `kind`-Teil ähnelt dabei den Dateiextensionen.

Die Strukturierung durch Kontexte lässt sich nun verwenden, um Modellkomponenten beispielsweise nach Anwendungsgebieten zu klassifizieren. Weiterhin verwenden die CoSim-Werkzeuge zur Erstellung und Kopplung von Modellkomponenten den Namensdienst und gruppieren die erzeugten Komponentenbeschreibungen in eigenen Kontexten. Ganz besonders eng arbeitet der Objekthüllendienst (Abschnitt 6.4.2) mit dem Namensdienst zusammen, da jede Objekthülle automatisch einen Namen zugewiesen bekommt. Die grafische Nutzungsschnittstelle für den Objekthüllendienst verwendet das vorgestellte URL-Format, um Objektreferenzen lesbar darzustellen.

6.7.2 Trading

Um einen Namensdienst für die Komponentensuche zu benutzen, muss schon ein Hinweis in Form eines Namens gegeben sein, der eine konkrete Komponente oder einen Kontext identifiziert, unter dem Komponenten eingetragen sind. Liegt kein Name vor, kann der CORBA Trading Service helfen. Dies ist ein attributbasierter Vermittlungsdienst, der Objektreferenzen anhand ihres IDL-Schnittstellen-Typs und weiterer beschreibender Merkmale findet.

In die Nutzung des Vermittlungsdienstes sind drei Teilnehmer verwickelt. Der *Importer* sucht ein vorhandenes Objekt, der *Exporter* stellt ein Objekt und die beschreibenden Attribute bereit, und der *Trader* vermittelt zwischen den ersten beiden Parteien (Abbildung 6.12). Hat der Importer sein Gesuch an den Trader übergeben, gleicht der Trader dieses Gesuch mit den vorher von verschiedenen Exportern erstellten Angeboten ab und liefert eine Liste passender Angebote zurück. Der Importer entnimmt dieser Liste einen geeigneten Exporter und führt auf dem angebotenen Objekt die gewünschten Interaktionen aus. Bei der Vermittlung können durch Vernetzung mehrerer Trader auch organisatorische Grenzen (Sicherheit, Administration) überbrückt werden (Müller u. a. 1996).

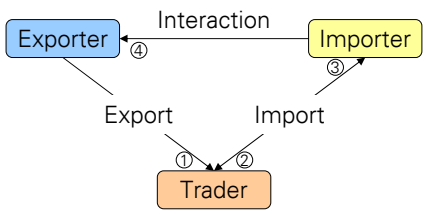


Abbildung 6.12: Teilnehmer der Dienstevermittlung

Die Form der Angebote und Gesuche muss vor der tatsächlichen Vermittlung

als *Diensttyp* definiert sein. Der Diensttyp legt den IDL-Schnittstellen-Typ der angebotenen bzw. gesuchten Objekte und die Namen sowie die Datentypen beschreibender Attribute fest. Eine Anfragesprache (Constraint Language) erlaubt dann die Formulierung von Bedingungen bzgl. der deklarierten Attribute.

Die Notwendigkeit eines Diensttyps stellt nicht nur Anwendungen vor Probleme, die bisher unbekannte IDL-Schnittstellen-Typen anbieten wollen (Merz u. a. 1994), sondern erschwert auch den Einsatz in der CoSim-Architektur, wo der IDL-Schnittstellen-Typ durch die Suche nach Modellfabriken bereits feststeht. Als Attribute bieten sich zunächst Texte für die Verhaltensbeschreibung und für die Bedeutung der Schnittstellen an. Auch eine Beschreibung der Zugangspunkte selbst sollte aufgenommen werden. Da jedoch die Zahl der Zugangspunkte je nach Modellfabrik variiert, müsste jede Modellfabrik ihren eigenen Diensttyp definieren, um alle Zugangspunkte in entsprechenden Attributen zu codieren. Zudem ist die Anfragesprache wenig ausdrucksmächtig. Für jeden Attributausdruck muss der Attributname bekannt sein, so dass die Beschreibung eines bestimmten Zugangspunktes nicht an ein eigenes Attribut gebunden werden darf. Die Prädikate der Anfragesprache erlauben auch nur einfache Vergleichsoperationen auf der Basis von numerischen Datentypen und eine Suche nach Textteilen.

Bei der Suche nach Kopplungspartnern einer bereits vorhandenen Komponente wären jedoch Prädikate sinnvoll, die eine Subtyp-Prüfung ermöglichen (Kontra- und Kovarianzregeln: Griffel 1998, S. 67), (Cardelli 1991, Abschn. 6.4). Dabei gehen in die Prüfung eines Funktionsaufrufs sowohl die Argumente als auch der Rückgabetyp ein. Die Mehrzahl der CoSim-Zugangspunkte repräsentiert zwar nur parameterlose Lese- und Schreiboperationen, doch eng gekoppelte, passiver Sender können auch parametrisierte Funktionen abbilden. Ein umfangreiches Typkonzept mitsamt der Realisierung eines Typmanagers stellt (Griffel 2001) vor. Da dort Komponenten nicht nur strukturell, sondern auch funktional beschrieben werden, muss ein global einheitliches Verständnis der vorgenommenen Abstraktion vorausgesetzt werden. Das gemeinsame Verständnis wird durch eine Domänenanalyse gewonnen und führt zu Beschreibungen, die den Referenzmodellen aus Abschnitt 5.2.8 entsprechen. Dieser Ansatz geht jedoch über die Zielsetzung dieser Arbeit hinaus.

Der CORBA Trading Service bietet also zunächst nur die Grundfunktionen für eine Komponentenvermittlung. Man kann die Suche auf Modellfabriken einschränken und in Beschreibungen nach Texten suchen. Für die gezielte Suche nach vorhandenen Zugangspunkten mit bestimmten Eigenschaften ist dieser Dienst nicht geeignet.

6.7.3 Fabrikverzeichnisse

Komponentenbeschreibungen sind selbst CORBA-Objekte und lassen sich wiederverwenden (vgl. Abschnitt 6.4.1). Eine Komponentenbeschreibung kann demnach von verschiedenen Modellfabriken genutzt werden. Ein Verzeichnis, das diese Abhängigkeiten enthält, liefert dann die Umkehrrelation zwischen Modellfabriken und ihren Komponentenbeschreibungen.

Dieses Fabrikverzeichnis wird über die Schnittstelle `Factory_Repository` (Datei `cosim_factory.idl`) als Dienst angeboten. Der einfache Suchmechanismus wertet die Objektreferenzen der Komponentenbeschreibungen aus und prüft sie auf Äquivalenz mit Hilfe des CORBA-Operators `is_equivalent`. Ein Abgleich

ähnlicher Komponentenbeschreibungen, die sich nur in den Namen oder in der Reihenfolge von Zugangspunkten unterscheiden, ist mit diesem einfachen Verfahren nicht möglich.

6.8 Fazit des CoSim-Ansatzes

Der vorgestellte CoSim-Ansatz verwendet CORBA für die Kommunikation mit verteilten Objekten. Dadurch kann ein hersteller- und sprachunabhängiger Zugang zu den Simulationskomponenten über die in IDL beschriebenen Schnittstellen nach dem Blackbox-Prinzip gewährleistet werden. Zur Datenübertragung setzt CoSim die CORBA-Hülle `Any` ein, in die sich sämtliche Datenwerte verpacken lassen, die einem in IDL definierbaren Datentyp angehören. Die standardisierte Codierung aller Datenwerte erlaubt somit auch die Verwendung modellspezifischer Datentypen.

Modellkomponenten liefern eine Beschreibung ihrer Zugangspunkte, die nach Nutzungsphasen klassifiziert werden. Diese Klassifikation liefert konkrete Hinweise für den Einsatz der Komponente im Rahmen einer Experimentierumgebung. CoSim unterstützt die Synchronisation bzgl. der Simulationszeit durch die Verbindung von Hauptwerten und Kontextwerten bei jeder Datenübertragung über die Zugangspunkte.

Damit erfüllt CoSim die Rahmenbedingungen aus Abschnitt 5.2. Zusätzlich ermöglicht der Ansatz verschiedene Varianten der Kommunikation hinsichtlich Richtung, Aktivität und Kopplungsart. Die Speicherung von Modellbeschreibungen und -fabriken in persistenten Verzeichnissen vereinfacht dann den Zugang zu Modellkomponenten durch die Modellentwickler.

Verhältnis zu anderen Ansätzen

Der CoSim-Ansatz entspricht dem DEVS-Ansatz bzgl. der Komponentenbeschreibung durch Zugangspunkte (Ports) und bietet eine natürliche Erweiterung für beliebige Führungsgrößen durch Kontextwerte. DEVS hingegen modelliert nur eine Führungsgröße (die Simulationszeit) explizit durch eine spezielle Zeitfortschrittsfunktion je Komponente. Auch das feingranulare Synchronisationsverfahren, das Lookahead-Werte auf der Basis von Zugangspunkten zulässt, wurde für DEVS schon vorgeschlagen.

Der Einsatz modellspezifischer Datentypen wurde vorrangig durch HLA und die zugehörigen OMT-Deklarationen vorangetrieben. Die Umsetzung in CoSim ergibt sich automatisch durch den Einsatz von IDL als Beschreibungssprache für neue Datentypen und durch das mit CORBA festgelegte Codierungsschema.

Für die Unterstützung von Experimentierszenarien teilte das Projekt MOBILE bereits Zugangspunkte in verschiedene Nutzungsphasen ein. Dieses Konzept nimmt CoSim wieder auf und erweitert die Klassifikation um den Zugang zur Simulationsuhr und zu Hilfsgrößen.

Eng verbunden mit dem CORBA-Komponentenmodell CCM ist die Idee, verschiedene Kommunikationsmechanismen zu unterstützen. Bei den vorgesehenen Kombinationen lassen sich Ein- und Ausgabeattribute, enge und lose Kopplung sowie aktive und passive Zugänge unterscheiden. CoSim trennt diese drei unabhängigen Eigenschaften und erlaubt alle acht möglichen Kombinationen.

Man kann CoSim abschließend als einen neuen Ansatz verstehen, der viele Eigenschaften anderer Ansätze kombiniert. Durch die hersteller- und sprachunabhängige Kommunikationsinfrastruktur wird so die technische Grundlage für einen flexiblen Einsatz im Anwendungsgebiet Simulation gelegt.

7 Konzeption von Werkzeugen

Um zu zeigen, dass der im Kapitel 6 vorgestellte CoSim-Ansatz auch modellunabhängig umsetzbar ist, folgt nun die konzeptuelle Beschreibung verschiedener Werkzeuge. Dabei wird nur die grundsätzliche Arbeitsweise jedes Werkzeugs betrachtet. Einige Details zur Realisierung finden sich in Kapitel 8.

Die Werkzeuge decken mit der Unterstützung für Komponentenentwickler, Modellentwickler und Modellnutzer bereits ein großes Spektrum der Rollen aus Abschnitt 5.2.7 ab. Da die Infrastruktur mit dem CoSim-Ansatz selbst bereits feststeht, und die vier im Folgenden beschriebenen Werkzeuge eine konkrete Umsetzung der Umgebung darstellen, sind auch die verbliebenen Rollen der Werkzeughersteller und Entwickler der Simulationsinfrastruktur bedient.

Der CoSim-Adapter-Generator richtet sich an die Komponentenentwickler. Sie können Komponenten in Java entwickeln, die sie durch Verwendung verschiedener Muster bei der Methodendeklaration mit wenig Aufwand in die CoSim-Architektur einbetten und als Modellfabrik bereitstellen. Vorhandene Komponenten koppeln die Modellentwickler mit dem CoSim-Component-Connector zu ausführbaren Modellen zusammen. Um existierende Komponenten aufzufinden, steht ein grafischer Nutzerzugang zu Verzeichnisdiensten bereit (Model-Finder). Die Durchführung von Simulationsläufen geschieht schließlich mit der Experimentierumgebung (ExpU).

7.1 Einbettung benutzerdefinierter Modelle

Der CoSim Adapter Generator (Cage) stellt ein Rahmenwerk zur Verfügung, mit dem sich benutzerdefinierte Modelle in die CoSim-Architektur einbetten lassen. Cage erzeugt für Modellkomponenten, die in der Programmiersprache Java vorliegen und einem speziellen Schema für die Vergabe von Methodennamen folgen, vollständige Implementationen von CoSim-Modellfabriken. Dadurch muss sich ein Modellentwickler nicht direkt mit der CoSim-Architektur und den umfangreichen Schnittstellen auseinandersetzen. Der Einsatz von CORBA-IDL wird nur dann notwendig, wenn Datentypen in einer Modellkomponente verwendet werden sollen, die zum Zeitpunkt der Cage-Realisierung selbst noch nicht bekannt waren. Dazu muss der Modellentwickler Quellcode gemäß dem IDL-Java-Mapping (OMG 2001b) für IDL-Stubs, -Skeletons und -Helper durch einen IDL-Compiler erzeugen lassen. Die übersetzten Java-Klassen stellt er dann der Cage-Realisierung zur Verfügung.

7.1.1 Grundsätzliche Arbeitsweise des Generators

Cage untersucht den Objektcode einer Modellimplementation mit dem Reflection Mechanismus der Programmiersprache Java, d.h. es wird kein Quellcode für die Anbindung benötigt. Natürlich verlangt ein Compiler Java-Quellcode,

um den Objektcode zu erzeugen, aber der Quellcode selbst muss für Cage nicht zugänglich sein. Um den Adapter für eine Modellrealisierung zu erstellen, erhält Cage also eine Klassendefinition vorgelegt und prüft, ob diese Klasse verschiedene Regeln einhält. Dazu müssen bestimmte Klassen und Schnittstellen vererbt und einige statische Methoden deklariert werden. Soweit nicht anders angegeben, beziehen sich alle folgenden Klassen und Schnittstellen auf die in Anhang B enthaltenen Definitionen des Cage-Rahmenwerks.

Die möglichen Zugangspunkte erkennt Cage anhand der Methodensignaturen aus der Klasse einer Modellrealisierung. Hier wird ein Namensschema verwendet, das der Deklaration von Attributen per JavaBeans (Hamilton 1997) ähnelt. In der Datei `Check_Declaration.java` findet sich ein Beispiel, das alle Varianten für die Deklaration von Zugangspunkten enthält. Durch ein weiteres Schema lassen sich innerhalb eines Kommunikationsschrittes neben einem Hauptwert auch Kontextwerte übertragen. Hier müssen die Klassen, die für Eingabeparameter und Rückgabewerte einer Methode verwendet werden, bestimmte Felder deklarieren, über die die Kontextwerte ausgetauscht werden können.

Entspricht die vorgelegte Klasse der Modellimplementation allen Deklarationsregeln, so erzeugt Cage daraus eine Modellfabrik sowie eine Modellbeschreibung mit entsprechenden Zugangspunkten und legt sie im Verzeichnis für Objekthüllen ab (vgl. Abschnitt 8.1). Außerdem entsteht Java-Quellcode für neue Unterklassen, der automatisch übersetzt wird. Die neuen Unterklassen binden die ursprüngliche Modellimplementation an das interne Rahmenwerk in Cage.

Um zu erkennen, welche Modellfabriken erzeugt werden sollen, überprüft Cage automatisch und regelmäßig bestimmte Verzeichnisse im lokalen Dateisystem. Entsprechen die enthaltenen Dateien eines Verzeichnisses den in Abschnitt 7.1.6 aufgeführten Regeln zur Modellablage, so wird der Inhalt als Modellfabrik eingelesen. Entdeckt Cage, dass die vorliegenden Definitionen einer bereits laufenden Modellfabrik geändert wurden, so wird diese Modellfabrik inklusive aller vorhandenen Modellexemplare zuerst deaktiviert, bevor versucht wird, die neuen Definitionen zu lesen.

Cage besitzt selbst eine CORBA-Schnittstelle (`::cage::Scanner`, Datei `cage.idl` in Anhang A), mit der sich der Erkennungsmechanismus für Modelldefinitionen fernsteuern lässt. Anstatt also zu warten, bis der Zeitpunkt für eine Verzeichniserkennung erreicht ist, können neue Definitionen auch nach Bedarf eingelesen werden. Per Steuerungsdialog (Abbildung 7.1) lassen sich einzelne Definitionen aktivieren und wieder löschen. Auch das Durchsuchen von Verzeichnissen kann nach Bedarf angestoßen werden.

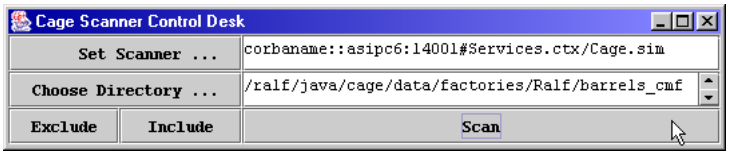


Abbildung 7.1: Steuerungsdialog Cage

7.1.2 Vererbung

Jede gültige Modellrealisierung muss die Schnittstelle `Model_Wrapper` implementieren. Darin sind die Übergänge zwischen Modellnutzungsphasen beschrieben. Die Modellrealisierung bekommt über die enthaltenen Methoden dieser Schnittstelle mitgeteilt, wann welche Übergänge vorgenommen werden sollen und kann den Lebenszyklus eines Modellexemplars entsprechend nachvollziehen.

Die Klasse `Model_Implementation` implementiert die Schnittstelle durch leere Methodenrümpfe. Erbt die Modellrealisierung von dieser Klasse, so stehen zusätzlich Methoden für den synchronisierten Zugriff auf die Simulationsuhr sowie zum kontrollierten Ausführen von nebenläufigem Java-Code (Threads) zur Verfügung. Jedes Modellexemplar bekommt dabei eine eigene Gruppe (ThreadGroup) zugeordnet, innerhalb derer der geladene Code ausgeführt werden muss. Die Gruppierung aller laufenden Threads vereinfacht das Auffinden eines gesuchten Threads beim Debugging. Die Methode `run_thread` sorgt dann für die richtige Einordnung neuer Threads.

Es ist weiterhin ein spezieller Konstruktor in der Modellrealisierung notwendig, der zwei Ausgabeströme entgegennimmt: Der erste Ausgabestrom sollte für Standardausgaben der Modellrealisierung genutzt werden, der zweite für Fehlermeldungen. Das Cage-Rahmenwerk erzeugt diese Ausgabeströme (vgl. Abschnitt 7.1.7). Sie sind dazu gedacht, Ausgaben unterschiedlicher Modellfabriken zu trennen, die innerhalb derselben virtuellen Maschine agieren.

7.1.3 Statische Methoden

Jede gültige Modellrealisierung muss einige statische Methoden implementieren, um Cage bei der Modellbeschreibung zu unterstützen. Die zugehörigen Signaturen lauten:

```
public static String         the_factory_name ();
public static String         the_factory_url  ();
public static String         the_class_name   ();
public static String         the_class_url    ();
public static Dependencies[] the_dependencies ();
```

Der Fabrikname (`the_factory_name`) wird als Name im CORBA Naming Service verwendet, unter dem Cage die zu erzeugende Modellfabrik einträgt. Der Verweis `the_factory_url` dient zu Informationszwecken, um ggf. weitere Erläuterungen zum Verhalten der Modellfabrik nachzuschlagen. Genauso geben `the_class_name` und `the_class_url` die entsprechenden Informationen für die zu erzeugende Modellbeschreibung an.

Die Modellabhängigkeiten (`the_dependencies`) spiegeln die Ein- und Ausgabebeziehungen der Zugangspunkte wider. Um den Wert einer Ausgabegröße zu berechnen, muss eine Modellrealisierung hin und wieder Werte von Eingabegrößen betrachten. Bei der Kopplung von Modellexemplaren werden dann die Abhängigkeiten zwischen den Zugangspunkten herangezogen, um mögliche Zyklen bei der Berechnung zu erkennen. All diese Abhängigkeiten sind zu beschreiben mit Hilfe der Datenstruktur `Dependencies`. Dabei wird der Name einer Ausgabegröße im Datenfeld `variable` hinterlegt. Im Datenfeld `requires` stehen die Namen aller Eingabegrößen, die für die Berechnung der Ausgabegröße notwendig sind.

7.1.4 Methoden für Zugangspunkte

Die notwendigen Methoden für den Zugang zu einer Modellgröße hängen direkt ab von der gewählten Kommunikationsart. Abschnitt 6.5 diskutierte die acht Varianten, die sich aus unterschiedlichen Werten für die drei Eigenschaften Richtung, Aktivität und Kopplung ergeben. Abbildung 7.2 ordnet die von Cage erwarteten Methoden nach diesen Varianten ein. Die beiden Varianten, die die lose Kopplung enthalten, und an denen der zugehörige Nachrichtenkanal nach dem Pull-Modell arbeiten soll, benötigen statt einer gleich drei Methoden. Das Pull-Modell sieht zunächst einmal zwei verschiedene Anfragen vor: die blockierende Methode `pull` und die nicht blockierende Methode `try_pull`. Für die `try_pull`-Methode werden zwei Rückgabewerte benötigt. Da in Java nur ein Rückgabewert je Methode erlaubt ist, und der zweite Rückgabewert im Falle nicht vorliegender Datenwerte keine Bedeutung hat, werden hier zwei getrennte Methoden ausgewertet, um die beiden Werte zu erhalten.

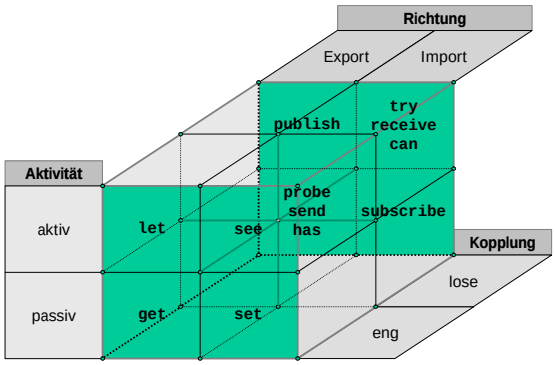


Abbildung 7.2: Einordnung der Methoden zur Datenkommunikation

Cage setzt nun je Variante ein spezielles Muster voraus, damit die Anbindung an das Rahmenwerk erfolgen kann. Jedes Muster enthält drei Freiheitsgrade, um verschiedene Datentypen, Nutzungsphasen und Benennungen des zugehörigen Zugangspunktes zuzulassen. Jeder Methodenname besteht aus den drei Teilen Präfix, Kurzname und Suffix. Das Präfix repräsentiert die Kommunikationsart als dreibuchstabige Abkürzung, der Kurzname identifiziert den eigentlichen Zugangspunkt, und das Suffix steht für die Nutzungsphase (`parameter`, `runtime`, `result`, `clock` oder `context`). Der Kurzname besteht aus einer Zeichenkette, die auch als Variablenname in Java verwendet werden darf. Zusätzlich sind Ziffern am Anfang der Zeichenkette erlaubt, da die Kurznamen nicht für sich alleine als Methodenname stehen, sondern durch Präfix und Suffix eingerahmt sind.

Als *Datentyp* einer Modellvariablen kann jeder Java-Basisdatentyp verwendet werden (`boolean`, `byte`, `char`, `double`, `float`, `long`, `int` oder `short`), der Java-Objekttyp `java.lang.String`, weiterhin jeder benutzerdefinierte IDL-Typ, für den ein IDL-Compiler die entsprechende `Helper`-Klasse erzeugt hat, oder eine benutzerdefinierte Java-Klasse zur Aufnahme von Kontextwerten (vgl. Abschnitt 7.1.5). Der Einsatz von Java-Arrays ist erlaubt, wenn ein Typverzeichnis (s.u.) angegeben wird, das einen entsprechenden Listentyp enthält. Für die Java-Basistypen sollte ein Verzeichnis diese Listentypen bereits enthalten, da ihre CORBA-Pendants im CORBA-Grundmodul deklariert und automatisch in

das Verzeichnis importiert wurden.

Tabelle 7.1 stellt die Signaturen vor, die zur Deklaration eines Zugangspunktes notwendig sind. Die letzte Signatur (`_non`) ist nur für das spezielle Szenario gedacht, in dem ein Zugangspunkt den Datentyp für eine Hilfsgröße definiert, die ausschließlich im Zusammenhang mit Kontextwerten benutzt wird und nicht direkt abfragbar sein soll. Für jede Modellvariable können noch weitere Methoden implementiert sein, die zusätzliche beschreibende Informationen bereitstellen (Tabelle 7.2). Diese optionalen Methoden sind mit den zugehörigen notwendigen Methoden durch einen gemeinsamen Kurznamen im Mittelteil des Methodennamens verbunden. Sie tragen alle das Präfix `the_`. Das Suffix gibt dann den Zweck der Beschreibung an.

Richtung	Kopplung	Aktivität	Signatur-Muster
Import	eng	aktiv	`public` *Typ* `see_`*Kurzname_szenario* `();`
Import	eng	passiv	`public void set_`*Kurzname_szenario* `(`*Typ var*`);`
Import	lose	aktiv	`public` *Typ* `rec_`*Kurzname_szenario* `();`
			`public` *Typ* `try_`*Kurzname_szenario* `();`
			`public boolean can_`*Kurzname_szenario* `();`
Import	lose	passiv	`public void sub_`*Kurzname_szenario* `(`*Typ var*`);`
Export	eng	aktiv	`public void let_`*Kurzname_szenario* `(`*Typ var*`);`
Export	eng	passiv	`public` *Typ* `get_`*Kurzname_szenario* `(`*Typ var*`);`
Export	lose	aktiv	`public void pub_`*Kurzname_szenario* `(`*Typ var*`);`
Export	lose	passiv	`public` *Typ* `snd_`*Kurzname_szenario* `();`
			`public` *Typ* `prb_`*Kurzname_szenario* `();`
			`public boolean has_`*Kurzname_szenario* `();`
—	—	—	`public` *Typ* `non_`*Kurzname*`_context ();`

Tabelle 7.1: Signaturen notwendiger Methoden

Zweck	Signatur-Muster
langer Name	`public static String the_`*Kurzname*`_name ();`
Anfangswert	`public static` *Typ* `the_`*Kurzname*`_initial ();`
Gültigkeitsbereiche	`public static` *Typ*`[] the_`*Kurzname*`_valids ();`
Typverzeichnis	`public static String the_`*Kurzname*`_repository ();`
Uhrzeit-Automatik	`public static boolean the_`*Kurzname*`_auto ();`
Uhrzeit-Untergrenze	`public` *Typ* `min_`*Kurzname*`_clock ();`
Uhrzeit Obergrenze	`public` *Typ* `max_`*Kurzname*`_clock ();`

Tabelle 7.2: Signaturen optionaler Methoden

Der zusätzliche lange Name (`_name`) kann auch Sonderzeichen enthalten und wird als Namenseintrag im CORBA Naming Service verwendet, unter dem der Zugangspunkt gespeichert wird. Fehlt der lange Name, so wird automatisch der Kurzname verwendet.

Die Modellvariable bekommt bei der Erzeugung eines neuen Modellexemplars den Initialwert (`_initial`) zugeordnet. Fehlt der Initialwert einer Eingabevariablen, die als Parameter verwendet wird, so muss der Benutzer des Model-

lexemplars den Wert dieser Variablen vor dem Start eines Simulationslaufs auf einen definierten Wert setzen, sonst erzeugt der Startaufruf eine Fehlermeldung.

Als Gültigkeitsbereich für mögliche Datenwerte (`_valids`) ist ein zweidimensionales Feld des Einzeldatentyps anzugeben. Die innere Feldgrenze beträgt 2, wobei der erste innere Datenwert eine untere Schranke definiert, und der zweite eine obere Schranke. Die äußere Feldgrenze unterliegt keinen Bedingungen. Alle äußeren Intervalle zusammen sind als Vereinigungsmenge aller gültigen Werte zu interpretieren. Um beispielsweise den ganzzahligen Gültigkeitsbereich $[-9; -2] \cup \{0\} \cup [2; 9]$ anzugeben, kann folgender Java-Ausdruck eingesetzt werden:

```
new int[][] {{-9, -2}, {0, 0}, {2, 9}}
```

Ein Typverzeichnis (`_repository`) wird als interoperable Objektreferenz (IOR) angegeben. Die IOR muss für ein CORBA Interface Repository stehen. Das Typverzeichnis ist immer dann anzugeben, wenn der Zugangspunkt einen modellspezifischen IDL-Typ als Datentyp verwendet. Dieser IDL-Typ muss auch im Typverzeichnis enthalten sein, damit Cage den Typaufbau untersuchen kann.

Für diejenigen Zugangspunkte, die den Zugriff auf die Simulationsuhr beschreiben, können Implementierungen dieses Zugriffs automatisch erzeugt werden. Insbesondere lassen sich automatisch Threads anstoßen, die die Abfrage bzw. Übermittlung von neuen Uhrzeitwerten vornehmen, wenn die Modellvariable selbst als aktiver Kommunikationspartner deklariert ist. Die Automatik wird aktiv, wenn die `_auto`-Methode den Wert `true` liefert.

Konzeptuell gibt es nur eine einzige Simulationsuhr. In der automatischen Variante wird diese Uhr als Java-Objektvariable vom Typ `double` repräsentiert. Es lassen sich jedoch mehrere Zugangspunkte zur Simulationsuhr deklarieren, um beispielsweise Lese- und Schreiboperationen zu ermöglich, oder um Repräsentationen durch andere Zahltypen (Ganzzahlen etc.) zu erzeugen. Die notwendigen Typumwandlungen nimmt Cage dann automatisch vor.

Die Gültigkeitsbereiche der Simulationsuhr können sich zwischen verschiedenen Simulationsläufen ändern. Sie hängen in der Regel von den Parametern ab, die vor Beginn der Laufzeitphase eingestellt wurden. Um eine angepasste Visualisierung des Simulationsfortschritts zu ermöglichen, sind die Gültigkeitsbereiche für die Simulationsuhr als nicht statische Methoden (`min_` und `max_`) zu implementieren. In manchen Situationen lässt sich die Endezeit eines Simulationslaufs nicht im Voraus bestimmen, etwa wenn ein stationäres Verhalten abgewartet wird. In diesen Fällen sollte die `max_`-Methode trotzdem eine grobe Schätzung abgeben, damit die Aktivität des Simulationslaufs erkennbar bleibt.

7.1.5 Abbildungsvorschriften für Kontextwerte

Einige Datenwerte von Modellvariablen lassen sich nur im Kontext anderer Datenwerte interpretieren. Beispielsweise sind mit dem Wert einer bestimmten Zelle aus einem Datenfeld auch die Indizes des Datenfeldes anzugeben, die die Zelle identifizieren. Damit ist es notwendig, bei der Übertragung von Datenwerten mehrere Einzelwerte zusammenzufassen. Eine Übertragung in mehreren Schritten ist nicht sinnvoll, da die Simulationszeit zwischen diesen Schritten fortschreiten könnte und somit zusätzliche Synchronisationsmechanismen notwendig würden.

Da es die Programmiersprache Java nicht erlaubt, mehrere Rückgabewerte eines Methodenaufrufs abzuliefern, verwendet Cage ein besonderes Objektmuster, um die Zusammenfassung mehrerer Datenwerte zu beschreiben. Für Ein- und Ausgabegrößen unterscheidet sich das Muster nicht. Jeder Datentyp eines Zugangspunktes, der Kontextwerte vorsieht, muss nach diesem Muster aufgebaut sein. Im Gegensatz zu den zwei Methoden für den Pull-Mechanismus bei der asynchronen Datenübertragung (vgl. Abschnitt 7.1.4) steht die Anzahl der Kontextwerte nicht fest. Deshalb ist es nicht möglich, feste Methodennamen in diesem Muster zu verwenden.

Zunächst sieht dieses Muster die Deklaration einer benutzerdefinierten Java Klasse vor, die von der Klasse `Value_Mapping` abgeleitet wird. Die Klasse `Value_Mapping` enthält eine Objektvariable `valid`. Diese Variable gibt an, ob die zu übertragenen Werte tatsächlich gültige Werte im Simulationsmodell darstellen. Als ungültig markierte Werte werden verwendet, um eine Synchronisation zwischen Kommunikationspartnern durchzuführen (in der Regel über Kontextwerte der Simulationsuhr, vgl. Abschnitt 6.6.2).

Weiterhin schreibt das Kontextmuster eine Objektvariable (nicht statisch) mit dem Namen `value` in der benutzerdefinierten Unterklasse vor. Der Typ der Objektvariablen gibt den Datentyp der Hauptgröße für die zu übertragenden Daten an. Kontexte sind bei der Datenübertragung immer als Schlüssel/Wert-Paare angegeben, wobei der entsprechende Zugangspunkt selbst als Schlüssel dient. Für jeden zu übertragenden Kontextwert muss ein Klasse, die dem Muster folgt, deshalb zwei weitere Variablen besitzen. Der Kontextschlüssel ist als Klassenvariable mit der Signatur `public static java.lang.String` anzugeben, wobei der (Java-) Variablenname mit `_name` enden muss. Diese Klassenvariable enthält dann den Namen des referenzierten Zugangspunktes. Der Kontextwert ist wieder als Objektvariable angegeben und endet mit `_value`. Die Namen der Schlüssel- und der Wertvariablen besitzen das gleiche Präfix und kennzeichnen somit die Zusammengehörigkeit dieser beiden Variablen. Die Klasse `Check_Mapping` in Anhang B gibt ein Beispiel, das dieses Muster für die Simulationszeit als Kontextwert einsetzt.

Einen Spezialfall, bei dem kein Hauptwert, sondern ausschließlich Kontextwerte benutzt werden, stellt die Angabe von Argumenten einer eng gekoppelten, passiven Ausgabegröße dar. Mit der zugehörigen `get`-Methode (Abbildung 7.2 und Tabelle 7.1) ist ein parametrisierter Abruf möglich. Hier braucht die benutzerdefinierte Unterklasse, die als Parameter angegeben wird, lediglich die Markierungsschnittstelle `Context_Mapping` zu implementieren. Die Schnittstelle ist leer, d.h. sie erfordert keine zusätzliche Realisierung von Methoden, aber Cage benötigt diese Markierung, um die Klasse von anderen Klassen zu unterscheiden, die nicht dieses Kontextmuster verwenden. Die `_name` und `_value` Variablen der Unterklasse geben wie gewohnt Schlüssel/Wert-Paare an.

7.1.6 Ablageregeln einzulesender Deklarationen

Beim Start erhält Cage ein zentrales *Datenverzeichnis* aus dem lokalen Dateisystem zugeordnet, das alle Ein- und Ausgaben zusammenhält. Unterhalb des Datenverzeichnisses befindet sich ein Verzeichnis `factories` für Modellfabriken (vgl. Tabelle 7.3). Cage untersucht das Fabrikverzeichnis regelmäßig, um Unterverzeichnisse mit neuen Modellfabriken aufzuspüren. Diese Unterverzeichnisse müssen folgende Namenskonventionen einhalten:

- Ein Verzeichnis mit Modelldefinitionen muss mit der Abkürzung `_cmf` enden (CoSim Model Factory).
- Es kann irgendwo unterhalb des `factories` Verzeichnisses liegen.
- Es enthält eine oder mehrere Hinweisdateien, die mit der Abkürzung `.cmw` (CoSim Model Wrapper) enden. Ein Modellhinweis ist eine Textdatei, die eine Liste von Java-Klassennamen aufführt (mit Angabe des vollständigen Java-Package). Jeder Listeneintrag steht in einer eigenen Zeile und weist Cage an, die Klasse als eigenständige Fabrikdefinition zu behandeln.

```
factories
  Benutzerverzeichnis
    Modellverzeichnis_cmf
      Modellbibliothek.jar
      Modellnamensraum
        Modellname_1.class
        Modellname_2.class
        Modell.cmw
          Modellnamensraum.Modellname_1
          Modellnamensraum.Modellname_2
logs
  Benutzerverzeichnis
    Modellverzeichnis
      Modellnamensraum.Modellname_1.err
      Modellnamensraum.Modellname_1.out
      Modellnamensraum.Modellname_2.err
      Modellnamensraum.Modellname_2.out
src
  Benutzerverzeichnis
    Modellverzeichnis
      de.unihh.informatik.bachmann.cage.subclass_generator
        Modellnamensraum
          Generated_Subclass_Of_Modellname_1.java
          Generated_Subclass_Of_Modellname_2.java
classes
  Benutzerverzeichnis
    Modellverzeichnis
      de.unihh.informatik.bachmann.cage.subclass_generator
        Modellnamensraum
          Generated_Subclass_Of_Modellname_1.class
          Generated_Subclass_Of_Modellname_2.class
extensions
  Werkzeugbibliothek.zip
  Simulationsbibliothek.jar
  Verweise.url
    file:///externes_Verzeichnis/Bibliothek.zip
    http://Zielrechner:port/entferntes_Verzeichnis/Bibliothek.jar
  Benutzernamensraum
    Benutzerklasse.class
```

Tabelle 7.3: Datenverzeichnis

7.1.7 Ablageregeln erzeugter Modellfabriken

Falls eine Klasse den Regeln zur Modelldefinition entspricht, erzeugt Cage verschiedene Ausgaben. Um zu beschreiben wohin diese Ausgaben gelangen, sind einige Definitionen nützlich, die Cage aus dem Speicherort der geladenen Modellklasse gewinnt.

- Als *Modellpräfix* wird der Restteil des Verzeichnispfades bezeichnet, der das `factories` Verzeichnis gerade nicht mehr umfasst. Das Modellpräfix enthält auch nicht die Abkürzung `_cmf`. Für die Tabelle 7.3 ergibt sich beispielsweise die Zeichenkette *`Benutzerverzeichnis`*`/`*`Modellverzeichnis`*`.`
- Jeder Eintrag eines Klassennamens in eine Hinweisdatei heißt *Modellname*. Dabei bleibt das Java-Package zunächst unberücksichtigt.
- Das Java-Package eines Klassennamens heißt *Modellnamensraum*.
- Da Cage als Ausgabe Quell- und Objektcode *erzeugt*, müssen Namenskonflikte mit bestehenden Klassen von vornherein vermieden werden. Dazu *konstruiert* Cage ein neues Java-Package, das sich aus dem Präfix `de.unihh.informatik.bachmann.cage.subclass_generator.` und dem Modellnamensraum zusammensetzt.
- Der *Datenkontext* ist ein Naming Service Context (vgl. Abschnitt 6.7.1), der ähnlich wie das Datenverzeichnis beim Start des Adapter Generators festgelegt wurde.

Die Ausgaben für eine gelesene Modellrealisierung werden nun im Datenverzeichnis wie folgt abgelegt:

- Je Modellname existieren zwei Protokolldateien, eine für Modellausgaben und eine für Modellfehler. Die Dateien befinden sich im Verzeichnis *`Datenverzeichnis`*`/logs/`*`Modellpräfix`*`/` und heißen *`Modellnamensraum`*`.`*`Modellname`*`.out` bzw. *`Modellnamensraum`*`.`*`Modellname`*`.err`.
- Je Modellname wurde eine Quellcode-Datei erzeugt, die eine Unterklasse der ursprünglichen Modellrealisierung darstellt. Diese Unterklasse ist im konstruierten Java-Package enthalten und heißt `Generated_Subclass_Of_`*`Modellname`*`.` Die erzeugten Quellcode-Dateien befinden sich im Verzeichnis *`Datenverzeichnis`*`/src/`*`konstruiertes_Package`*`/`. Sie binden die ursprüngliche Modellimplementation an das interne Rahmenwerk in Cage (vgl. Abschnitt 8.3.3) und realisieren die Uhrzeit- sowie die Thread-Automatik (vgl. Abschnitte 7.1.4 und 8.3.5).
- Jede erzeugte Quellcode-Datei wurde übersetzt und ihr Objektcode ist abgelegt im Verzeichnis *`Datenverzeichnis`*`/classes/`*`konstruiertes_Package`*`/`.

Je Modelldefinition wird eine Modellbeschreibung erzeugt und in einem neuen Unterkontext gespeichert. Diese Modellbeschreibung enthält die Zugangspunkte, die gemäß den Regeln zur Modelldefinition erkannt wurden. Der neue Unterkontext entsteht in mehreren Stufen aus dem Modellpräfix und dem vollständigen Klassennamen.

- Je Verzeichnis im Modellpräfix existiert ein entsprechender Kontext im Naming Service mit dem gleichen Namen wie das Verzeichnis, zuzüglich der Endung `.ctx`.
- Der letzte Unterkontext für die neue Modellbeschreibung ist dann mit dem vollständigen Klassennamen bezeichnet und endet genauso mit `.ctx`.

In diesem letzten Kontext sind nun alle Objekte der Modelldefinition gebunden. Um die Namen dieser Objekte zu ermitteln, wurden die statischen Methoden `get_class_name` bzw. `get_factory_name` aus der angegebenen Java-Klasse ausgewertet. Es entstehen mehrere Bindungen. Die zugehörigen Schnittstellen der erzeugten CORBA-Objekte finden sich im Anhang A wieder.

- Die neue Modellfabrik selbst steht als `Model_Factory`-Objekt unter dem Namen *`factory_name`*`.factory` zur Verfügung.
- Eine Beschreibung der Modellabhängigkeiten wurde als `Lookup_Dependencies`-Objekt unter dem Namen *`factory_name`*`.dependencies` abgelegt.
- Die Klassifikation der Zugangspunkte findet sich als `Lookup_Class`-Objekt unter dem Namen *`class_name`*`.class` wieder.
- Je Zugangspunkt erscheint ein Eintrag als `Lookup_Connection`-Objekt unter dem Namen des Zugangspunktes. Die Endung eines Eintrags richtet sich hier nach der Klassifikation des Zugangspunktes (`.parameter`, `.runtime`, `.result`, `.context` oder `.clock`).
- Besitzt ein Zugangspunkt einen voreingestellten Wert, so existiert auch hierfür ein Eintrag als `Lookup_Value`-Objekt mit dem Namen des Zugangspunktes und der Endung `.initial`.
- Besitzt ein Zugangspunkt einen oder mehrere Gültigkeitsbereiche für mögliche Datenwerte, so existiert für jede Unterschranke ein Eintrag als `Lookup_Value`-Objekt mit dem Namen des Zugangspunktes und der Endung `.lower`, und je Oberschranke ein Eintrag mit der Endung `.upper`. Sind mehrere Intervalle (als Vereinigung der Wertemengen) angegeben, nummeriert Cage die Namen der Schranken ab dem zweiten Intervallindex aufsteigend, beginnend mit dem Zahlwert „1“ des zweiten Index.

7.1.8 Nachladen von Klassen

Da Cage in der Lage sein muss, sämtliche Objektcodes zur Durchführung von Simulationsläufen während des Betriebs nachzuladen, wurde eine Hierarchie von `java.lang.ClassLoader` Objekten installiert, um die modellspezifischen Klassen zu finden:

- Die Wurzel der ClassLoader Hierarchie besteht aus einem *Java Runtime Environment* (JDK 1.3.1). Da diese Umgebung jedoch einige Klassen der CORBA Spezifikation 2.3 nur unzureichend implementiert, hat die verwendete ORB-Bibliothek (z.B. VisiBroker für Java 4.5) allerhöchste Priorität und ist als Aufrufoption `-Xbootclasspath/p:` anzugeben, um die entsprechenden Klassen des JDK nicht zur Geltung kommen zu lassen.

```
corbaname:iiop:Zielrechner:port/NameService#cage_context.ctx
  Benutzerverzeichnis.ctx
    Modellverzeichnis.ctx
      Modellnamensraum.ctx
        Fabrikname.factory
        Fabrikname.dependencies
        Beschreibungsname.class
        Parameter_1.parameter
        Parameter_1.initial
        Parameter_2.parameter
        Parameter_2.lower
        Parameter_2.upper
        Parameter_2_1.lower
        Parameter_2_1.upper
        Laufzeitgröße_1.runtime
        Ergebnisgröße_1.result
```

Tabelle 7.4: Erzeugte Einträge beim Namensdienst

- Der Suchpfad des Standard-Laders besteht aus allen beim Programmstart angegebenen Klassenpfaden. Darunter fallen die notwendigen Klassen, um Cage selbst zu laden (z.B. `de.unihh.informatik.bachmann.cage.reader.Scanner_Impl`) und die benötigten Bibliotheken von Drittanbietern. Der Standard-Lader ist unterhalb des Wurzel-Laders angesiedelt.

- Unterhalb des Datenverzeichnisses gibt es ein Unterverzeichnis mit dem Namen `extensions`. Alle dort enthaltenen `.zip` oder `.jar`-Bibliotheken (auch in beliebigen tieferen Unterverzeichnissen) sowie dieses Unterverzeichnis selbst ergeben den Suchpfad des nächsten ClassLoaders. Hier sollten Bibliotheken abgelegt werden, die von mehreren Modelldefinitionen gemeinsam benutzt werden können (beispielsweise das Rahmenwerk Desmo-J für ereignisdiskrete Simulation, vgl. Abschnitt 9.2.4). Die Reihenfolge der einzelnen Sucheinträge ist unspezifisch, d.h. es sollten keine Annahmen über eine bestimmte (Teil)-Reihenfolge gemacht werden. Dieser Erweiterungs-Lader ist unterhalb des Standard-Laders angesiedelt.

- Jedes Verzeichnis mit neuen Modellfabriken bekommt einen eigenen Lader zugeordnet. Der Suchpfad besteht dabei aus dem Verzeichnis der Modellfabriken selbst sowie allen darin enthaltenen `.zip` oder `.jar` Bibliotheken (auch in beliebigen tieferen Unterverzeichnissen). Auch hier ist die Reihenfolge der einzelnen Sucheinträge unspezifisch. Dieser Verzeichnis-Lader ist unterhalb des Erweiterungs-Laders angesiedelt.

Da die `.zip` oder `.jar` Bibliotheken in beliebigen Unterverzeichnissen liegen dürfen, sucht Cage auch alle vorhandenen Unterverzeichnisse im Vorwege ab. Wird mit dem Steuerungsdialog ein zu durchsuchendes Modellverzeichnis angegeben, so sollte dieses Verzeichnis frei von weiteren, von Cage nicht zu inspizierenden Verzeichnissen sein, um die Suche nach Bibliotheken in angemessenem zeitlichen Rahmen zu halten. Aus diesem Grund ist das Heimatverzeichnis eines Benutzers in der Regel *nicht* zur Ablage von Modellbeschreibungen geeignet. Es sollte hier ein eigenes Verzeichnis für externe Modellbeschreibungen vorliegen.

Anstatt Bibliotheken direkt im Erweiterungs- bzw. im Modellverzeichnis abzulegen, besteht die Möglichkeit einer weiteren Indirektionsstufe. Trifft Cage auf eine Datei mit der Endung `.url`, so wird diese Datei als Liste aus URLs auf externe Bibliotheken interpretiert. Erlaubte Zugriffsprotokolle sind dabei `file://`, `ftp://` und `http://`.

7.2 Grafischer Zugang zu Verzeichnisdiensten

Der Zugang zu Verzeichnisdiensten wie zum CORBA Namensdienst (Abschnitt 6.7.1) oder zum CoSim Objekthüllendienst (Abschnitt 6.4.2) direkt über die deklarierten CORBA Schnittstellen ist für einen Modellierer wenig komfortabel, da eine Programmiersprache zur Abfrage eingesetzt werden muss. Vielmehr ist eine direkte Manipulation der vorhandenen Strukturen sinnvoll, die ungeübten Modellnutzern den Zugang zur CoSim-Umgebung durch grafische Benutzungsschnittstellen erleichtert (vgl. Shneiderman 1998, S. 71f. u. Kap. 6).

Abbildung 7.3: Console

Deshalb wurde die Java-Anwendung *Console* entwickelt, die auf vielen Plattformen auch durch Drag & Drop-Gesten nutzbar ist. Mit dieser Anwendung lassen sich die Strukturen des CORBA Namensdienstes, des CORBA Schnittstellenverzeichnisses, des CoSim Objekthüllendienstes und des CoSim Suchdienstes für Modellfabriken einsehen und teilweise verändern. Jede Schaltfläche in Abbildung 7.3 erzeugt ein neues Fenster, das eines der im Folgenden beschriebenen Werkzeuge repräsentiert.

7.2.1 Naming Context Browser

Ein Kontext im CORBA Namensdienst wird angezeigt innerhalb eines *Naming Context Browser* Fensters. Die darunterliegenden Einträge erscheinen als baumartige Struktur. Obwohl sich mit dem CORBA Namensdienst auch rekursive und netzartige Strukturen erzeugen lassen, sind in der Praxis fast ausschließlich Baumstrukturen anzutreffen, so dass die gewählte Anzeigeform ausreicht. Abbildung 7.4 zeigt unter dem Eintrag `Services.ctx` die im Rahmen der CoSim-Umgebung verwendeten Dienste an. Eingetragenen Diensten, deren Objektreferenz tatsächlich ansprechbar ist, steht ein „+“-Zeichen voran, inaktive Dienste tragen das „-“-Zeichen. Damit die Kontextreferenz für den menschlichen Benutzer lesbar bleibt, ist im Baum die URL-Form angegeben (vgl. Abschnitt 6.7.1). Sie enthält an den notwendigen Stellen das Quotierungszeichen „\“, um beispielsweise zwischen der Punktnotation `Id.Kind` und dem Zeichen „.“ zu unterscheiden.

Durch Schaltflächen lässt sich zunächst der zuoberst dargestellte Kontext festlegen. Danach können auch neue Einträge im selektierten Kontext erstellt und vorhandene Einträge verändert oder gelöscht werden. Der IDL-Typ und die Objektreferenz (als IOR sowie als URL) eines markierten Eintrags sind in den oberen Textfeldern angezeigt. Durch Kopierbefehl oder Mausziehen lassen sich

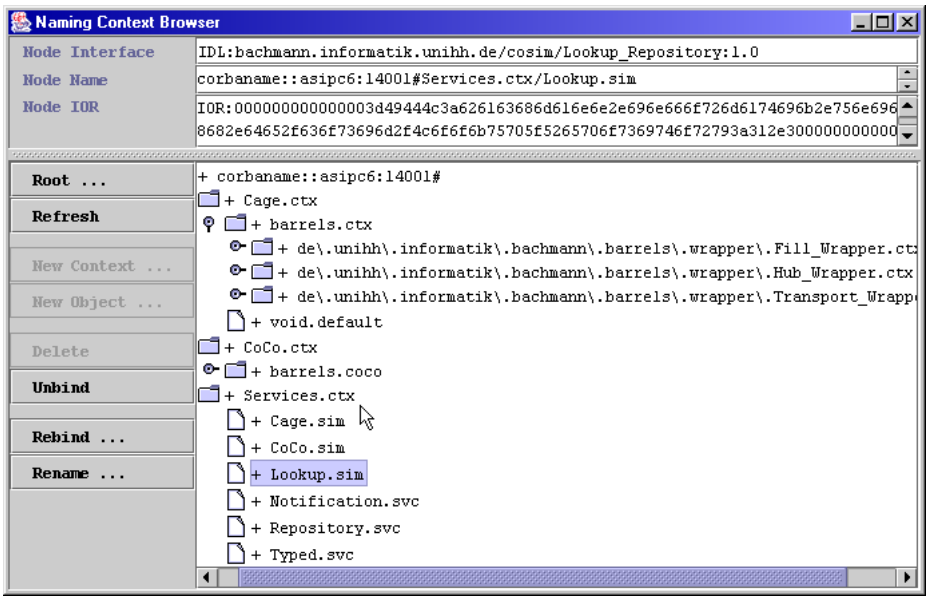


Abbildung 7.4: Naming Context Browser

Einträge kopieren. Kopien enthalten sowohl die ausgewählten Objektreferenzen als auch die damit verbundenen Namen als Text (vgl. Abschnitt 8.2), so dass diese Kopien sowohl in Fenstern des Namensdienstes selbst als auch in anderen Anwendungen einsetzbar sind. In der Experimentierumgebung (Abschnitt 7.4) beispielsweise muss der Benutzer Modellfabriken zur Durchführung von Simulationsläufen eingeben. Die notwendigen Objektreferenzen kann er direkt aus dem Fenster eines Namensdienstes entnehmen.

7.2.2 Lookup Generator

Für den CoSim Objekthüllendienst (Abschnitt 6.4.2) steht ebenfalls ein Werkzeug zur Betrachtung vorhandener und zur Erstellung neuer Einträge bereit. Der *Lookup Generator* (Abbildung 7.5) stellt im oberen Teil den Namen einer Objekthülle und im unteren Teil die verpackte Datenstruktur dar. Über den in Abschnitt 6.4.2 eingeführten, strukturierten Namen lassen sich vorhandene Objekthüllen bei dem angegebenen Verzeichnisdienst nachschlagen, abspeichern, löschen oder umbenennen. Im Feld `NamingContext` findet sich der Kontext im Namensdienst wieder, unter dem ein Eintrag (Felder `Id` und `Kind`) erfolgt ist. Der IDL-Typ zur eingetragenen Objektreferenz, die vom Objekthüllendienst verwaltet wird, findet sich im Feld `Holder Type` wieder.

Als verpackte Datenstrukturen können alle per IDL definierbaren Strukturen vorkommen. Auch hier bietet sich eine baumartige Darstellung an, da sich alle Datenstrukturen in Name/Wert-Paare zerlegen lassen. Der Name findet sich direkt im Baum wieder. Ein zugeordneter Wert ist entweder als Unterknoten verfügbar oder in speziellen Textfeldern (`Type Shown`, `Value Shown`) eingetragen.

Gerade um neue Datenstrukturen zu erstellen, lässt sich die Kopiertechnik per Drag & Drop effektiv im Lookup Generator einsetzen. Beispielsweise bestehen Modellbeschreibungen aus einer Partition von Zugangspunkten. Sobald die

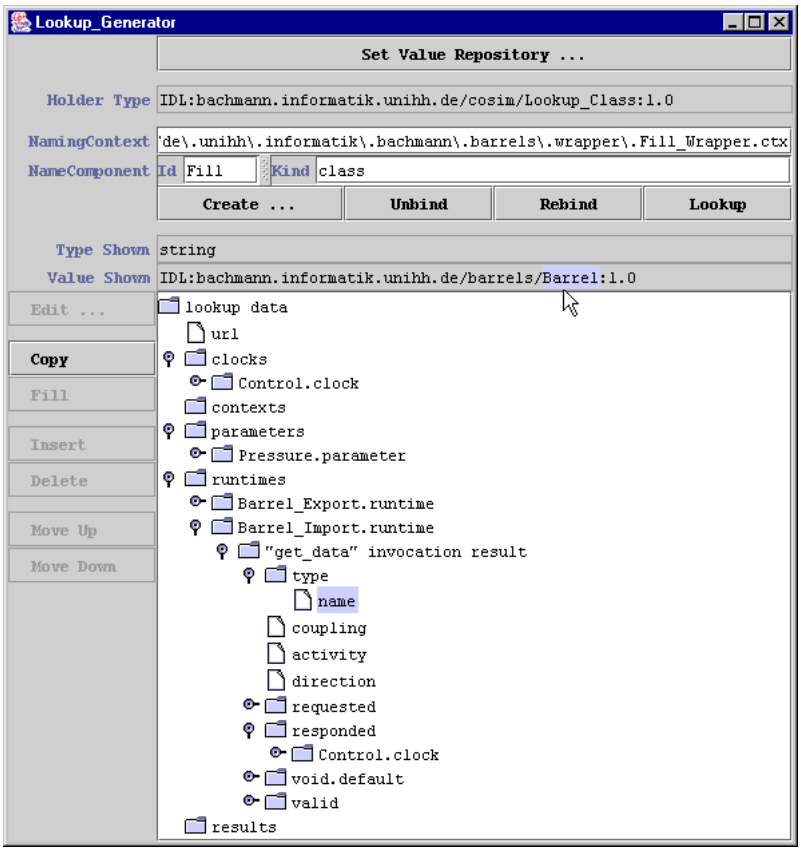


Abbildung 7.5: Lookup Generator

Objektreferenzen für die Zugangspunkte existieren, können sie in die entsprechende Kategorie der Modellklassenbeschreibung per Mausziehen eingefügt werden. Die neue Modellbeschreibung entsteht dann durch Namensvergabe für die zusammengesetzte Datenstruktur und Abspeichern im Objekthüllenverzeichnis.

7.2.3 Interface Repository Browser

Der Lookup Generator benötigt zur Erzeugung neuer Objekthüllen deren Schnittstellentyp. Um auch hier Drag & Drop bzw. Kopiertechniken anwenden zu können, wurde das Werkzeug *Interface Repository Browser* für den Zugang zu den Inhalten von CORBA-Schnittstellenverzeichnissen realisiert. Wiederum erscheinen die Inhalte als Baumstruktur, bedingt durch die modulare Struktur der IDL-Definitionen. Auf das Schnittstellenverzeichnis kann in der aktuellen Realisierung nur lesend zugegriffen werden. Dies bedeutet in der Praxis aber keinerlei Einschränkung, da neue Datenstrukturen üblicherweise als IDL-Datei vorliegen und die zur Zeit verfügbaren Schnittstellenverzeichnisse diese Dateien verarbeiten können.

Abbildung 7.6 zeigt einen Ausschnitt aus dem Inhalt eines Schnittstellenverzeichnisses, das für die Realisierung des Beispielmodells in Abschnitt 9 herangezogen wird. Die Module `Primitives` und `CORBA` sind in jedem Verzeichnis

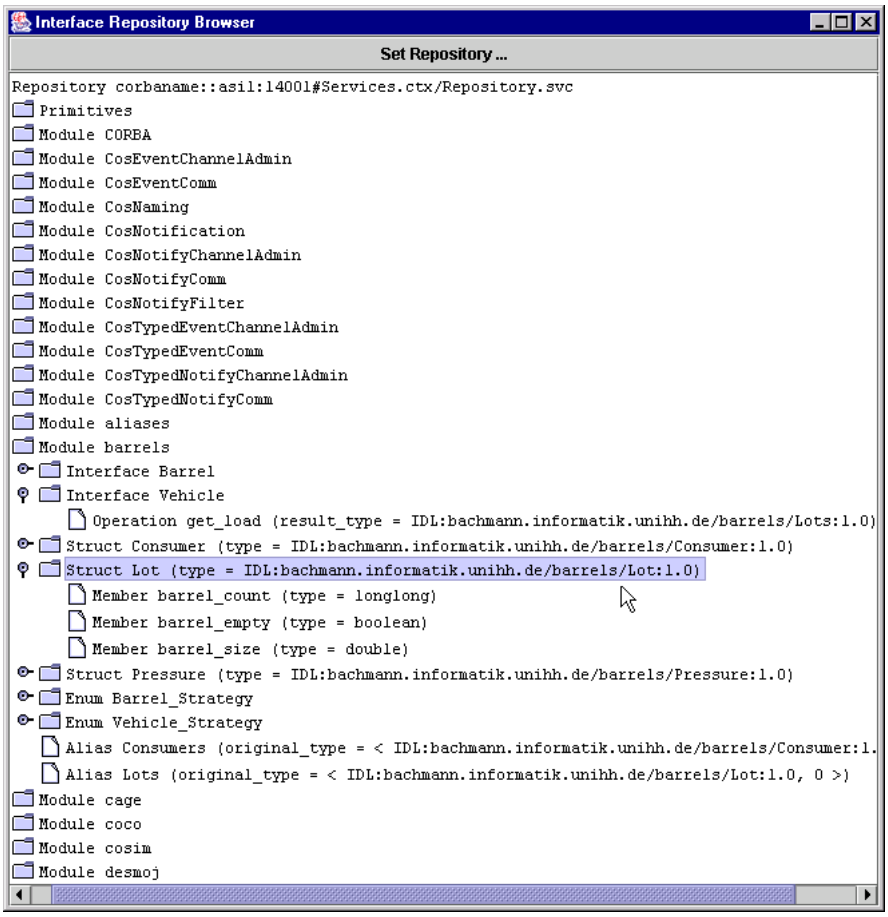


Abbildung 7.6: Interface Repository Browser

automatisch enthalten und stellen die Typrepräsentation von grundlegenden Objekttypen sowie von Basisdatentypen im Rahmen der CORBA-Spezifikation bereit. Die Module mit dem Präfix `Cos` (Common Object Services) gehören zu den CORBA Grunddiensten *Naming*, *Event* und *Notification*. Zur CoSim-Umgebung gehören die Module `alias`, `cosim`, `cage` und `coco`. Speziell für das Beispielmodell werden die Module `barrels` und `desmoj` benötigt.

7.2.4 Factory Repository Browser

Die Inhalte eines Fabrikverzeichnisses (vgl. Abschnitt 6.7.3) lassen sich betrachten und verändern durch das Werkzeug *Factory Repository Browser* (Abbildung 7.7). Die Zuordnung von Modellbeschreibungen zu Modellfabriken ist durch eine horizontale Teilung des Fensters dargestellt. Auf der linken Seite sind die bekannten Modellbeschreibungen eingetragen, rechts erscheinen die zugeordneten Modellfabriken. Der Aufbau einer Modellbeschreibung lässt sich genau wie im Lookup Generator als Baumstruktur detailliert weiterverfolgen.

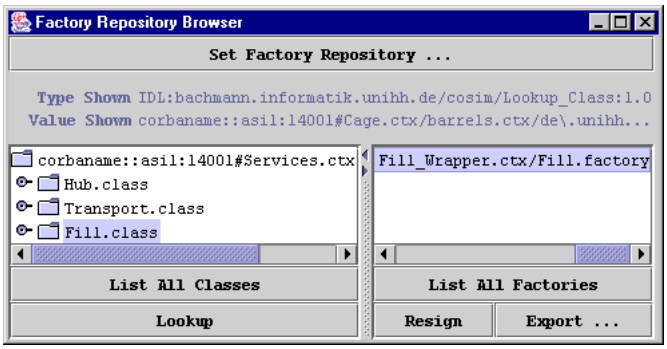


Abbildung 7.7: Factory Repository Browser

7.3 Kopplung von Modellkomponenten

Das wesentliche Merkmal einer CoSim-Komponente ist die Menge ihrer Zugangspunkte. Die Zugangspunkte bestimmen den möglichen Datenfluss hin zur Komponente bzw. von ihr weg. Um Komponenten zu koppeln, spezifiziert man nun ähnlich wie im DEVS-Ansatz (Abschnitt 4.3.3) den Datenfluss zwischen mehreren Komponenten und erhält so eine neue, zusammengesetzte Komponente. Damit wird eine hierarchische Modellbildung möglich, denn die gekoppelten Komponenten können ihrerseits wieder zusammengesetzt sein (Abbildung 7.8).

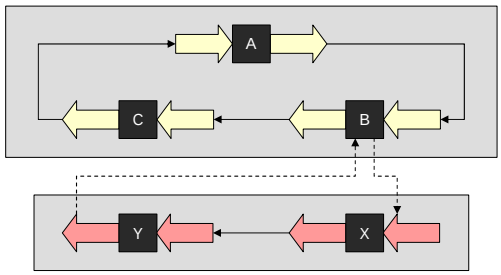


Abbildung 7.8: Hierarchische Modellbildung

Page u. a. (1998) nennen zwei Extreme aus der Palette der Möglichkeiten, durch die ein Modellentwickler die Modellkopplungen beschreiben kann. Einerseits lassen sich Kopplungen rein visuell zusammensetzen, etwa durch Graphenediteren. In diesem Fall ist gar keine Programmierung mehr notwendig, wodurch allerdings auch Flexibilität verloren geht, etwa bei m-zu-n-Beziehungen (vgl. Abschnitt 5.3). Das andere Extrem erhält man, wenn die Kopplungen durch den Modellentwickler programmiert werden müssen. Dabei lassen sich beliebige Datentransformationen und -weiterleitungen realisieren. Der Aufwand zur Darstellung des Datenflusses wird aber in der Regel hoch sein. Zusätzlicher Aufwand entsteht, wenn sich der Modellentwickler erst in die verwendete Programmiersprache einarbeiten muss. Zwischen diesen beiden Extremen bewegen sich spezielle Beschreibungssprachen, wie etwa MSL (vgl. MOBILE in Abschnitt 4.4.5).

Eine grafische Darstellung kann durch geeignete Gruppierung der Modellteile

einem Betrachter schneller die wesentlichen Zusammenhänge deutlich machen (vgl. Häuslein 1993, Abschn. 5.2.4). Als nachteilig kann sich bei einer grafischen Darstellung der notwendige Platzbedarf an Darstellungsfläche herausstellen. Enthalten die Kopplungen nämlich viele Einzelkomponenten oder viele benutzte Zugangspunkte, so sind auch entsprechend viele Datenwege anzugeben, die durch Linienverbindungen dargestellt werden. Bei vielen Linienverbindungen fällt es dann schwer, die zugehörigen Knoten aufzufinden. Abhilfe können hier indirekte Verbindungen schaffen, die etwa durch gleichlautende Symbolbenennungen eingeführt werden. Auch durch unterschiedliche Sichten auf ein Diagramm lässt sich dieses Problem umgehen.

Der CoSim Component Connector (CoCo) ist ein Kopplungswerkzeug, das sowohl die visuelle als auch die programmierbare Kopplung ermöglicht. CoCo enthält einen Grapheneditor, um Diagramme für gekoppelte Simulationsmodelle zu erstellen. Innerhalb des Grapheneditors lassen spezielle Knoten den programmierten Zugriff auf das zu Grunde liegende Rahmenwerk zu. Weiterhin stellt CoCo die zugehörige Ausführungsumgebung zur Verfügung, mit der die Kopplungsdiagramme interpretiert und als CoSim-Modellfabriken zur Verfügung gestellt werden können.

7.3.1 Das Kopplungswerkzeug im Überblick

Die beiden Teile des Kopplungswerkzeugs CoCo (Grapheneditor und Ausführungsumgebung) bauen direkt aufeinander auf. Sie werden jedoch als unabhängige Anwendungen aufgerufen. Nachdem der Modellersteller eine Kopplungsspezifikation mit dem Grapheneditor erstellt hat, stößt er eine Konsistenzprüfung an. Verläuft diese Prüfung erfolgreich, so erzeugt der Grapheneditor eine spezielle Zwischenrepräsentation als Datei mit Endung `.coco`. Die Ausführungsumgebung liest dann eine oder mehrere dieser Dateien ein und erzeugt daraus Modellfabriken, die die entsprechenden CoSim-Schnittstellen implementieren. Die Aktivierung der Modellfabriken erfolgt mit CoCo über einen ähnlichen Mechanismus wie mit Cage für die Einbettung atomarer Modellrealisierungen (Scanner, vgl. Abbildung 7.1).

Der Grapheneditor verwendet eine eigene Symbolsprache, um Modellexemplare und ihre Zugangspunkte sowie die Wege der Datenübertragung darzustellen. Die Knoten im Graphen einer Kopplungsspezifikation sind im Kopplungsdiagramm durch verschiedene Symbole für Modellexemplare und ihre Zugangspunkte repräsentiert. Innerhalb eines Diagramms unterscheidet man durch Farbcodierung zwischen internen und externen Zugangspunkten. Die externen Zugangspunkte stehen als Zugangspunkte des gekoppelten Modells zur Verfügung, die internen Zugangspunkte gehören immer zu einem benutzten Modellexemplar innerhalb des Diagramms. Wenn das Kommunikationsverhalten von externen Zugangspunkten angesprochen wird, so ist immer das Verhalten aus der Sicht der zusammengesetzten Modellkomponente gemeint, und nicht die Sicht eines Nutzers dieser Komponente. Treten externe Zugangspunkte in der zusammengesetzten Modellkomponente als Datensender auf, so erhält die Komponente vom Modellnutzer Daten geliefert, und auch bei aktiven Zugangspunkten geht die Kommunikation vom Modellnutzer aus. Die Sicht auf interne Zugangspunkte ist genau umgekehrt, hier tritt die zusammengesetzte Modellkomponente als Nutzer gegenüber den anderen Modellkomponenten auf.

Linienverbindungen geben die Datenwege zwischen Zugangspunkten an. Sie

repräsentieren die Kanten im Graph der Kopplungsspezifikation. Die Ausführungsumgebung erstellt zu jeder Kante einen Datenpuffer, der auch selbsttätig eine Datenübertragung veranlassen kann und übertragene Daten zwischenspeichert. Das Speicherungsverhalten eines Datenpuffers lässt sich den verschiedenen Übertragungssituationen anpassen, die sich aus den Kommunikationseigenschaften der beiden beteiligten Zugangspunkte ergeben. Auf diese Weise spielt es keine Rolle mehr, ob ein Zugangspunkt aktiv oder passiv bzw. lose oder eng gekoppelt ist. Natürlich können immer nur Datensender mit Datenempfängern verknüpft werden.

Verlassen mehrere Kanten einen Knoten, so wird jeder Wert, der für den Zugangspunkt eines Knotens ermittelt wurde, an alle Datenpuffer der ausgehenden Kanten weitergeleitet (*Multicasting*, s. Abbildung 7.9). Führen mehrere Kanten zu einem Knoten hin, so wird der erste gefundene Wert aller zugeordneten Datenpuffer als neuer Wert des Zugangspunktes übernommen.

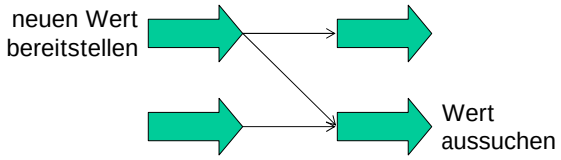


Abbildung 7.9: Multicast

Der Umweg über spezielle Knoten für Berechnungen erlaubt es, die übertragenen Daten zu transformieren. Anstatt eine eigene Sprache zu entwickeln, um die Transformationen anzugeben, verwendet CoCo die Sprache Java. Ein Modellentwickler kann in entsprechenden Textfeldern also direkt Java-Code angeben. Den Zugang zur Ausführungsumgebung stellt CoCo durch ein Rahmenwerk her. Anhang C stellt die Schnittstellen vor, über die ein Modellentwickler auf die Ausführungsumgebung zugreifen kann.

Der Benutzerzugang zum Grapheneditor erfolgt über ein zentrales Fenster, das mehrere Diagramme verwaltet. In Abbildung 7.10 ist ein Diagramm in minimierter Form geöffnet (nur mit der Kopfzeile `[...]/barrels.draw`). Neben verschiedenen Menüpunkten enthält das Fenster eine Werkzeugleiste, mit der Symbole und Linienverbindungen in geöffnete Diagramme eingefügt werden können.

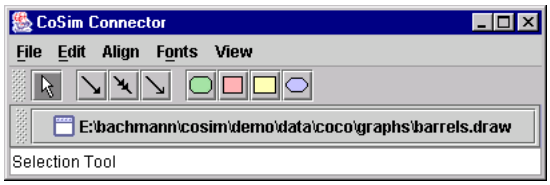


Abbildung 7.10: Grapheneditor

7.3.2 Kopplungsdiagramme

Die Symbolsprache für Kopplungsdiagramme dient dazu, die Datenübertragung zwischen Zugangspunkten und mögliche Transformationen der übertragenen

Daten festzulegen. Dabei geben verschiedene Markierungen innerhalb von Symbolen oder auf Linienverbindungen das Kommunikationsverhalten der einzelnen Zugangspunkte an. Da es nicht vorrangiges Ziel dieser Arbeit ist, eine möglichst ergonomische Nutzungsoberfläche zu entwerfen, wurde auf empirische Tests des Grapheneditors und seiner Symbolik durch Modellentwickler verzichtet. Hier geht es lediglich darum, die Kommunikationseigenschaften von Zugangspunkten im Diagramm zu unterscheiden, um die entsprechenden Einstellungen für die Datenübertragung vornehmen zu können.

Abschnitt 9.5 wird später das entsprechende Kopplungsdiagramm für das Beispielmodell aus Abschnitt 3.1 behandeln. Die folgenden Erläuterungen bereiten also eine Interpretation dieses Diagramms vor.

Knoten

Die Knoten einer Kopplungsspezifikation tragen einen Namen, der innerhalb der Spezifikation eindeutig sein muss. Einen Knoten erkennt man im Kopplungsdiagramm als farbig ausgefüllte Form. Je nach Art des Knotens stehen unterschiedliche Formen zur Verfügung (Abbildung 7.11). Da sich in einer Graustufen-Ansicht einige Farben (rot, grün, blau) kaum unterscheiden lassen, fällt der Codierung von Knotenarten durch verschiedene Formen eine wesentliche Bedeutung zu. Treten gleiche Formen auf, so muss die Farbcodierung auch in einer Graustufen-Ansicht noch kontrastreich sein, was im Falle der gelben (hellen) und roten (dunklen) Rechtecke erfüllt ist.

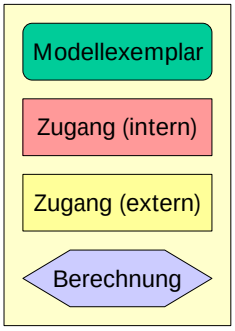
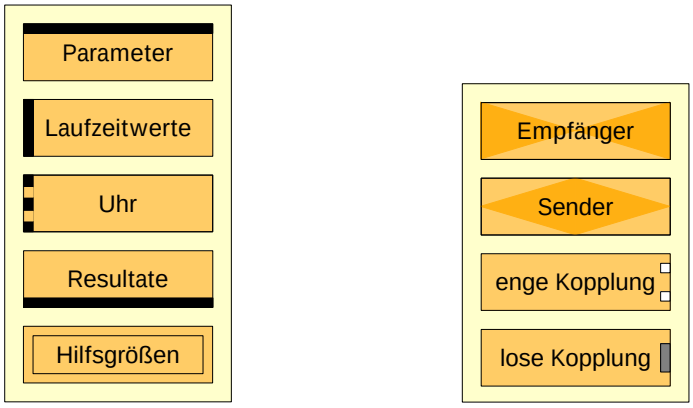


Abbildung 7.11: Knotenarten

Modellexemplare entstehen erst während der Durchführung einer Modellkopplung, hier also durch den Einsatz der Ausführungsumgebung, indem eine Anfrage zur Exemplarerzeugung an eine Modellfabrik ergeht. Bei der Kopplungsspezifikation mit dem Grapheneditor stellt ein grünes abgerundetes Rechteck nun den Platzhalter für ein Modellexemplar dar. Dieses Symbol erhält dann eine Modellfabrik zugeordnet. Die Modellfabrik liefert eine Modellbeschreibung, aus der die Zugangspunkte hervorgehen. Rot gefüllte Rechtecke stehen für die Zugangspunkte der enthaltenen Modellexemplare. Eine gelbe Füllung stellt die Zugangspunkte dar, über die die gerade bearbeitete Modellkopplung später zugänglich sein wird.

Eine Markierung am inneren Rand jedes Zugangspunktes (vgl. Abbildung 7.12(a)) zeigt seine Einordnung in eine Kategorie gemäß Abschnitt 6.4.1 an. Zugangspunkte der Parameterphase (`SETTING`) sind mit einem waagerechten

schwarzen Balken am oberen Symbolrand markiert. In der Resultatsphase (`GETTING`) ist dieser Balken am unteren Symbolrand zu erkennen. Die Laufzeitphase (`RUNNING`) markiert ein senkrechter schwarzer Balken am linken Symbolrand. Eine vorhandene Uhrzeitrepräsentation ist ebenfalls nur in der Laufzeitphase gültig. Dabei ist der zugehörige senkrechte Markierungsbalken in regelmäßigen Abständen unterbrochen. Zugangspunkte, die nur als Hilfsgrößen auftreten, sind markiert durch einen inneren rechteckigen Rahmen um die Symbolbeschriftung herum.



(a) Kategorien (b) Kommunikationsverhalten

Abbildung 7.12: Markierungen von Zugangssymbolen

Im Inneren eines Zugangssymbols gibt es weitere Markierungen, die Informationen zum Kommunikationsverhalten bereitstellen. (vgl. Abbildung 7.12(b)). Zwei abdunkelnde Dreiecke zeigen die mögliche Richtung des Datenflusses an. Sind diese Dreiecke einander zugewandt und zeigen zum Symbolinneren, so entsteht die Form eines „Briefumschlags“ und deutet eine Eingabegröße an. Die beiden Dreiecke einer Ausgabegröße zeigen nach außen und bilden eine Raute. Eine weitere Markierung befindet sich am rechten Rand eines Zugangssymbols. Zwei weiß gefüllte Quadrate zeigen einen Zugangspunkt an, der durch eng gekoppelte Kommunikation angesprochen wird. Bei lose gekoppelten Zugangspunkten erscheint ein grau gefülltes Rechteck. Die dritte Kommunikationseigenschaft (Aktivität) zeigen Markierungen auf ein- und ausgehenden Linienverbindungen an (vgl. Abbildung 7.15).

Um Daten auf dem Weg von einem Zugangspunkt zum anderen zu transformieren, werden Berechnungssymbole eingesetzt, die als blaue Sechsecke zu erkennen sind. Ein Ausdruck in der Programmiersprache Java gibt dann die Transformation an (vgl. Abschnitt 7.3.5). Diese Transformationsausdrücke lassen sich durch Dialoge angeben, die aus dem Diagramm heraus auf doppelten Mausklick hin erscheinen. Auch die Zuordnung von Objektreferenzen für Modellfabriken und Zugangspunkte geschieht per Dialog.

Während das Kopplungsdiagramm bearbeitet wird, lassen sich die Symbole einzelner Knoten oder Gruppen von Knoten per Maus selektieren. Jedes selektierte Symbol trägt temporäre Markierungen, die über sein umschließendes Rechteck hinaus ragen (vgl. Abbildung 7.13). In jeder Ecke dieses Rechtecks befindet sich ein weißes Quadrat, womit sich die Symbole durch Ziehen der

Markierungspunkte in ihrer Ausdehnung verändern lassen.

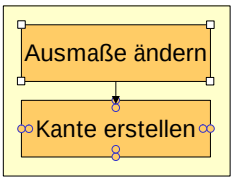


Abbildung 7.13: Markierungen für Interaktionen

In der Mitte jeder Seite des umschließenden Rechtecks geben blaue Kreisränder den Start- bzw. Endpunkt neuer Kanten an. Können mit einem Symbol mehrere Kantenarten verbunden werden, so erscheinen auch entsprechend viele aneinander gereihte Kreisränder. Durch Ziehen aus einem Kreisrand lässt sich eine neue Kante erstellen, wenn die Maustaste über einem gültigen Symbol losgelassen wird. Die neu zu erstellende Verbindung erscheint dabei als „Gummiband“, dessen Ausgangspunkt festgehalten wird.

Kanten

Kanten in einem Kopplungsdiagramm bestehen aus einzelnen geradlinigen Teilstücken, die durch Unterteilungspunkte abgegrenzt werden. Diese Teilungspunkte sind nur sichtbar, wenn die entsprechende Kante selektiert ist. Grundsätzlich werden drei verschiedene Arten von Kanten unterschieden.

- Kanten ohne Markierungen in der Mitte jedes Teilstücks stellen die Zugehörigkeit zwischen Modellexemplaren und ihren Zugangspunkten her (Abbildung 7.14). Auf der Seite des Zugangspunktes befindet sich ein Aktivitätssymbol. Dabei gibt die Dreiecksspitze die Richtung des Datenflusses an. Zeigt die Spitze zu einem Zugangssymbol hin, so tritt der Zugangspunkt als Datensender auf. Bei Datenempfängern zeigt die Spitze vom Symbol weg. Der Datenfluss geht quasi „durch den Zugangspunkt hindurch“.

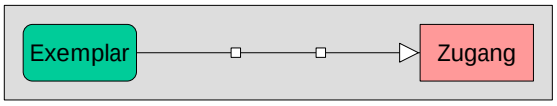


Abbildung 7.14: Zugehörigkeit

- Kanten zur Darstellung des Datenflusses zwischen Zugangspunkten oder Berechnungsknoten besitzen zwei ausgefüllte Dreiecksmarkierungen (Abbildung 7.15). Diese beiden Markierungen befinden sich sowohl im ersten als auch im letzten Teilstück. Sie stellen die Aktivität des hinteren bzw. vorderen Knotens bei der Datenübertragung dar. Eine weiße Füllung bedeutet Aktivität, eine schwarze steht für Passivität. In jedem inneren Teilstück befindet sich ein graues Dreieckssymbol, das die durchgängige Verfolgung des Datenflusses ermöglicht.

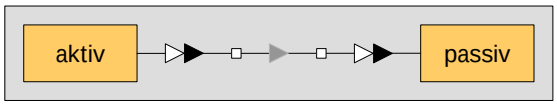


Abbildung 7.15: Datenfluss

- Kontextkanten gehen nur von Berechnungsknoten (blaue Sechsecke) aus und besitzen einen offenen Pfeil in der Mitte jeder Teilstrecke (Abbildung 7.16). Sie werden in zwei Fällen zur Definition von Hilfsgrößen benötigt. Im ersten Fall stellt der Berechnungsknoten neben dem Hauptwert noch weitere Kontextwerte zur Verfügung. Dann ist jedes Zielsymbol einer ausgehenden Kontextkante ein Symbol für einen Zugangspunkt, der in die Kategorie „Hilfsgröße“ eingeordnet wurde. Im Dialog des Berechnungsknotens erscheint deshalb je Zugangspunkt ein weiteres Feld, in das der Ausdruck für den abzuliefernden Hilfswert eingetragen werden kann. Im zweiten Fall ist genau eine Kontextkante vorhanden, die als Ziel einen internen Zugangspunkt hat. Dieser Zugangspunkt besitzt ein Kommunikationsverhalten, das parametrisierte Funktionsaufrufe ermöglicht und tritt als ein passiver, eng gekoppelter Datensender auf (Cage `get`-Variante). Der Berechnungsknoten hat hier die Aufgabe, sämtliche Parameterwerte vor einer Anfrage zusammenzustellen. Je Parameterwert ist wiederum ein Ausdrucksfeld vorhanden.

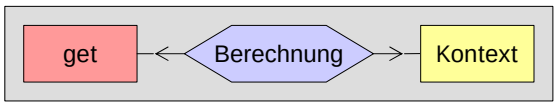


Abbildung 7.16: Kontext

7.3.3 Zuordnungsdialoge

Abschnitt 7.3.2 stellte u.a. die Knoten für Modellexemplare und Zugangspunkte vor. Ein Modellentwickler muss diesen Knoten im Rahmen der Kopplungsspezifikation CORBA-Objektreferenzen zuordnen. Für die Modellexemplare wird die erzeugende Modellfabrik benötigt, und die Zugangspunkte werden direkt mit ihren Symbolen in Verbindung gebracht. CoCo ermöglicht diese Zuordnungen durch verschiedene Dialoge. Alle folgenden Textfelder, in die Objektreferenzen einzutragen sind, akzeptieren `Paste`-Befehle aus einer Zwischenablage. Die Objektreferenzen erlauben neben der direkten IOR-Codierung auch einen URL für die Protokolle `http`, `ftp`, `file`, `iiop` oder `corbaname`. Die zugehörige Schaltfläche öffnet einen Unterdialog, mit dem sich Objektreferenzen aus einer Datei einlesen lassen. Dazu wird der übliche Dateiauswahldialog verwendet. Beide Dialogelemente akzeptieren auch `Drag`-Befehle aus dem Werkzeug `Naming Browser` (vgl. Abschnitt 7.2.1).

Um einem Knoten für Modellexemplare eine erzeugende Modellfabrik zuzuordnen, wird ein spezieller Dialog verwendet (Abbildung 7.17), der verschiedene Objektreferenzen abfragt. Da in der Phase der Kopplungsspezifikation nur die Modellklasse und die Modellabhängigkeiten ausgewertet werden, kann

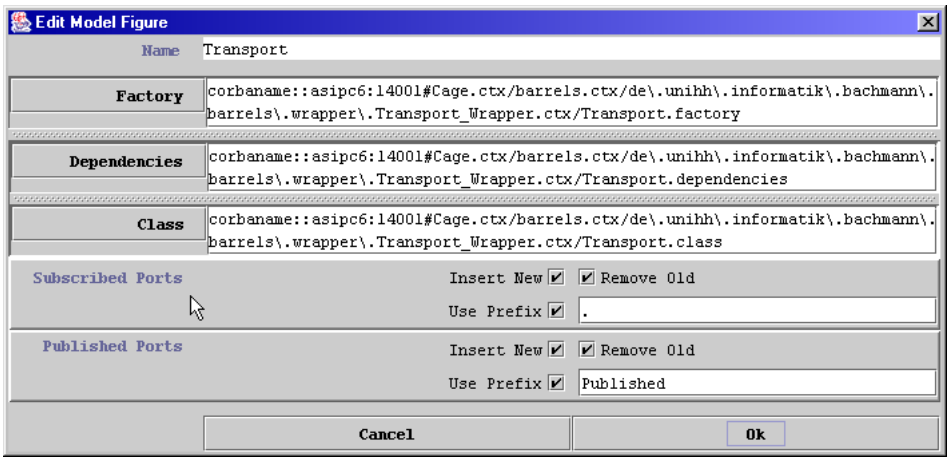


Abbildung 7.17: Zuordnung einer Modellfabrik

das Hauptfeld `Factory` auch leer bleiben. Dann sind die Felder `Class` und `Dependencies` jedoch auszufüllen. Falls die Objektreferenz im Feld `Factory` gültig ist und eine existierende Modellfabrik liefert, so sind die Felder `Class` für die Objektreferenz einer Modellklasse und `Dependencies` für die Objektreferenz der Modellabhängigkeiten ohne Bedeutung, da sich diese Objektreferenzen aus der Modellfabrik ergeben. Diese Felder erhalten automatisch die ermittelten Objektreferenzen zugewiesen.

Das Ankreuzfeld `Insert New` gibt an, ob in das Kopplungsdiagramm alle Knoten für Zugangspunkte eingefügt werden sollen, die noch nicht im Diagramm vorhanden sind. Genauso benutzt man das Ankreuzfeld `Remove Old`, um nicht mehr benötigte Zugangspunkte aus dem Diagramm zu entfernen. Die entsprechenden Kanten einzufügender oder zu löschender Knoten, die vom Knoten dieser Modellfabrik ausgehen sollen, löscht bzw. erzeugt CoCo automatisch mit. Diese Automatik erspart dem Modellersteller viel Arbeit. Er muss die neuen Knoten und Kanten nur noch im Diagramm anordnen.

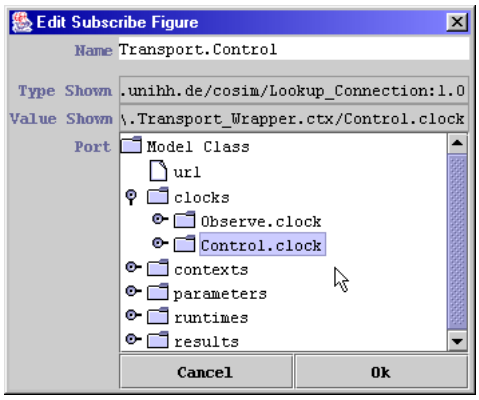


Abbildung 7.18: Zuordnung eines internen Zugangspunktes

Interne Zugangspunkte stehen niemals für sich allein, sondern ergeben sich

aus einer Modellklasse, die ein vorhandenes Modellexemplar beschreibt. Ein interner Zugangspunkt lässt sich dem entsprechenden Knoten nur dann zuordnen, wenn bereits eine Kante zu einem Knoten eines Modellexemplars vorhanden ist und dort eine existierende Modellklasse ermittelt wurde. Sind diese Bedingungen erfüllt, so erscheint im Feld `Port` eine Ansicht der Modellklasse als Baumstruktur. Die zweite Ansichtsebene zeigt die Kategorien für Zugangspunkte (`clocks`, `contexts`, `parameters`, `runtimes` oder `results`) als Ordner an. Innerhalb eines Ordners muss nun genau ein Zugangspunkt ausgewählt werden. Diese Ansicht gleicht der Ansicht im Werkzeug Lookup Generator (vgl. Abschnitt 7.2.2). Auch die Felder `Value Shown` und `Type Shown` stellen die gleichen Detailinformationen zu aktuell ausgewählten Blättern dar.

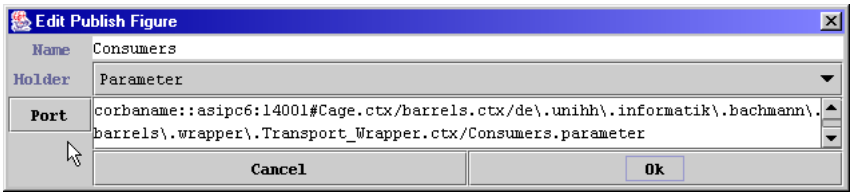


Abbildung 7.19: Zuordnung eines externen Zugangspunktes

Beschreibungen für externe Zugangspunkte lassen sich mit dem Grapheneditor nicht erstellen. Hierfür ist wiederum das Werkzeug Lookup Generator aus Abschnitt 7.2.2 geeignet. Der Editor erlaubt nur die Zuordnung einer vorher erstellten Objektreferenz, die im Feld `Port` einzutragen ist. Zusätzlich muss noch in der Auswahlliste `Holder` angegeben sein, unter welcher Kategorie diese Objektreferenz einzuordnen ist. CoCo erzeugt für jeden externen Zugangspunkt im Rahmen der `.coco`-Datei eine Kopie, damit spätere Änderungen an der ursprünglichen Definition keinen Einfluss mehr auf die zusammengestellte Komponente haben.

7.3.4 Speicherungsverhalten eines Datenpuffers

Mit jeder Kante zwischen zwei Knoten eines Kopplungsdiagramms ist ein Datenpuffer verknüpft, der übertragene Daten zwischenspeichert und in FIFO-Reihenfolge wieder abliefert. Um die Arbeitsweise eines Datenpuffers genauer festzulegen, lassen sich verschiedene Einstellungen vornehmen.

Zunächst einmal kann der Puffer eine begrenzte Kapazität zugeschrieben bekommen (vgl. Abbildung 7.21(b)). Falls eine begrenzte Kapazität verwendet wird, muss auch angegeben sein, wie mit Werten zu verfahren ist, bei deren Eintreffen die Kapazitätsgrenze bereits erreicht ist. Diese Werte können verworfen oder in den Datenpuffer aufgenommen werden. Soll der Wert aufgenommen werden, fällt der erste Wert aus dem Datenpuffer heraus. Sobald der Puffer einmal einen Wert enthält, kann der Puffer dafür sorgen, dass immer ein Wert für aktive Empfänger zur Abfrage vorhanden ist, indem der letzte vorhandene Wert im Puffer stehen bleibt (Ankreuzfeld `Hold` in Abbildung 7.21(b)).

Abbildung 7.20 zeigt alle möglichen Übertragungsvarianten mit der Beschriftung für Übertragungsmethoden aus dem Werkzeug Cage. Die passiven Knoten benötigen meist eine besondere Behandlung. Bei passiven Empfängern muss der Datenpuffer für den Aufruf der entsprechenden Empfangsmethode sorgen, wenn

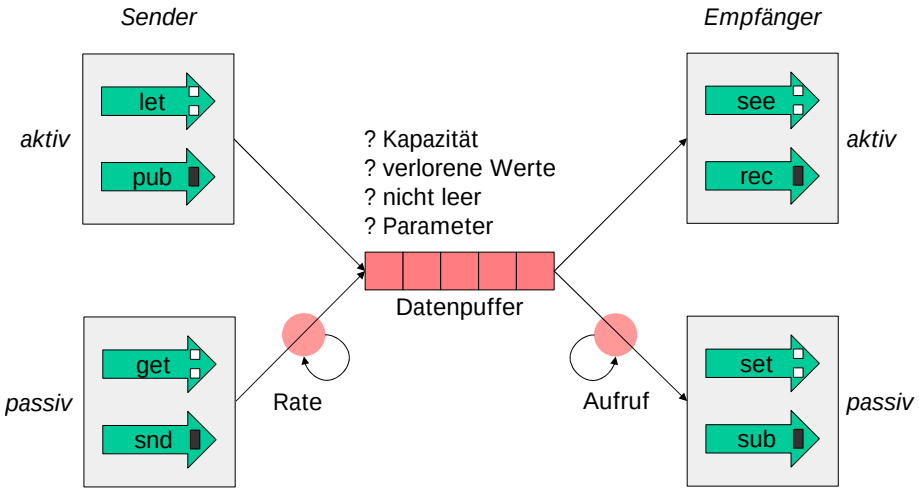
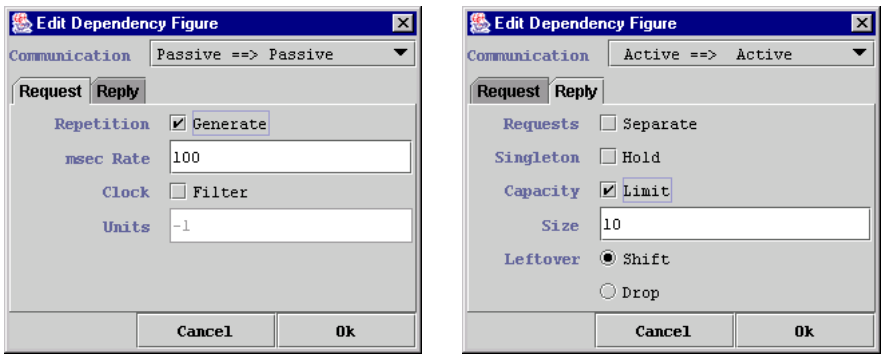


Abbildung 7.20: Datenpuffer

neue Datenwerte vorliegen. Passive Sender verlangen eine Aktivität des Datenpuffers, um die entsprechenden Sendemethoden aufzurufen. Sind sowohl Sender als auch Empfänger passiv, muss der Datenpuffer selbsttätig Anfragen erzeugen. Durch das Eingabefeld `Rate` des Dialogs in Abbildung 7.21(a) lassen sich die Anfragen in regelmäßigen Realzeit-Abständen einstellen. Ein Filter sorgt dafür, dass keine Antworten in den Puffer gelangen, die nicht mindestens die angegebene Anzahl an Zeiteinheiten der Simulationszeit auseinander liegen. Dieses Verhalten ist auch bei aktiven Sendern nützlich, die sehr häufig Zeitstempel mit geringfügigen Unterschieden liefern.



(a) `request` (b) `reply`

Abbildung 7.21: Einstellungen eines Datenpuffers

Ist ein eng gekoppelter Empfänger aktiv, so stellt er parametrisierte Anfragen. Falls also mindestens ein Parameter eingesetzt wurde, sollte der Datenpuffer auch nur solche Antworten ausliefern, die zu den übergebenen Parameterwerten gehören. Dieses Verhalten steht durch das Ankreuzfeld `Separate` in Abbildung 7.21(b) zur Verfügung. Wenn sich bei zwei eng gekoppelten Zugangspunkten die

eine Seite aktiv, die andere passiv verhält, könnten prinzipiell auch die Modelle direkt bei der gegenüberliegenden Stelle angemeldet werden. Dann wird der Datenpuffer vollständig umgangen und ein möglicher Flaschenhals bei der Datenübertragung eliminiert. Diese Variante bietet die aktuelle CoCo Realisierung jedoch noch nicht.

7.3.5 Konvertierungsausdrücke

Damit einfache Konvertierungsfunktionen ad hoc benutzbar sind, und nicht als komplette Modellfabrik vorhanden sein müssen, erlaubt es CoCo, entsprechende Konvertierungsausdrücke zu formulieren. Diese Ausdrücke treten typischerweise für mehrfach verwendete (konstante) arithmetische Ausdrücke, als Umformung oder als Verknüpfung auf (vgl. Abbildung 7.22).

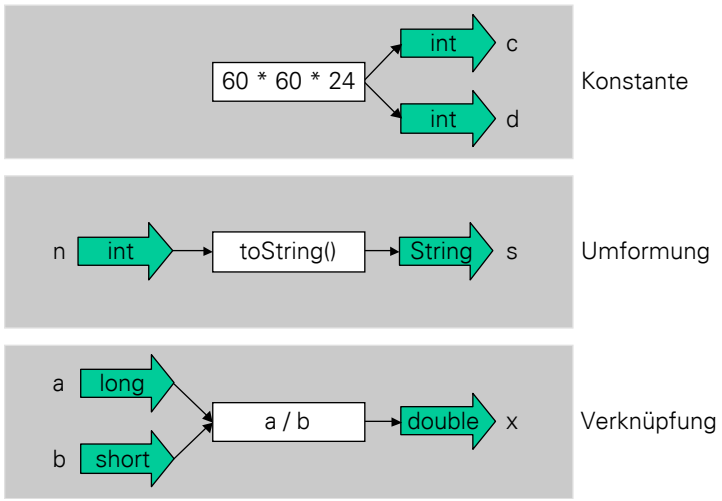


Abbildung 7.22: Konvertierungsausdrücke

Die Definition einer Konvertierungsfunktion besteht grundsätzlich aus mehreren Teilen, genauso wie jeder übertragene Wert aus mehreren Teilen besteht. CoSim verwendet nach Abschnitt 6.6 zur Datenübertragung die zusammengesetzte Datenstruktur `Guided_Value`, die aus einem Hauptwert und beliebig vielen Hilfswerten besteht. Der Hauptteil einer Konvertierungsfunktion in CoCo gibt eine Funktion für den Hauptwert `simple` dieser Datenstruktur an, und jeder benötigte Hilfswert definiert einen weiteren Teil, für den eine entsprechende Funktion anzugeben ist. Der Modellersteller legt den CORBA-Datentyp des Hauptteils explizit fest. Die Datentypen der Hilfswerte sind durch die entsprechenden Hilfsvariablen bereits vorgegeben.

CoCo benutzt Java zur Definition von Konvertierungsfunktionen. Für die Berechnungsknoten der Kopplungsspezifikation lässt sich eine Funktion entweder als Java-Text oder als Java-Klasse angeben. Bei der Eingabe als Java-Text muss der Reiter `Text` im Knotendialog aktiv sein (vgl. Abbildung 7.23). Nun kann jeder einzelne Definitionsteil mit der Auswahlliste `Part` eingestellt und getrennt bearbeitet werden. Eine korrekte Definition spiegelt den Rumpf einer Java-Methode (ohne geschweifte Klammern) wider und endet mit einer `return`-

Anweisung. Jede Methode bekommt genau einen Parameter übergeben, nämlich eine Umgebungsdefinition, aus der sich die aktuellen Anfrageparameter extrahieren lassen. Die Werte für die Anfrageparameter liefern die eingehenden Knoten des aktuellen Berechnungsknotens. Auch die eingehenden Knoten liefern wieder Hauptwerte und Kontextwerte. Alle benutzbaren Teile sind im Feld `Imports` aufgelistet und lassen sich mit dem Knopf `Insert Expression` in den Definitionstext an der aktuellen Cursorposition einsetzen. In der Abbildung 7.23 sind ein `Import` Eintrag und der eingefügte Text markiert.

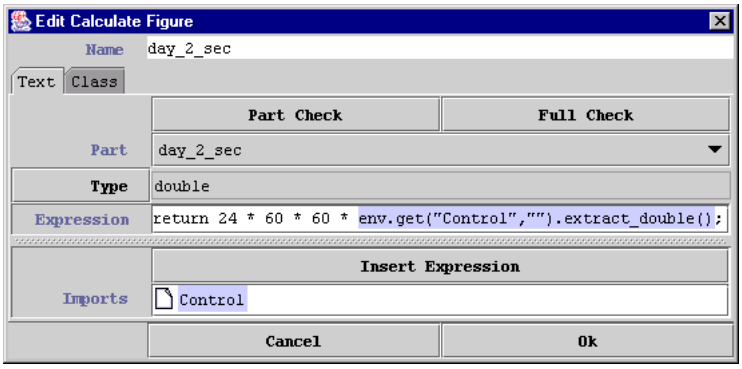


Abbildung 7.23: Datentransformation per Ausdruck

Der Kopf der Java-Funktion, die die Haupt- und Kontextwerte berechnet, steht mit der Angabe des Rückgabedatentyps bereits fest und wird vor jeder Ausdrucksprüfung automatisch generiert (`Part Check` oder `Full Check`). Der Rückgabetyp entspricht dem Java-Mapping (OMG 2001b) für den eingestellten CORBA-Typ. So wird beispielsweise für den CORBA-Typ `double` das folgende Methodengerüst erzeugt:

```
public double calculate_123 (Environment env) {;;;}
```

Benutzerdefinierte IDL-Datenstrukturen bilden hier eine Ausnahme: Da diese Typen im Allgemeinen noch nicht existierten, als die CoCo-Umgebung erstellt wurde, muss der Modellentwickler direkt mit CORBA-Datenstrukturen umgehen. Dazu wird die typunabhängige Hülle `Any` als Rückgabedatentyp eingesetzt, in die sich beliebige Datenwerte verpacken lassen. Der Entwickler eines Definitionsteils muss dann sicherstellen, dass dieses `Any`-Konstrukt einen Wert vom tatsächlich erwarteten Typ erhält.

Für die Eingabe von Transformationsfunktionen als Java-Klasse muss der Reiter `Class` im Knotendialog aktiv sein (vgl. Abbildung 7.24). Nun kann eine Klasse in der Auswahlliste `Class Name` eingestellt oder eingegeben werden. Die gewählte Klasse muss die Schnittstelle `de.unihh.informatik.bachmann.coco.calculation.Calculation_Value` (Anhang C) implementieren. Für den Spezialfall einer Definition, die ausschließlich als Kontext für parametrisierte Anfragen benutzt wird (vgl. `get` in Abbildung 7.16), genügt auch die Implementation der Schnittstelle `de.unihh.informatik.bachmann.coco.calculation.Calculation_Context`). Über diese Schnittstellen deklariert die Transformationsklasse, welche Datentypen sie abliefern wird, und ob sie dabei als aktiver oder passiver Sender agieren wird. Die Deklaration erfolgt dabei getrennt je ausgehender Kante. Zur Unterscheidung der Kanten dienen die Namen der Zielknoten ausgehender Kanten. Nach der Initialisierung erhält jedes neue Objekt

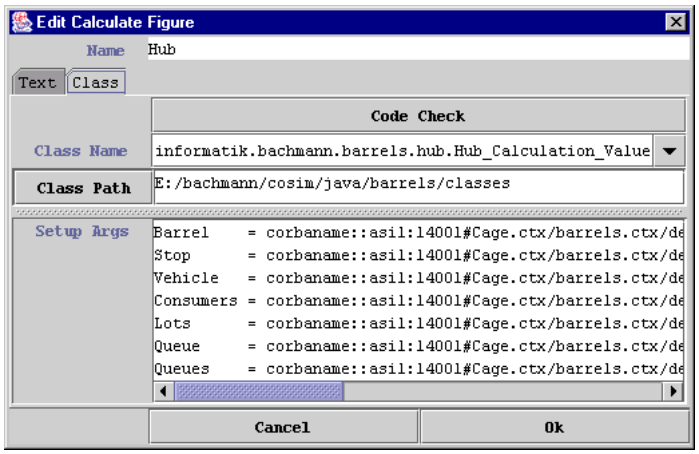


Abbildung 7.24: Datentransformation per Klasse

der Transformationsklasse den Inhalt des Feldes `Setup Args` durch die Methode `setup_calculation`. Somit können auch allgemeine Transformationsklassen eingesetzt werden, die erst im Rahmen der Kopplungsspezifikation eine Anpassung erfahren. In Abbildung 7.24 beispielsweise benötigt die dargestellte Beladefunktion (Abschnitt 9.4) die Typbeschreibungen für die CORBA-Objekte, die über die Datenpuffer versendet oder empfangen werden. Diese Typen ermittelt die Funktion aus den Zugangspunkten, die als Parameter übergeben werden. Sie kann dann feststellen, ob die eingehenden und ausgehenden Knoten auch typkonforme Daten liefern.

Die CoCo-Umgebung stellt zwei vordefinierte Klassen zur Verfügung: eine Simulationsuhr und einen Multicast-Mechanismus. Die Simulationsuhr berechnet einfach das Minimum aller eingehenden Knoten. Sie erhält als Konfigurationseinstellung den Initialwert sowie den Ausgabedatentyp. Als Eingabe- und als Ausgabedatentyp können alle Zahltypen eingesetzt werden. Für die Konvertierung sorgt die Uhr selbsttätig. Die Multicast-Klasse realisiert genau das eingangs in Abschnitt 7.3.1 angesprochene Verhalten. Bei einer Anfrage nach einem neuen Wert für diesen Berechnungsknoten werden alle angeschlossenen Knoten nach einem Wert aus dem zugehörigen Datenpuffer abgefragt. Die erste Antwort ergibt den Wert der Anfrage. Alle anderen Antworten verbleiben in ihren jeweiligen Datenpuffern und stehen für nachfolgende Anfragen zur Verfügung.

Falls die gewählte Klasse nicht Teil der CoCo-Umgebung, sondern extern definiert ist, können zusätzliche Angaben im Feld `Class Path` notwendig sein (vgl. Abschnitt 7.3.7).

7.3.6 Konsistenzregeln eines Kopplungsgraphen

Damit ein Kopplungsdiagramm erfolgreich in die Zwischenrepräsentation als `.coco`-Datei übersetzt werden kann, muss das Diagramm einige Konsistenzregeln erfüllen. Wenn im Folgenden ein aktiviertes CORBA-Objekt angesprochen wird, so heißt das, die Implementation ist gestartet und beantwortet Anfragen an das Objekt.

Allgemeines

- Alle vergebenen Bezeichner für die Knoten eines Diagramms dürfen nur einmal vorkommen, damit die Fehlermeldungen und Warnungen der Konsistenzprüfung bei einer Textausgabe den betroffenen Knoten zugeordnet werden können.

Modellexemplar

- Jedem Modellexemplar muss eine Modellklasse als aktiviertes CORBA-Objekt zugewiesen sein. Die Modellklasse kann auch aus einer Modellfabrik ermittelt werden.
- Jedem Modellexemplar müssen Modellabhängigkeiten als aktiviertes CORBA-Objekt zugewiesen sein. Die Modellabhängigkeiten können auch aus einer Modellfabrik ermittelt werden.
- Jedem Modellexemplar muss eine Modellfabrik zugewiesen sein. Falls die Modellfabrik zum Zeitpunkt der Speicherung als `.coco`-Datei nicht aktiviert ist, so wird eine Warnung ausgegeben und die Datei trotzdem erstellt. Sobald jedoch die `.coco`-Datei eingelesen wird, um eine neue Modellfabrik zu erzeugen, müssen die Modellfabriken der eingesetzten Modellexemplare aktiviert sein.

Interne Zugangsknoten

- Ausgabegrößen erlauben keine eingehenden Datenflüsse.
- Eingabegrößen erlauben keine ausgehenden Datenflüsse.
- Jeder Zugangsknoten muss durch genau eine Zugehörigkeitskante mit einem Modellknoten verbunden sein.
- Es muss ein aktivierter Zugangspunkt ausgewählt sein.
- Eng gekoppelte, passive Ausgabegrößen verlangen eine Kontextdefinition, um die Parameter des zu Grunde liegenden Prozeduraufrufs anzugeben. Es muss daher genau eine eingehende Kontextkante existieren, die zu einem Berechnungsknoten führt.

Externe Zugangsknoten

- Ausgabegrößen erlauben keine ausgehenden Datenflüsse.
- Eingabegrößen erlauben keine eingehenden Datenflüsse.
- Es muss ein aktivierter Zugangspunkt zugeordnet sein.

Berechnungen

- Definitionen als Text enthalten einen Java-Methodenrumpf, der mit einer `return` Anweisung endet. Der Methodenrumpf muss zusammen mit dem automatisch erzeugten Methodengerüst durch einen Java-Compiler übersetzbar sein.

- Definitionen als Klasse müssen über den angegebenen Klassenpfad ladbar sein.
- Die definierte Klasse muss die Typen der Datenwerte akzeptieren, die von eingehenden Knoten erzeugt werden. Die Schnittstellen der Klassendefinitionen enthalten für diesen Zweck eine Methode `check_calculation`. Für Text-Definitionen erfolgt diese Prüfung bereits durch den Java-Compiler.

Datenfluss

- Zyklen im Datenfluss führen zu Warnungen, aber die `.coco`-Datei wird trotzdem erstellt. Um Zyklen zu erkennen, die sich über mehrere Modellexemplare erstrecken, finden die angegebenen Modellabhängigkeiten der zugehörigen Modellfabriken ihre Anwendung.
- Die Typen von Sende- und Empfangsknoten müssen übereinstimmen. Die Typprüfung erfolgt auf der Basis des CORBA-Typ-Operators `_is_equivalent`.
- Für eine beidseitig passive Datenübertragung muss eine positive Abfragerate angegeben sein.

7.3.7 Ausführung von Kopplungsdiagrammen

Aus dem Menüpunkt `File/Generate` lässt sich die Übersetzung eines Kopplungsdiagramms in eine `.coco`-Datei als Zwischenrepräsentation anstoßen. Dabei werden zunächst die Konsistenzregeln aus Abschnitt 7.3.6 überprüft. Fällt diese Prüfung erfolgreich aus, sind noch einige beschreibende Daten zu der neuen Modellfabrik anzugeben. Da mit der neuen Modellfabrik auch eine neue Modellbeschreibung entsteht, werden für beide Objekte ein Name und ein URL verlangt. Der URL ist jeweils für weitere Hinweise zur Modellnutzung gedacht. Der Name regelt den Eintrag im CORBA Namensdienst. Außerdem ist anzugeben, unter welchem Dateinamen die `.coco`-Datei zu speichern ist. Diese Datei liest die Ausführungsumgebung ein und interpretiert die enthaltene Kopplungsspezifikation als neue Modellfabrik.

Zwischenrepräsentation

Die Zwischenrepräsentation verwendet das `JAR` Archivformat, um verschiedene Daten in einer `.coco`-Datei zusammenzufassen. Die einzelnen Teile lassen sich also mit gängigen `JAR` oder `ZIP` Archiv-Werkzeugen extrahieren. Hauptbestandteil des Archivs ist eine per Java Serialization gewonnene Form des Kopplungsgraphen als Eintrag mit dem Namen `graph.ser`. Wurzel der abgelegten Datenstruktur ist ein Objekt der Klasse `de.unihh.informatik.bachmann.coco.writer.Checked_Graph`.

Weiterhin sind im Archiv alle Berechnungsdefinitionen des Kopplungsgraphen abgelegt, die als Texte angegeben wurden. Je Berechnungsknoten existiert eine generierte Klasse sowohl als übersetzter Java-Bytecode als auch im Quellcode. Die Namen der erzeugten Klassen setzen sich zusammen aus dem Präfix `de.unihh.informatik.bachmann.coco.evaluation.Calculation_` und einem Hashcode. Die erzeugten Methodennamen folgen einem ähnlichen Muster. Das Präfix

ist hier `calculate_`, gefolgt von einem Hashcode. Die Klassen- und Methodennamen aus einem Kopplungsgraphen unterscheiden sich in jeder neu erzeugten `.coco`-Datei nur, falls neue Symbole für die beteiligten Knoten einer Definition erzeugt wurden.

Klassenpfade

Beim Start erhält die CoCo-Ausführungsumgebung ein zentrales *Datenverzeichnis* zugeordnet, genauso wie Cage zur Einbettung von Java basierten Modellrealiserungen (vgl. Abschnitt 7.1.6). Auch hier existiert das Unterverzeichnis `extensions` für `.jar` und `.zip` Bibliotheken, die von mehreren Kopplungsspezifikationen aus nutzbar sind. Damit die Berechnungsknoten eines Kopplungsdiagramms eine eingestellte Klassendefinition auch laden können, stellt die Ausführungsumgebung für jeden Berechnungsknoten eine Hierarchie von Klassenpfaden zusammen:

- Klassenpfade des CoCo-Scanner-Aufrufs
- Alle `.zip` oder `.jar` Bibliotheken im `extensions` Verzeichnis
- Die `.coco` Datei selbst
- Das Verzeichnis, in dem die `.coco` Datei gefunden wurde
- Alle `.zip` oder `.jar` Bibliotheken in dem Verzeichnis der `.coco` Datei.
- Klassenpfade aus der Definition des Knotens

CoCo ordnet dann jedem Berechnungsknoten einen eigenen Java `ClassLoader` zu, der die angegebenen Klassenpfade verwendet.

Ausgaben

Für jede geladene `.coco`-Datei erzeugt die Ausführungsumgebung Ausgaben im lokalen Dateisystem und in einem CORBA Namensdienst. Dabei wird ein *Modellpräfix* verwendet, ähnlich wie es für Cage in Abschnitt 7.1.7 beschrieben ist, um die Kopplungsspezifikationen eindeutig zu unterscheiden. Hier ergibt sich das Modellpräfix aus dem vollständigen Pfad der `.coco`-Datei im lokalen Dateisystem, vermindert um den Pfad für das Datenverzeichnis (vgl. Tabelle 7.5). Um Einträge bei einem CORBA Namensdienst zu erzeugen, erhält die CoCo Ausführungsumgebung beim Start einen zentralen *Datenkontext* zugeordnet. Mit diesen Einstellungen entstehen folgende Ausgaben:

- Ein neuer CORBA NamingContext unterhalb des Datenkontextes mit dem Namen *`Datenkontext/Modellpräfix`*`.ctx`.
- Drei Einträge im neuen Unterkontext: Unter dem Namen der Modellfabrik erscheint die Modellfabrik selbst mit der Erweiterung `.factory`. Die Modellabhängigkeiten erhalten den gleichen Namen mit der Erweiterung `.dependencies`. Für die Modellklasse wurde ein eigener Name vergeben, der um die Endung `.class` erweitert wird.
- Ein Eintrag für jeden externen Zugangsknoten: CoCo kopiert die beschreibenden Informationen jedes externen Zugangspunktes, damit Änderungen nach Erzeugung der `.coco` Datei keine Inkonsistenzen auslösen können.

- Je eine Protokoll-Datei für Fehlermeldungen und Modellmeldungen: Die Daten befinden sich im Unterverzeichnis *Datenverzeichnis*/logs und heißen *Modellpräfix*.err bzw. *Modellpräfix*.out.
- Unterhalb des Datenverzeichnisses existieren außerdem die Unterverzeichnisse archive, classes, defined, loading und sources. Sie können vom Grapheneditor temporär erzeugte Dateien enthalten und werden bei jedem Start des Editors bereinigt.

Pfad	Modellpräfix
/Datenverzeichnis/modell_a.coco	modell_a
/anderes_Verzeichnis/modell_a.coco	*/anderes_Verzeichnis*/modell_a

Tabelle 7.5: Ermittlung des Modellpräfix

7.4 Durchführung von Simulationsläufen

Simulationsmodelle werden in der Regel erstellt, um im Rahmen von Simulationsexperimenten eine Auswertung des Modellverhaltens vorzunehmen. Teilweise sind dazu sehr viele Simulationsläufe notwendig. Eine Experimentierumgebung ermöglicht den Modellnutzern, diese Menge von Simulationsläufen durchzuführen und komfortabel zu verwalten. Mit dem CoSim-Ansatz entstand auch ein Konzept für die Durchführung von Simulationsexperimenten, das in (Schöllhammer 2001) als Werkzeug mit dem Namen „ExpU“ prototypisch umgesetzt wurde.

Durch die Verwendung der ExpU muss sich ein Modellnutzer nicht mehr mit den CORBA-Schnittstellen von Modellfabriken und -exemplaren beschäftigen. Stattdessen erlaubt eine grafische Oberfläche den modellunabhängigen Zugang zu Simulationsexperimenten. Das Konzept der Experimentierumgebung setzt dabei die Forderung nach einer Trennung von Modell und Experiment (Zeigler 1984) unmittelbar um, d.h. Experimente lassen sich unabhängig von Modellimplementationen definieren und verwalten. Diese Trennung wird auch direkt von der CoSim-Architektur gefördert, da sämtliche Modellkomponenten über eine einheitliche Schnittstelle zugänglich sind. Dadurch können Szenarien, die nach Seila und Miller (1999) aus Eingabeparametern und Auswertefunktionen bestehen, bereits vor der Erstellung irgendeines Modells festgelegt werden. Bleibt das Anwendungsszenario fest, so lassen sich Konkurrenzmodelle einfach auswerten, in dem ein Experiment verschiedene Modellfabriken verwendet, womit auch die in (Hilty u. a. 1998, Kap. A.4) angestrebte diskursorientierte Simulation möglich wird, um beispielsweise die Konsensfindung bei strittigen Modellierungsentscheidungen voranzutreiben.

Für die Konzeption einer Experimentierumgebung gilt bzgl. der Nutzungsschnittstelle Ähnliches wie für das Kopplungswerkzeug CoCo in Abschnitt 7.3. Auch hier kann man zwischen Programmierung durch den Modellbenutzer und Unterstützung durch Dialoge einer grafischen Nutzungsschnittstelle abwägen. Zwischenlösungen, wie z.B. spezielle Experimentiersprachen, sind ebenfalls möglich. Die Konzeption der ExpU sieht einerseits eine grafische Schnittstelle vor, um den Einarbeitungsaufwand für Modellbenutzer möglichst gering zu halten.

Anderseits enthält die ExpU auch Schnittstellen, über die neue Auswertemechanismen für spezielle Simulationsstudien in das Rahmenwerk integriert werden können.

Die einzelnen Teilbereiche der ExpU ergeben sich hauptsächlich aus den Arbeiten von Langer (1989), Dörnhöfer (1991) und Wittmann (1993), die Anforderungen an Experimentierumgebungen ausführlich diskutieren. Diese Anforderungen fanden schließlich eine Umsetzung im Rahmen des Simulationssystems Simplex3 (Schmidt 2000). So sieht die ExpU dann auch die wesentlichen Merkmale einer Experimentierumgebung wie Experimentvariablen, Parameterbelegungen und -variationen, Beobachtung und Filterung von Laufzeitgrößen sowie die Steuerung der einzelnen Simulationsläufe vor.

Basis einer Experimentierumgebung ist immer ein Datenhaltungssystem, in dem sich Experimentspezifikationen und die aus einem Experiment gewonnenen Simulationsergebnisse dauerhaft ablegen lassen (Herren u. a. 1997). Die Speicherung der Ergebnisse, zusammen mit einer ausführlichen Dokumentation, ist hier sehr wichtig, weil nicht gewährleistet werden kann, dass sich die Simulationsläufe nach längerer Zeit erneut durchführen lassen. Die verwendeten Modellfabriken liegen in der Regel ja nicht lokal vor, und zu ihrer Implementation besteht wegen des Blackbox-Prinzips auch kein Zugang.

7.4.1 Experimentvariablen

Damit die Definition von Experimenten wirklich unabhängig von existierenden Modellrealisierungen möglich wird, dürfen in der Experimentdefinition keine direkten Bezüge zu den Zugangspunkten aus einem bestimmten Modell existieren. Die ExpU führt dazu spezielle Experimentvariablen ein. Die Experimentvariablen sind vergleichbar mit den Zugangspunkten im Modell, sie werden jedoch nur für die ExpU verwendet. Ein Modellbenutzer definiert ein Experiment ausschließlich auf der Basis von Experimentvariablen. Für jede Modellfabrik, die innerhalb eines Experiments benutzt wird, erstellt er eine Zuordnung von Experimentvariablen zu korrespondierenden Zugangspunkten im Modell. Abbildung 7.25 zeigt die Zuordnung, die im ExpU-GUI durch eine einfache Tabelle realisiert ist.

Variable Kind	Lookup Name	Variable Name
parameter	Pressure.Fill.1	Fill.1
parameter	Pressure.Fill.2	Fill.2
parameter	Pressure.Fill.3	Fill.3
parameter	Initial	Init
parameter	Vehicles.Capacity	Capacity
parameter	Strategy.Vehicle	Vehicle
parameter	Stop	Stop
parameter	Vehicles.Count	Count
parameter	Strategy.Barrel	Barrel
parameter	Consumers	Consumers
result	Waiting	Waited
result	Returns	Returned
result	Filling	Filled
result	Deliveries	Delivered

Abbildung 7.25: Experimentvariablen

7.4.2 Parametereingabe

Die ExpU ist eine generische Anwendung, genauso wie Cage und CoCo. Sie kann also mit Datentypen umgehen, die zum Zeitpunkt der Werkzeugimplementation noch nicht existierten. Besonders wichtig ist diese Eigenschaft bei der Versorgung von Modellexemplaren mit Parameterwerten. Die Modellexemplare einer Modellfabrik können spezielle Datenstrukturen verlangen, nach denen die übergebenen Parameterwerte aufgebaut sein müssen. Die ExpU erlaubt deshalb auch den Aufbau komplexer Datenstrukturen und verwendet hier denselben Mechanismus wie der Lookup Generator zur Erzeugung von Objekthüllen (vgl. Abschnitt 7.2.2). Dort besteht bereits die Möglichkeit, beliebige Datenwerte für Initialwerte von Zugangspunkten anzugeben. Die ExpU-Implementation verwendet in diesem Fall einfach den in Abschnitt 8.2 vorgestellten Java-Code wieder.

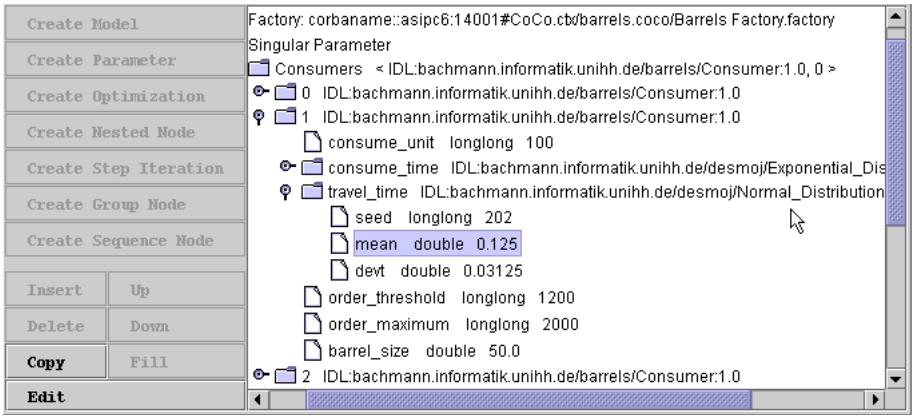


Abbildung 7.26: Parameterspezifikation

Die einzelnen Parameter finden sich in einer übergeordneten Baumstruktur wieder, die verschiedene Möglichkeiten bietet, um Gruppen von Simulationsläufen anzugeben. Abbildung 7.26 zeigt die zugehörigen Schaltflächen mit dem Präfix „Create“ auf der linken Seite, die neue Knoten in die Baumstruktur einfügen. Die ersten beiden Einträge (`Create Model` und `Create Parameter`) erzeugen Blätter in der Baumstruktur. Unter diesem Knoten lassen sich also keine weiteren Knoten anordnen. Das in einem Simulationslauf verwendete Modellexemplar betrachtet die ExpU einfach als speziellen Parameter, der für jeden Lauf notwendig ist und als Belegung die Objektreferenz einer Modellfabrik verlangt.

Die nachfolgenden Einträge (`Create Optimization` bis `Create Sequence Node`) fügen innere Knoten ein, d.h. hier können Verästelungen vorkommen. Die Verästelungen beschreiben, wie sich ein komplexes Experiment aus Schleifen von Simulationsläufen zusammensetzt. Die einfachste Struktur stellt die Gruppe dar (`Create Group Node`). Sie fasst mehrere Einstellungen zusammen, die gemeinsam vor dem nächsten Simulationslauf vorzunehmen sind. Die Sequenz (`Create Sequence Node`) führt hingegen nach jeder untergeordneten Einstellung einen Simulationslauf durch. Dabei lässt sich festlegen, ob die einzelnen Läufe tatsächlich nacheinander erfolgen sollen, oder ob sie unabhängig und nebenläufig ausgeführt werden dürfen. Eine spezielle Variante der Sequenz steht mit dem

Schrittweitenmodus zur Verfügung (`Create Step Node`). Hier gibt der Experimentersteller für Parameter mit numerischem Datentyp Start und Zielwert sowie eine Schrittweite vor. Diesen Parameter variiert die ExpU dann in äquidistanten Schritten solange, bis der Zielwert erreicht ist. Mit jedem Schritt erfolgt dann auch ein Simulationslauf. Die Verschachtelung (`Create Nested Node`) führt für jeden untergeordneten Knoten eine neue Schleife ein. Die untergeordneten Knoten beschreiben dabei eine Folge von Simulationsläufen, beispielsweise durch schrittweise Variation. Diese Folgen von Simulationsläufen erfolgen geschachtelt, wobei der unterste Knoten die innerste Schleife definiert, also am häufigsten durchlaufen wird.

Die Optimierung (`Create Optimization`) stellt ein komplexes Variationsschema bereit. Im allgemeinen Fall benennt der Experimentersteller hier eine Menge von Parametern mit numerischem Datentyp, die gemeinsam variiert werden dürfen. Das Optimierungsverfahren sucht dann nach einer Parameterkombination, die den Wert einer vom Experimentersteller vorgegebenen Zielfunktion maximiert bzw. minimiert, wobei wiederum vom Experimentersteller vorgegebene Randbedingungen einzuhalten sind. Während dieser Suche entscheidet das Optimierungsverfahren selbst, welche Folge von Simulationsläufen unter welchen Parameterkombinationen durchgeführt werden soll. Um verschiedene Optimierungsverfahren einfließen zu lassen, sah der ursprüngliche Entwurf ein Rahmenwerk ähnlich wie in (Gehlsen und Page 2001) vor. Die weitere Umsetzung bzw. eine Verwendung des angegebenen Rahmenwerks konnte jedoch aus Zeitgründen nicht erfolgen.

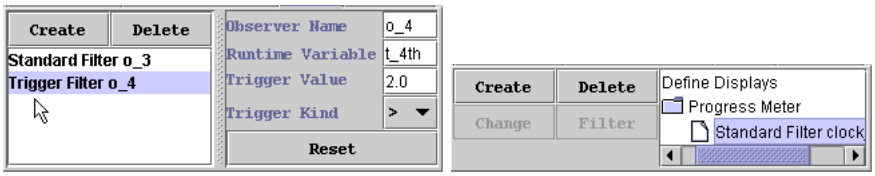
Weitere Kontrollstrukturen, wie etwa Schleifen mit beliebigen Abbruchkriterien, lassen sich prinzipiell in die Baumstruktur der Parametervariation einbringen. Hier bietet sich ein weiteres Rahmenwerk an, das ein Anstoßen von Simulationsläufen und den Zugriff auf die Zugangspunkte zur Belegung von Modellparametern sowie zur Auswertung von Laufzeitwerten ermöglicht. Auch dieses Konzept wurde jedoch noch nicht umgesetzt.

7.4.3 Beobachter

Während eines Simulationslaufs, also in der Nutzungsphase `RUNNING`, stellt ein Modellexemplar seiner Außenwelt über entsprechende Zugangspunkte Daten zur weiteren Verarbeitung bereit. Ein einziger Zugangspunkt kann dabei eine Folge von Laufzeitwerten liefern, die meist mit einem Zeitstempel als Kontextwert versehen sind. Ein bestimmtes Szenario benötigt aber nicht unbedingt alle gelieferten Daten. Es reicht häufig aus, nur einen Teil der Daten aufzuzeichnen und später auszuwerten. Damit die Menge der aufzuzeichnenden Daten auf die notwendigen Anteile beschränkt werden kann, verwendet die ExpU Datenfilter, die den *Beobachtern* aus (Langer 1989) entsprechen.

Ein Filter basiert auf einer Experimentvariablen und gibt die Bedingungen an, unter denen ein Laufzeitwert aufgezeichnet wird. Nur über die angegebenen Filter eines Experiments erfolgen auch tatsächlich Aufzeichnungen und werden im Datenhaltungssystem gespeichert. Die ExpU stellt bereits einfache Filter für numerische Datentypen zur Verfügung (Abbildung 7.27(a)). Für Filter mit komplexen Randbedingungen existiert ein Rahmenwerk, über das sich weitere, benutzerdefinierte Filter einbinden lassen (vgl. Schöllhammer 2001, Abschn. 5.6).

Auch für die möglichen Anzeigen während eines Simulationslaufs dienen die



(a) Datenfilter (b) Anzeigedefinition

Abbildung 7.27: Beobachter

Filter als Grundlage. Da mit einem Filter bereits Aufzeichnungen festliegen, greift eine Anzeige direkt auf diese Aufzeichnungen zurück. Der Ausdruck „Beobachter“ entstand in genau diesem Zusammenhang. Die ExpU bietet einige einfache Anzeigeinstrumente wie Wertetabellen, Tachonadeln oder Rollbalken an. Abbildung 7.27(b) definiert beispielsweise eine Fortschrittsanzeige für die Simulationsuhr als Rollbalken und verwendet dazu einen für alle Werte durchlässigen Standardfilter. Da viele gewünschte Visualisierungen modellspezifisch oder zumindest domänenspezifisch sind, können auch hier weitere Instrumente über spezielle Schnittstellen eingebracht werden.

7.4.4 Laufsteuerung

Im Parameterteil eines Simulationsexperiments definiert ein Modellnutzer zunächst die Folge durchzuführender Simulationsläufe. Aus der Steuerungsansicht der ExpU (Abbildung 7.28) stößt er dann die Ausführung an. Dabei lässt sich der Fortschritt eines Simulationslaufs auch interaktiv steuern, wenn die Modellfabrik diese Steuerung über einen entsprechenden Zugangspunkt zur Simulationsuhr ermöglicht. Dann dient das Feld `Limit Time` für den markierten Lauf als Grenze, bis zu der die Simulationszeit fortschreiten darf.

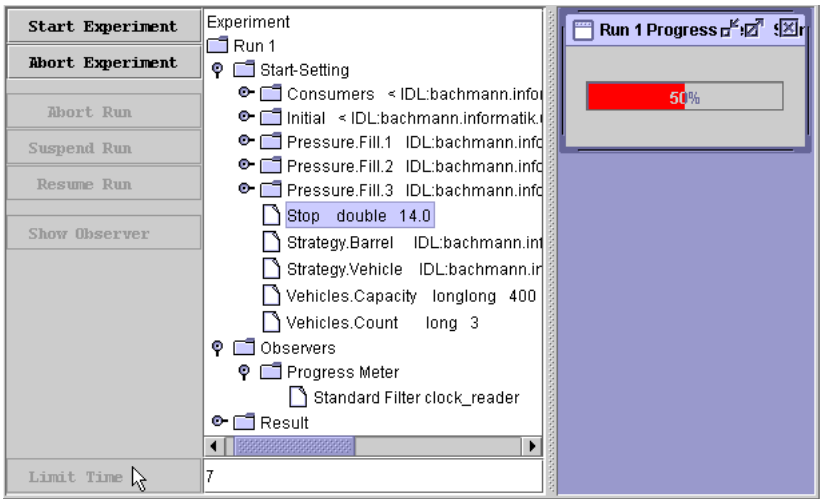


Abbildung 7.28: Laufsteuerung

Die Steuerungsansicht enthält weiterhin für jeden Simulationslauf die ver-

wendeten Parametrisierungen, die vorliegenden Aufzeichnungen der Filter und ihre Anzeigeinstrumente. Nachdem ein Lauf beendet ist, stehen auch die Ergebniswerte zur Verfügung, die durch das Modellexemplar direkt berechnet wurden. In der aktuellen ExpU-Realisierung können bisher jedoch während eines Simulationslaufs noch keine Werte interaktiv an das Modellexemplar geliefert werden. Eine solche Möglichkeit teilt einen einzelnen Simulationslauf nach (Langer 1989) in mehrere Laufabschnitte ein, die dann auch in den Verwaltungs- und Dokumentationsstrukturen der ExpU nachzuvollziehen wären.

7.4.5 Statistik

Nach Durchführung der Simulationsläufe, die durch eine Experimentspezifikation angegeben wurden, benötigt ein Benutzer die Möglichkeit, die aufgezeichneten Daten der Simulationsläufe weiter zu analysieren, beispielsweise um Kenngrößen wie Maxima, Minima oder Mittelwerte von Zahlreihen zu ermitteln. Diese einfachen Kenngrößen sind bereits in die ExpU integriert (vgl. Abbildung 7.29). Meist reichen die Standardverfahren jedoch nicht aus für eine detaillierte Bewertung der Simulationsergebnisse, so dass die Experimentierumgebung Schnittstellen für komplexere Auswertungen zur Verfügung stellt. Diese Schnittstellen erlauben die Auswertung sämtlicher aufgezeichneter Daten, auch über die Grenzen eines einzelnen Simulationslaufs hinweg.

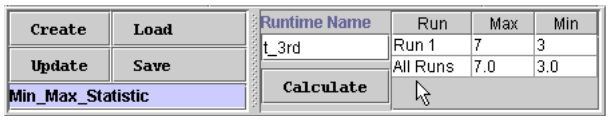


Abbildung 7.29: Statistik

Auch noch so ausgefeilte Statistikwerkzeuge bieten keine Garantie für verlässliche Simulationsergebnisse. Sie befreien die Modellbenutzer nicht von einer Gültigkeitsbewertung der vorliegenden Eingabedaten und Modellrealisierungen und besitzen selbst Grenzen, innerhalb derer sich überhaupt gültige Werte ermitteln lassen.

7.5 Zusammenfassung der Werkzeuge

Der CoSim-Adapter-Generator bettet Modellkomponenten, die in der Programmiersprache Java vorliegen, in die CoSim-Architektur ein. Mit Hilfe dieses Werkzeugs lassen sich CoSim-Komponenten ohne direkten Kontakt mit den umfangreichen IDL-Schnittstellen erzeugen. Der zusätzliche Programmieraufwand für den Komponentenentwickler ist gering. Ein Entwickler muss die Schnittstellen für Übergänge zwischen den Nutzungsphasen ausfüllen und je Zugangspunkt meist nur eine weitere Methode bereitstellen. Dabei sind Regeln für die Methodennamen zu beachten, um die gewünschte Kommunikationsart zu erhalten. Weitere Regeln existieren, um Haupt- und Kontextwerte zusammenzuhalten. Der Generator verwendet ein spezielles Ablagesystem, um vorhandene Modellkomponenten im Dateisystem aufzufinden und zu aktivieren.

Der CoSim-Component-Connector koppelt vorhandene Modellkomponenten und verwendet einen Grapheneditor mit spezieller Symbolsprache, um den Da-

tenfluss zwischen beteiligten Zugangspunkten durch die Modellentwickler spezifizieren zu lassen. Durch konfigurierbare Datenpuffer werden die Unterschiede zwischen Kommunikationsarten überbrückt.

Einfache Transformationen der übertragenen Daten kann ein Modellentwickler bei der Kopplung als Java-Ausdruck angeben. Für komplexere Transformationen kommen Java-Klassen zum Einsatz, denen der Zugriff auf die Ausführungsumgebung gewährt wird. Auch die Verwendung von Kontextwerten unterstützen beide Transformationsmechanismen.

Die Experimentierumgebung erlaubt einem Modellnutzer, vorhandene Modellfabriken ohne Programmierung zu nutzen. Dabei werden alle Phasen des Experimentierens unterstützt: Parametrisierung, Laufdurchführung und Auswertung. Mehrfache Simulationsläufe lassen sich durch verschiedene Iterationsformen angeben. Während eines Simulationslaufs können eine Datenfilterung und eine einfache Visualisierung erfolgen. Nach dem Abschluss eines Experiments lässt sich die Auswertung von Statistikfunktionen anstoßen. Schnittstellen zu den einzelnen Aufgabenbereichen erlauben es den Modellnutzern, die Funktionalität der Umgebung in verschiedene Richtungen zu erweitern.

Alle Experimenteinstellungen sind strikt von den verwendeten Modellen getrennt. Dazu führt die Experimentierumgebung eigene Experimentvariablen ein und bildet sie auf die modellspezifischen Zugangspunkte ab.

Die Modellfabriken und ihre Beschreibungen durch Zugangspunkte liegen in verschiedenen Verzeichnissen vor. Eine grafische Benutzungsschnittstelle erlaubt den komfortablen Zugang auf die Verzeichnisinhalte durch die Modellentwickler und Modellbenutzer. Auswahl-, Kopier- und Verschiebeaktionen sind mit wenig Aufwand durch Drag & Drop- bzw. Cut & Paste-Techniken durchführbar, sogar über die Grenzen eines Verzeichniswerkzeugs hinaus. Auch für die Angabe von Objektreferenzen bei der Modellkopplung und bei der Experimentdurchführung können diese Techniken eingesetzt werden.

Die vorgestellten vier Werkzeuge zur Modellerstellung und -kopplung, zur Experimentdurchführung und zum Verzeichniszugang stehen bisher als prototypische Realisierungen bereit. Sie decken bereits einen großen Teil der Aufgaben im Bereich der komponentenorientierten Simulation ab. Bisher fehlt es jedoch noch an geeigneten Aufbereitungen vieler Anwendungsgebiete der Simulation in Form von Referenzmodellen und Modellklassifikationen. Wenn hier entsprechende Vorleistungen erbracht wurden, kann sich ein Markt für Simulationskomponenten aufbauen. Dann wird auch der Bedarf entstehen, kommerzielle Simulatoren in eine größere Umgebung wie CoSim einzubetten und die zugehörigen Adapter zu entwerfen.

8 Realisierung der Umgebung

Sämtliche Werkzeuge der CoSim-Umgebung wurden in Java realisiert, um sie auf einer großen Anzahl von Plattformen einsetzen zu können. Keines der Werkzeuge macht spezielle Annahmen über die verwendete Plattform, lediglich eine vollständige Java 1.2 Umgebung (inklusive Compiler) ist notwendig.

Der Benutzerzugang zu den einzelnen Werkzeugen wurde bereits in Kapitel 7 dargestellt. In diesem Kapitel wird es nun um Einzelheiten der vorhandenen Implementation gehen. Dabei kann natürlich nicht der gesamte Quellcode im Umfang von etwa 10 Megabyte diskutiert werden. Etwa die Hälfte des Codes wurde zudem automatisch durch Übersetzung von IDL-Deklarationen erzeugt oder entstand durch Überarbeitung externer Quellen. Vielmehr sollen die einzelnen Abschnitte diejenigen Problemlösungen behandeln, die möglicherweise schwer zu durchdringen sind.

Da einige Werkzeuge eine grafische Benutzungsschnittstelle bieten, floss dort viel Programmierarbeit ein, die jedoch hauptsächlich aus der Anordnung von grafischen Elementen und deren Ereignisbehandlung besteht. Diese Teile sind im Wesentlichen Fleißarbeit. Schwieriger waren jedoch Probleme, die durch Reflexion oder automatische Codegenerierung gelöst wurden. Mit der Beschreibung dieser Problemlösungen wird deutlich, dass die CoSim-Umgebung einen Prototypen darstellt, der in den wichtigen funktionalen Punkten bis in die Tiefe realisiert wurde und damit die Umsetzbarkeit des CoSim-Ansatzes nachweist.

Auf die Realisierung der Experimentierumgebung (ExpU) muss hier nicht weiter eingegangen werden. Sie wurde im Rahmen einer Diplomarbeit (Schöllhammer 2001) entwickelt. Dort finden sich auch Details zur Implementation. Das Factory Repository taucht nicht auf, da die Grundidee zum Verwalten von Zuordnungen zwischen IOR-Texten der Modellfabriken und Modellbeschreibungen bereits auf allgemeinerer Basis in Java-Hashtables zu finden ist.

8.1 Das Lookup-Repository

Das Lookup-Repository dient zur persistenten Speicherung von Modellbeschreibungen. Dazu gehören die Modellbeschreibungen und die Zugangspunkte, aber auch die optionalen Initialwerte und Gültigkeitsbereiche der Zugangspunkte. In der Dienstdefinition für das Lookup-Repository im Anhang A (Datei `cosim_lookup.idl`). finden sich die Methoden zum Erzeugen (`export`), Nachschlagen (`lookup`) und Löschen (`resign`) von Einträgen. Auch ein Inhaltsverzeichnis (`report`) ist vorgesehen.

Die Datenstrukturen, die für eine Modellbeschreibung benötigt werden, sind einerseits sehr unterschiedlich. Andererseits werden die notwendigen Implementierungsschritte zur Speicherung der Datenstrukturen ähnlich sein. Deshalb erfolgte die Realisierung des Lookup-Repositories datentypunabhängig, d.h. es lassen sich beliebige Datenwerte speichern, deren Typen per IDL definierbar

sind. Sollen Objektreferenzen gespeichert werden, so lässt sich nur die Repräsentation als IOR-Zeichenkette ablegen. Üblicherweise setzt man hierbei nur persistente Objektreferenzen ein, also nur solche, die nach dem Beenden und Neustarten ihres Serverprozesses immer noch unter derselben IOR zugänglich sind.

Für die Beschreibung von Modellen werden strukturierte IDL-Datentypen (`struct`, `sequence`, `enum`) eingesetzt. Ausprägungen dieser Typen lassen sich bei Anfragen nur als Kopie übertragen und besitzen keine eigene Identität. Die Identität wird ihnen erst durch eine geeignete Objekthülle (wie in Abschnitt 6.4.2, Zeichnung 6.5) verliehen. Da die Objektidentität in ihrer IOR-Form als lange Zahlenreihe für den Menschen nur schlecht zu lesen oder einzugeben ist, werden die Objekthüllen innerhalb eines Namensdienstes verzeichnet (vgl. Abschnitt 6.7.1), wobei der Objektname durch den Nutzer des Lookup-Repositories vergeben wird. Als Nutzer treten jedoch auch die Werkzeuge *Cage* und *CoCo* auf, so dass nur ein Teil des gesamten Namens direkt durch den Modellentwickler bestimmt wird. Der vollständig aufgelöste Objektname aus dem Namensdienst liefert zunächst die Objektreferenz für die gespeicherte Objekthülle. Aus der Objekthülle müssen schließlich noch die eigentlichen Nutzdaten extrahiert werden. Je nach Art der Objekthülle erhält man dabei unterschiedliche Datentypen. Die Hülle `Lookup_Class` beispielsweise repräsentiert eine Modellbeschreibung, die die Klassifikation der Zugangspunkte enthält, und liefert eine Struktur `Model_Class`. Zugangspunkte wiederum sind in `Lookup_Connection`-Objekte verhüllt und als Struktur `Model_Connection` beschrieben.

Um das Lookup-Repository zu realisieren, sind drei zentrale Aufgaben zu lösen, die in den folgenden Abschnitten aufgegriffen werden.

- Wie lassen sich beliebige Typen von Nutzdaten einbringen und entsprechenden Arten von Objekthüllen zuordnen?
- Wie sieht ein geeignetes Speicherformat für beliebige Nutzdaten aus?
- Welche Ablagestruktur erlaubt das Auffinden der gespeicherten Daten durch `corbaname`-URLs?

8.1.1 Zuordnung von Nutzdaten und Objekthüllen

Da das Lookup-Repository mit verschiedenen Typen von Nutzdaten umgeht, wird ein Verfahren gesucht, das für einen bestimmten Nutzdatentyp einen eigenen Objekthüllentyp beschreibt. Dazu wurde ein Namensschema auf der Basis von IDL-Deklarationen verwendet.

Alle Objekthüllen sollen als Referenz nutzbar sein. Deshalb muss eine IDL-Schnittstellendeklaration verwendet werden. Es bietet sich an, eine gemeinsame Schnittstelle für Objekthüllen zu entwerfen, von der alle spezifischen Objekthüllen erben; sie heißt `Lookup_Content`. Diese Schnittstelle erlaubt dem Benutzer einer entsprechenden Objektreferenz, über die Methode `get_type` die eigentliche, von `Lookup_Content` abgeleitete Klasse der Objektreferenz zu erfragen, um danach die Nutzdaten zu extrahieren. Außerdem ist die Schnittstelle `Lookup_Content` abgeleitet von `Lookup_Object` und ermöglicht über die Methode `get_name` die Umkehrabbildung zum Nachschlagen im Namensdienst. Ein solches Objekt kann selbst darüber Auskunft geben, an welcher Stelle es im Namensdienst abgelegt wurde.

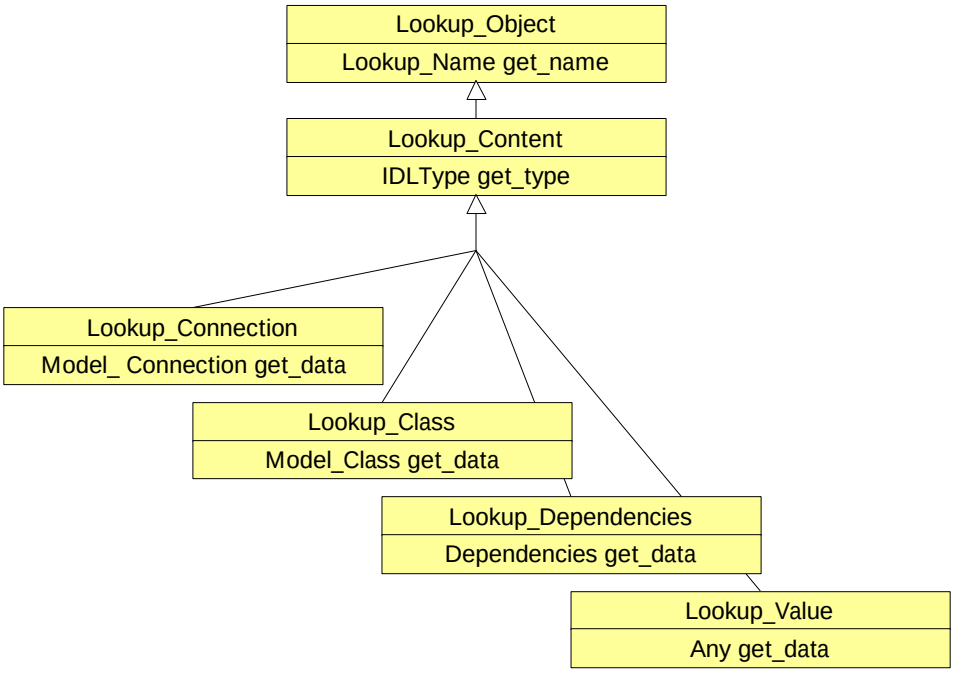


Abbildung 8.1: Namensschema für Objekthüllen

Eine Schnittstellendeklaration wird als Objekthülle akzeptiert, wenn sie von `Lookup_Content` abgeleitet ist und genau eine weitere parameterlose Methode `get_data` mit Rückgabetyp enthält. Diese Methode liefert dann die Nutzdaten. Die CoSim-Umgebung verwendet vier dieser Deklarationen, die der Beschreibung von Modellen und ihren Zugangspunkten dienen (Tabelle 8.1).

Objekthülle	**Inhaltstyp**	**Beschreibung**
Lookup_Connection	Model_Connection	Definition eines Zugangspunktes
Lookup_Class	Model_Class	Partition aller Zugangspunkte einer Modellbeschreibung
Lookup_Dependencies	Dependencies	Abhängigkeiten zwischen Zugangspunkten innerhalb eines Modellexemplars
Lookup_Value	Any	Ein beliebiger Datenwert (beispielsweise als Anfangswert eines Zugangspunktes)

Tabelle 8.1: CoSim-Typen für Objekthüllen

Die Schnittstellendeklaration für eine Objekthülle unterliegt nur einer einzigen Randbedingung: Sie muss zur Laufzeit vom Lookup-Repository inspiziert werden können. Da das Lookup-Repository selbst CORBA Interface-Repositories verwendet, um unbekannte Typdefinitionen nachzuschlagen, wird diese Bedingung erfüllt, wenn die Schnittstellendeklaration in einem Interface-Repository enthalten ist. Das Interface-Repository kann die Schnittstellendeklaration aber nur übernehmen, wenn dort auch der Rückgabetyp der Methode

`get_data` bekannt gemacht wird. Durch administrative Maßnahmen lassen sich auch noch nach dem Start beider Dienste (Lookup-Repository und Interface-Repository) weitere Schnittstellendeklarationen in das Interface-Repository einfügen.

Damit das Lookup-Repository dem vorgestellten Namensschema auch noch folgen kann, wenn ganz neue Deklarationen verwendet werden, die zum Zeitpunkt der Programmierung dieses Dienstes noch gar nicht existierten, verwendet die Java-Implementation für die Objekthüllen eine sog. `DynamicImplementation`. Dabei werden Anfragen nicht durch vorkompilierte Aufbereitungscodes (Skeletons) verarbeitet, sondern die Implementationsklasse bekommt die Anfragen in einem Rohzustand übergeben und muss selbst für die Interpretation der Objektidentität, des Methodennamens und der übergebenen Parametercodierung sorgen. Dieses Verfahren wird als *Dynamic Skeleton Interface* bezeichnet. Dafür kann ein einziges Objekt (DefaultServant) dieser Implementationsklasse jedoch viele CORBA-Objektidentitäten verwalten. Auch der *Datentyp* lässt sich für jedes verwaltete CORBA-Objekt einzeln einstellen. Details zur Verwaltung und Aktivierung von dynamischen Realisierungen finden sich in der Spezifikation für Portable Object Adapter (POA: OMG 1999, Kap. 11).

Jede vom Lookup-Repository herausgegebene Objekthülle enthält nun in ihrer zugehörigen Objektreferenz eine Codierung für den Ablageort ihrer gespeicherten Daten. Dadurch kann ein DefaultServant die Anfragen `get_name`, `get_type` und `get_data` beantworten, nachdem die gespeicherten Daten aus der Ablage eingelesen wurden. Die herausgegebenen Objektreferenzen sind selbst persistent, d.h. ein Client kann eine Objektreferenz lokal speichern. Sie verliert nicht ihre Gültigkeit, wenn das Lookup-Repository neu gestartet wird.

8.1.2 Speicherformat für IDL-Datenstrukturen

Es existiert bereits ein Übertragungsformat für beliebige IDL-Datenstrukturen (Common Data Representation, CDR: OMG 1999, Abschn. 15.3), das alle Object Request Broker zur Beantwortung von Anfragen erkennen müssen. Die Codierung in dieses Format und wieder zurück (Marshalling, Unmarshalling) erledigen üblicherweise Hilfsklassen, die durch einen IDL-Compiler erzeugt wurden. Für Typen, die erst nach der Programmierung des Lookup-Repositories entstanden sind, liegen diese Hilfsklassen nicht vor. Ohne diese Hilfsklassen ist das Marshalling und Unmarshalling nur auf Umwegen möglich. Zwar gibt es ein Verfahren, um beliebige Datentypen als Java-Objekte einer festen Klasse zu repräsentieren (org.omg.CORBA.Any: vgl. OMG 2001b, Abschn. 2.14), aber diese Repräsentation ist abhängig vom verwendeten Object Request Broker. Die Schnittstellen zum Zugriff auf Daten beliebigen Typs sind also festgelegt, nur die Implementation unterscheidet sich zwischen den verschiedenen ORBs. Weiterhin gestatten diese Schnittstellen keine portable Serialisierung der enthaltenen Daten, d.h. das Lesen und Schreiben dieser Daten mit Hilfe von Java-Datenströmen ist nicht möglich.

Abhilfe schaffen hier Schnittstellen, durch die sich beliebige Datenstrukturen *konstruieren* lassen (DynamicAny: vgl. OMG 1999, Kap. 9). Um DynamicAny-Strukturen in Zusammenhang mit vorher unbekannten Datentypen zu benutzen, müssen die neuen Typen in einem Interface-Repository auffindbar sein. Aus dem Interface-Repository wird eine Objektreferenz geliefert, deren Schnittstellen eine detaillierte Introspektion des Typaufbaus erlauben, und zwar unabhängig vom

verwendeten ORB. Jeder ORB kann diese Objektreferenz in eine eigene Typrepräsentation umwandeln und mit dieser internen Typrepräsentation Behälter für Objekte des angegebenen Typs konstruieren. Die Behälter lassen sich über die DynamicAny-Schnittstellen auf portable Art und Weise so manipulieren, dass voll kompatible CORBA-Datenstrukturen mit den gewünschten Inhalten entstehen, d.h. die gespeicherten Nutzdaten von Objekthüllen können in diese Behälter übertragen werden. Die DynamicAny-Schnittstellen ermöglichen außerdem eine Zerlegung komplexer Datenstrukturen in eine Folge von Werten der Basisdatentypen wie Ganzzahlen, Booleschen Werten, Zeichenketten etc. Diese Basisdatentypen sind serialisierbar und lassen sich problemlos mit den üblichen Java-Konstrukten in Datenströme schreiben. Damit ist also eine portable Speicherung der Nutzdaten von Objekthüllen möglich.

Sämtliche Nutzdaten sind als serialisierte Objekte abgelegt. Abschnitt 8.1.3 zeigt, wie dies auf der Basis von Verzeichnissen und Dateien organisiert wird. Als Hilfsstrukturen zur Serialisierung wurden zwei Klassen `Type` und `Serial` eingeführt.

```
class Type implements Serializable {
  public String repository = null;
  public String id         = null;
  public int    primitive  = -1;}

class Serial implements Serializable {
  public Type         type = null;
  public Serializable data = null;}
```

Jedes Nutzdatum ist als Objekt der Klasse `Serial` codiert. Dazu gehört eine Typbeschreibung, aus der sich der weitere Aufbau der Datencodierung ableiten lässt. Jede Typbeschreibung ist in einem Interface-Repository enthalten (Feld `Type.repository`). Dabei muss man unterscheiden, ob es sich um einen vordefinierten (primitiven) Datentyp wie `long`, `boolean`, `::CORBA::Object`, `any` etc. handelt, oder ob der Typ benutzerdefiniert und mit einem Namen versehen ist. Die Zugriffsmethoden des Interface-Repository unterscheiden nämlich zwischen primitiven und benutzerdefinierten Typen. Im primitiven Fall besitzt der Datentyp eine ganzzahlige Codierung und das Feld `Type.primitive` enthält einen positiven Wert. Über diesen Wert lässt sich aus dem Interface-Repository die Typbeschreibung immer direkt entnehmen. Andernfalls steht der Typname im Feld `Type.id` und kann im Interface-Repository gesucht werden. Diese Suche kann fehlschlagen, wenn der Typ nicht durch administrative Maßnahmen im angegebenen Interface-Repository eingetragen wurde. Daher rührt die Randbedingung, dass verwendbare Objekthüllen und Nutzdatentypen in einem Interface-Repository enthalten sein müssen.

Die tatsächliche Datencodierung im Feld `Serial.data` hängt unmittelbar vom Aufbau des Nutzdatentyps ab. Die folgenden Regeln sind beim Marshalling bzw. Unmarshalling ggf. rekursiv anzuwenden.

- Den primitiven Datentypen stehen entsprechende Java-Typen gegenüber. Sie können dem IDL-Java-Mapping entnommen werden (OMG 2001b, Abschn. 2.2). Für die Basisdatentypen im Java-Mapping werden dann die entsprechenden Java-Objektklassen verwendet. Beispielsweise ist dem IDL-Typ `long` der Java-Typ `int` zugeordnet. In das Feld `Serial.data` wird dann ein Wert der Klasse `java.lang.Integer` eingetragen.

- IDL-Ganzzahlen großer Länge (`fixed`) liegen als Zeichenketten vor.
- Sämtliche IDL-Typen, die als `interface` deklariert sind, führen zur Speicherung der Objektreferenz als IOR-Zeichenkette.
- Die Elemente von Aufzählungen `enum` lassen sich als Ganzzahlen darstellen.
- Strukturierte Typen (`struct`, `except` und `ValueBase`) bestehen aus einer Folge benannter Elemente. Entsprechende Daten werden einfach als Array ihrer Elemente gespeichert, genauso wie die Elemente der IDL-Sequenztypen `array` und `sequence`.
- Variante IDL-Strukturen (`union`) sind als Array der festen Länge zwei repräsentiert. Der erste Eintrag enthält den Wert des Feldes `discriminator` aus der zugehörigen Java-Datenstruktur. Der zweite Eintrag enthält die variante Datenstruktur.
- Lokale IDL-Typrepräsentationen (`TypeCode`) werden im Interface-Repository aufgesucht und als IOR-Zeichenkette des entsprechenden `IDLType`-Eintrags abgelegt.
- Der Typ `any` ist lediglich eine weitere Hülle für beliebige Datenstrukturen. Hier wird das verpackte Datum als alleiniges Objekt rekursiv verfolgt.
- Die IDL-Typen `null` und `void` lassen das Datenfeld unbelegt.

Eingangs wurde bereits deutlich, dass es nur für persistente Objektreferenzen sinnvoll ist, sie im Lookup-Repository abzulegen. Einige Implementationen (z.B. Visibroker 4.5.1, OpenORB 1.2) liefern für die Typbeschreibungen (`IDLType`) im Interface-Repository nur *transiente* Objektreferenzen. Bei jedem Neustart dieser Interface-Repositories erzeugt beispielsweise die Suche nach benutzerdefinierten Typen trotz unveränderter Inhalte im Interface-Repository immer unterschiedliche Objektreferenzen. Diese Situation verhindert die praktische Nutzung von Modellbeschreibungen, da die IDL-Struktur `Model_Connection` ja gerade den Typ einer Modellgröße enthält. Damit auch solche Strukturen einsetzbar werden, benutzt das Lookup-Repository einen Trick, der aus einer transienten Typbeschreibung eine persistente macht.

Analog zur Typcodierung für die Speicherung von Nutzdaten beliebiger Datentypen (Java-Klasse `Type`) wird nun eine *IDL*-Schnittstelle definiert, die die identifizierenden Daten einer Typbeschreibung liefert.

```
interface Persistent_Type : ::CORBA::IDLType {
  ::CORBA::Repository get_repository ();
  string              get_repository_id ();
  long                get_repository_primitive ();
};
```

Das Lookup-Repository erzeugt dann für jede zu speichernde Typbeschreibung eine neue Objektreferenz und benutzt dazu einen persistenten Aktivierungsmechanismus der POA-Schnittstellen. Die neue Objektreferenz enthält in ihrer Objektidentität codiert die identifizierenden Merkmale der ursprünglichen Typbeschreibung, nämlich das übergeordnete Interface-Repository, den Typnamen oder die Zahlencodierung eines primitiven Typs. Die Objektreferenz des Interface-Repositories, also der primäre Dienstzugangspunkt, ist in allen

bekannten Implementierungen selbst persistent. Deshalb ist die ursprüngliche Typbeschreibung mit diesen drei Merkmalen eindeutig und lässt sich auch nach einem Neustart des Interface-Repositories rekonstruieren. Über die DynamicAny-Schnittstellen können die betroffenen Objektreferenzen vor dem Speichern und vor der Herausgabe problemlos umgewandelt werden. Es entstehen hier keine Konflikte mit dem IDL-Typsystem, da die neuen Objektreferenzen (`Persistent_Type`) von der usprünglichen Deklaration (`IDLType`) erben und somit zuweisungskompatibel sind.

8.1.3 Ablagestruktur der Nutzdaten

Im Lookup-Repository sollen sich Objekthüllen anhand des vergebenen Namens als `corbaname`-URL leicht wiederfinden lassen. Es kann jedoch mehrere URL-Texte geben, die die gleiche Objektreferenz liefern, etwa durch Verwendung einer IP-Adresse anstatt eines DNS-Namens für die Rechneridentifikation, oder weil der Namensdienst mehrere Wege zwischen zwei Kontexten zulässt. Deshalb kann kein URL-Text als Identifikation abgelegter Nutzdaten dienen, sondern es muss ein mehrstufiges Schema auf der Basis von Objekten verwendet werden. Die erste Stufe liefert den Kontext im Namensdienst. Ein Kontext ist selbst ein CORBA-Objekt und wird deshalb eindeutig gekennzeichnet durch die IOR-Zahlencodierung seiner Objektreferenz. Innerhalb dieses Kontexts identifizieren wiederum alle Namen die zugehörigen Nutzdaten eindeutig. Ein Name besteht aus den Teilen `kind` und `id`. Das Lookup-Repository verwendet hier eine zweite Stufe je `kind`-Angabe, und als dritte Stufe der Identifikation ist die `id`-Angabe vorgesehen.

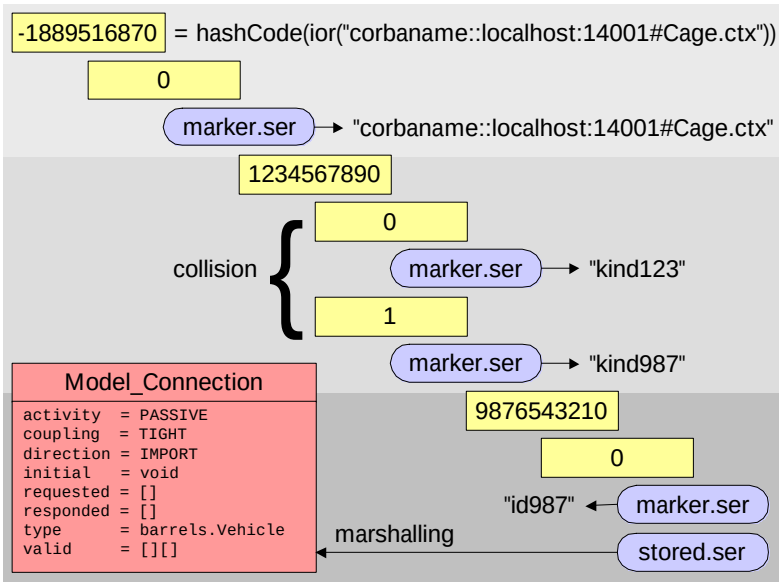


Abbildung 8.2: Speicherinhalt eines Lookup-Repositories

Die Verzeichnisse und Dateien eines Dateisystems eignen sich in diesem Zusammenhang als physikalische Ablagestruktur, da auch sie eine hierarchische Navigation ermöglichen. Damit jedoch keine Konflikte mit den Dateinamen im

verwendeten Dateisystem entstehen (etwa hinsichtlich der verwendbaren Zeichen oder der maximalen Länge), können weder die IOR-Zahlencodierung noch die vorhandenen `kind-` und `id`-Angaben direkt als Dateinamen eingesetzt werden, sondern müssen zunächst umcodiert werden. Um die Länge der Dateinamen zu begrenzen, greift das Lookup-Repository auf Verfahren der Streuspeicherung (HashCodes) zurück. Die Programmiersprache Java liefert bereits durch die Methode `hashCode` für jedes Objekt eine 32-bit Ganzzahl. Die textuelle Repräsentation dieser Zahl kann problemlos als Dateiname eingesetzt werden. Bei der Vergabe von HashCodes können jedoch Kollisionen auftreten, d.h. verschiedene Objekte liefern dieselbe Ganzzahl. Nach dem Zugriff über den HashCode ist also eine anschließende Kollisionsauflösung notwendig. Das Lookup-Repository führt deshalb jeweils eine weitere Zwischenstufe ein, in der je kollidierendem Wert ein Unterverzeichnis eingerichtet wird, das den ursprünglichen Identifikator (hier *Marker* genannt) enthält. Der Marker ist dann in einer eigenen Datei abgelegt. Die Unterverzeichnisse werden so nummeriert, dass jeweils die kleinste freie Nummer verwendet wird und keine Ganzzahlüberläufe auftreten. Zur Kollisionsauflösung müssen also alle Kollisionsverzeichnisse nach dem ursprünglichen Marker durchsucht werden. Das Streuspeicherverfahren ist jedoch so angelegt, dass Kollisionen selten auftreten und die Kollisionen gleichmäßig auf die HashCodes verteilt sind. Dadurch müssen nur wenige Verzeichnisse in die Kollisionsauflösung einbezogen werden. Abbildung 8.2 zeigt einen typischen Ablageinhalt. Die „`.ser`“-Dateien enthalten die Nutzdaten bzw. ursprünglichen Markerwerte.

8.2 Der Model-Finder

Unter dem Präfix `de.unihh.informatik.bachmann.model_finder` wurden verschiedene Java Packages zusammengefasst, die den in Abschnitt 7.2 beschriebenen Zugang zu Verzeichnisdiensten durch grafische Benutzungsschnittstellen ermöglichen. Die Realisierung dieser Benutzungsschnittstellen erfolgte meist nach dem gleichen Schema. Nachdem die GUI-Elemente in Dialogen und Fenstern angeordnet waren, mussten die Aktionen der Schaltflächen programmiert werden, um die weiteren Elemente mit den richtigen Inhalten zu füllen. Im Folgenden werden zwei Themen kurz dargestellt, die häufiger auftauchten und Gemeinsamkeiten beim Zugang zu den einzelnen Verzeichnisdiensten verdeutlichen. Es geht dabei um die Fallunterscheidungen für CORBA-Typen und die Datenübertragung zwischen den einzelnen Fenstern per Drag & Drop und über eine Zwischenablage.

8.2.1 Fallunterscheidungen

Der größte Teil des erstellten Quellcodes für den Verzeichniszugang gruppiert sich um Fallunterscheidungen, die für die Verwendung beliebiger Datentypen notwendig werden. Der Lookup Generator aus Abbildung 7.5 beispielsweise stellt im Hauptanzeigefeld beliebige CORBA Datenstrukturen als Baum dar. Für jeden primitiven CORBA Datentyp und für jedes Muster, nach dem CORBA Datentypen aufgebaut werden können (`struct`, `sequence`, `interface` etc.), gibt es eine spezielle Klasse, die die vorliegende Typkonstruktion behandelt und als Knoten im Baum darstellt. Der Wert des sog. Typecode Kind (`TCKind`) des behandelten CORBA Datentyps sorgt hier für die Unterscheidung der einzelnen

Klassen.

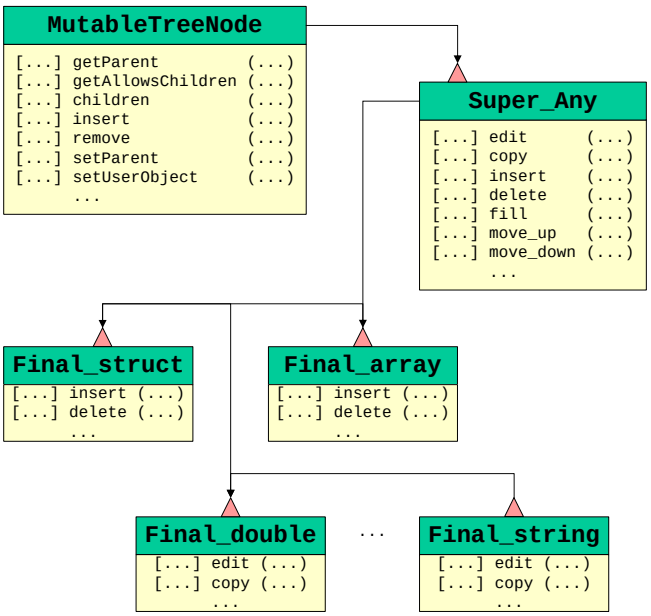


Abbildung 8.3: Klassenhierarchie für Typ-Knoten

Die gemeinsame Oberklasse `Super_Any` implementiert die allgemeinen Funktionen zur Darstellung von Knoten in Graphen, die durch das Rahmenwerk Java/Swing als Schnittstelle `MutableTreeNode` vorgesehen sind. Weiterhin enthält die Oberklasse Standardimplementationen für die Operationen, die die Schaltflächen des Lookup Generators benötigen. Dazu gehören Befehle wie `Edit`, `Delete` und `Insert`, die in den Unterklassen mit dem Präfix `Final_` an den tatsächlich vorliegenden Datentyp angepasst werden (Abbildung 8.3). Die darunter liegenden Datenstrukturen lassen sich wie im Lookup Repository am Besten als `DynamicAny` speichern und manipulieren (vgl. Abschnitt 8.1.2).

Ähnliche Fallunterscheidungen finden sich auch in der Realisierung des Interface Repository Browsers (Abschnitt 7.2.3). Dieses Verzeichnis kann über die Benutzungsschnittstelle jedoch nicht manipuliert werden. Deshalb waren hier keine Befehle zu realisieren, und auf viele Unterklassen für einzelne Knoten konnte verzichtet werden.

8.2.2 Datentransfer zwischen Dialogelementen

Da sämtliche CoSim-Werkzeuge, die eine GUI-Unterstützung anbieten, in der Programmiersprache Java realisiert sind, bietet sich die Nutzung des Java/AWT-Mechanismus zum Datentransfer zwischen verschiedenen Dialogelementen an. Zum Realisierungszeitpunkt der Werkzeuge Console und CoCo, die den größten Teil der GUI-Funktionalität tragen, war die Java-Version 1.3 aktuell. Die nachfolgenden Ausführungen beziehen sich daher auf diese Version. Inzwischen wurden mit der Einführung der Java-Version 1.4 einige Änderungen im Bereich Datentransfer in das Paket Swing aufgenommen, die in den CoSim-Werkzeugen keine Berücksichtigung mehr finden konnten.

Um in Java Daten zwischen Dialogelementen auszutauschen, benötigt man zunächst einmal sog. `DataFlavours`. Diese Datenstrukturen stellen die möglichen Formate eines Datentransfers dar. In der Java-Umgebung ist nur ein einfaches Format zur Übertragung von Texten definiert. Da die CoSim-Umgebung auch komplexere Datenstrukturen wie CORBA `Any` Objekte oder strukturierte Namen zum Nachschlagen in einem Namensdienst (vgl. Abschnitt 6.4.2) für den Transfer vorsieht, definiert das Package `de.unihh.informatik.bachmann.model_finder.datatransfer` mehrere eigene Formate (s. Tabelle 8.2). Die Formate `IDLType_Data` und `IOR_Data` speichern die Inhalte zwar auch als Text, sie unterscheiden sich jedoch von Freitext aus anderen Dialogelementen, da mit den CoSim-Formaten auch immer eine spezielle Bedeutung verbunden ist. Einige Dialogelemente der CoSim-Umgebung verlangen also ganz bestimmte Texte, die etwa IORs oder CORBA Typen repräsentieren.

Format	**Codierung**	**Inhalt**
`Any_Data`	`byte[]`	beliebige CORBA Datenstrukturen
`IDLType_Data`	`String`	Objektreferenz einer Typdefinition im CORBA Interface Repository
`Lookup_Name_Data`	`Lookup_Name`	strukturierter Name zum Nachschlagen im Namensdienst
`IOR_Data`	`String`	beliebige Objektreferenz

Tabelle 8.2: Formate für den Datentransfer

Der Datentransfer selbst erfolgt über Objekte, die der Java/AWT-Schnittstelle `Transferable` genügen. Diese Objekte verwalten die Zuordnung zwischen den Nutzdaten und ihren zugeordneten `DataFlavours`. Auch hier stellt die Java-Umgebung nur den einfachen Mechanismus für Texte zur Verfügung. CoSim definiert dazu noch zwei weitere Verwalter. Ein `Object_Transferable` Objekt kann genau ein beliebiges Datenobjekt entgegennehmen, das in serialisierter Form vorliegt. In einem `Multi_Transferable` Objekt lassen sich mehrere Objekte gleichzeitig und in verschiedenen Formatierungen unterbringen. Der Lookup Generator nutzt diese Eigenschaft beispielsweise aus, um neben den im Baum dargestellten Nutzdaten auch immer den aktuell vergebenen strukturierten Namen mit zu übertragen. Bei Drag & Drop Operationen kann dann das Dialogelement, das als Ziel des Datentransfers durch den Benutzer bestimmt wurde, selbst über die benötigten Daten und das zugehörige Format entscheiden.

Die `Transferable` Objekte wiederum verwaltet die Java-Umgebung selbst. Sie werden sowohl für Drag & Drop Operationen als auch für Cut & Paste Operationen über die Zwischenablage eingesetzt. Da viele Dialogelemente der CoSim-Werkzeuge auf Drag-Operationen hin Daten liefern bzw. auf Drop-Aktionen hin Daten verarbeiten, lassen sich Kopier- und Verschiebetätigkeiten mit wenigen Benutzereingriffen umsetzen.

8.3 Der CoSim-Adapter-Generator

Die Aufgabe des CoSim-Adapter-Generators (*Cage*) ist es, benutzerdefinierte Modelle entgegenzunehmen und zugehörige Adapter zu produzieren, die die Modellfunktionalität über CoSim-konforme Schnittstellen zur Verfügung stellen. Die notwendigen Deklarationen, die der Modellersteller bei der Benutzung des CoSim-Adapter-Generators vornehmen muss, wurden ausführlich in Abschnitt 7.1 vorgestellt. Zentrale Aufgaben für die Umsetzung dieses Adapter-Generators waren:

- Reflexion der benutzerdefinierten Methoden und der modellspezifischen Typen
- Vermittlung zwischen CoSim-Schnittstellen und benutzerdefinierten Methoden
- Konvertierung zwischen benutzerdefinierten Kontext-Strukturen und CoSim-Kontext-Strukturen
- Standardimplementation für automatische Uhrzeit-Aufrufe

8.3.1 Reflexion benutzerdefinierter Methoden

Der Benutzer übergibt Cage zunächst einen Klassennamen, der die Java-Klasse einer neuen Modelldefinition angibt. Diese Übergabe erfolgt durch eine Modelldefinitionsdatei (`.cmf`, Abschnitt 7.1.6). Aus der Position dieser Datei im Dateisystem bestimmt Cage einen neuen Klassenpfad und einen neuen `ClassLoader`. Mit Hilfe des `ClassLoaders` wird der Objektcode der neuen Modelldefinition als Java-Klasse geladen. Die Schnittstellen der `java.lang.reflect`-API ermöglichen nun die Überprüfung der Methodendefinitionen in der neuen Java-Klasse hinsichtlich der Regeln für Modelldefinitionen (Abschnitte 7.1.2 bis 7.1.4). Das Ergebnis der Reflexion ist eine Zwischenrepräsentation der Modelldefinition (`Checked_Class`), die den direkten Zugriff auf die überprüften Methoden erlaubt, um später die Umsetzung zwischen CoSim-Schnittstellen und benutzerdefinierten Methoden mit wenig Aufwand zu ermöglichen.

```
public class Checked_Class {

  Class reflected; // loaded model definition

  Constructor constructor; // using in & out PrintStreams
      Method   destructor; // given by "destroy()"

  Method static_factory_name;
  Method static_factory_url ;
  Method static_class_name  ;
  Method static_class_url   ;
  Method static_dependencies;

  Checked_Connection[] clocks    ;  // from method name suffix
  Checked_Connection[] contexts  ;  // from method name suffix
  Checked_Connection[] parameters;  // from method name suffix
  Checked_Connection[] runtimes  ;  // from method name suffix
  Checked_Connection[] results   ;} // from method name suffix
```

```
  // ========================================================================

public class Checked_Connection {

  String  identifier; // method name without prefix or suffix
  Class   type      ; // java type for main value
  Class   requested ; // java type for requested values
  Class   responded ; // java type for responded values
  Class   helper    ; // IDL-Helper class

  HashMap requested_names ; // Field -> name
  HashMap requested_fields; // name  -> Field
  HashMap responded_names ; // Field -> name
  HashMap responded_fields; // name  -> Field
  Field   responded_value ; // main value field

  Model_State          state; // one of RUNNING, GETTING, SETTING
  Connection_Coupling  coupling ;
  Connection_Activity  activity ;
  Connection_Direction direction;

  Method callee; //     blocking  main method
  Method testee; // non-blocking  test-method
  Method probee; // non-blocking probe-method
  Method cminee; // minimum clock value
  Method cmaxee; // maximum clock value

  Method static_name      ;
  Method static_auto      ;
  Method static_initial   ;
  Method static_valids    ;
  Method static_repository;
  Method static_type      ;}
```

Cage betrachtet der Reihe nach alle Methoden der neuen Klasse. Sobald eine Methode den Regeln entspricht, entsteht ein neuer `Checked_Connection`-Eintrag. Regeln, die mehrere Methoden umfassen (`snd/has/prb`- und `rec/can/try`-Muster), feuern nur bei der Hauptmethode (`snd` und `rec`). Die weiteren Nebenmethoden (`has/prb` und `can/try`, sowie `min/max` für Simulationszeiten) werden direkt mit der Hauptmethode geprüft, damit keine Einträge mehrfach erfolgen.

Nachdem alle `Checked_Connection`-Einträge erledigt sind, prüft Cage noch, ob die referenzierten Kontextwerte und Modellabhängigkeiten im Modell auch tatsächlich deklariert wurden. Referenzen erfolgten in der Modelldefinition ausschließlich durch Namen von Modellgrößen. Verwendete Namen müssen sich abschließend also in der Menge aller erkannten `Checked_Connection`-Einträge wiederfinden lassen.

Verläuft auch diese Prüfung erfolgreich, erzeugt Cage im zugehörigen Lookup-Repository aus der Zwischenrepräsentation die vollständige Modellbeschreibung. Damit die Zuordnungen von erzeugten Objekthüllen, Zwischenrepräsentation und Namen der Zugangspunkte später einfach wiederzufinden sind, werden diese Zuordnungen in verschiedenen `HashMaps` abgelegt.

```
public class Hashed_Class {
```

```
Lookup_Dependencies lookup_dependencies;
Lookup_Class        lookup_class       ;

HashMap hashed_checked; // Checked_Connection -->  Lookup_Connection
HashMap hashed_lookup;  // Lookup ior         --> Checked_Connection
HashMap hashed_names;}  // name               -->  Lookup_Connection
```

8.3.2 Reflexion modellspezifischer Typen

Mit Cage ist auch die Verwendung von modellspezifischen Typen möglich, die zum Zeitpunkt der Cage-Implementation noch nicht existierten. Cage muss jedoch in die Lage versetzt werden, für Zugangspunkte Typbeschreibungen im Lookup-Repository einzufügen und das Marshalling bzw. Unmarshalling für übertragene Datenwerte durchzuführen. Ein typunabhängiger Zugang zu Cage (wie für das Lookup-Repository) durch einen Modellersteller ist hier nicht sinnvoll, da die typabhängigen Zugriffsmethoden ja verwendet werden sollen, um das Modell zu entwerfen und zu übersetzen.

Für das Marshalling bzw. Unmarshalling muss der Modellersteller deshalb die neuen Typen in IDL definieren und mit Hilfe eines IDL-Compilers Hilfsklassen erzeugen, die die Java-Umsetzung der IDL-Typen darstellen und die Anbindung an den verwendeten ORB regeln. Eine umgesetzte Java-Klasse ist dann als Parameter- oder Rückgabetyp in der zu reflektierenden Methodendefinition verwendbar. Neben dieser Java-Klasse erzeugt der IDL-Compiler noch eine sog. `Helper`-Klasse, die Methoden zum Marshalling bzw. Unmarshalling enthält. Diese Klasse ist auch im Feld `Checked_Connection.helper` eingetragen, vorausgesetzt der Klassenpfad für die Modelldefinition ermöglicht das Auffinden des zugehörigen Objektcodes.

Um eine Typbeschreibung einzufügen, wertet Cage die `Helper`-Klasse aus und erfährt zunächst den Namen der zugehörigen Typbeschreibung. Die Beschreibung selbst wird einem Interface-Repository entnommen, das der Modellersteller mit Hilfe einer statischen `_repository`-Methode angeben kann, die den Regeln zur Definition von Zugangspunkten entspricht (Abschnitt 7.1.4).

Da sich in jedem Interface-Repository bereits Definitionen für Sequenzen der Basistypen (z.B. `::CORBA::BooleanSeq`) befinden sollten, können einfache Java-Arrays dieser Basistypen in den Methodensignaturen direkt eingesetzt werden. Für Sequenzen höherer Ordnung und für Sequenzen modellspezifischer Typen sind jedoch eigene IDL-Deklarationen notwendig, da die Typbeschreibungen anhand ihres Namens im Interface-Repository gesucht werden. In den Methodensignaturen können dann Java-Arrays einer Dimension verwendet werden, die der Schachtelungstiefe vorhandener IDL-Deklarationen entspricht.

Das Fässermodell aus Kapitel 3 verwendet beispielsweise im Beladeteil eine Sequenz aus `barrels::Lot`-Strukturen (Datei barrels.idl in Anhang A), um den Anfangsbestand auf dem Gelände des Abfüllbetriebs anzugeben:

```
public void set_initial_load_parameter (Lot[] load) {}
```

Sowohl für die IDL-Struktur `barrels::Lot` als auch für die entsprechende Sequenz `barrels::Lots` müssen die Java-Umsetzungen der IDL-Deklaration im modellspezifischen Klassenpfad zugänglich sein.

8.3.3 Vermittlung benutzerdefinierter Methoden

Die benutzerdefinierten Methoden eines Cage-Modells dienen

- zur statischen Beschreibung des Modells,
- zur Vorbereitung von Zustandsübergängen oder
- zum Empfang bzw. Versenden von Datenwerten der Zugangspunkte.

Die statischen Methoden zur Beschreibung eines Modells werden nur ein einziges Mal ausgewertet, kurz bevor die Einträge in das Lookup-Repository erfolgen.

Um die Anforderungen für Zustandsübergänge (CoSim-Methode `change_state` in der Datei *cosim_ model.idl*) in der Modellrealisierung mit Java nachvollziehen zu können, ruft Cage einfach die vorgesehenen parameterlosen Methoden der Java-Schnittstelle `Model_Wrapper` auf (Anhang B). Je Zustandsübergang, also je Kante in Abbildung 6.3, ist dort eine eigene Methode vorhanden. Über entsprechende `Exceptions` kann das Modell Probleme melden und einen Zustandsübergang ablehnen.

Für die Übertragung von Datenwerten sind jedoch insgesamt acht verschiedene Kommunikationsmuster vorgesehen (Abschnitt 7.1), die aus der Unterscheidung der drei Merkmale eines Zugangspunktes hervorgehen (Kommunikationsrichtung, Aktivität und Art der Kopplung; Abbildung 7.2). Zentrale Vermittlungsinstanz für alle Anfragen nach Datenwerten ist ein Objekt der Java-Klasse `cage.Connector`, das mit jedem Modellexemplar neu erzeugt wird. Der `Connector` enthält für jedes mögliche Kommunikationsmuster den Programmcode, der die vorgesehene Vermittlungsrichtung beherrscht, also die Aufrufe der CoSim-Schnittstellen an die benutzerdefinierten Java-Methoden weiterleitet oder Aufrufe der Java-Schnittstellen in Aufrufe der CoSim-Schnittstellen transformiert. Um von den verschiedenen Typen der Datenwerte zu abstrahieren, verwendet der `Connector` je Methodeneintrag aus Abbildung 7.2 interne Schnittstellen, die das Kommunikationsmuster repräsentieren und als Parameter- bzw. Rückgabetyp nur eine schwache Typisierung vornehmen (`Object` für Java-Werte und `::CORBA::Any` für CORBA-Werte). Diese Schnittstellen nehmen also eine funktionale Abstraktion ähnlich wie in funktionalen Programmiersprachen vor, indem sie den Code zur Vermittlung kapseln und auf verschiedenen modellspezifischen Datentypen operieren können.

Für das aktive Kommunikationsmuster einer Eingabegröße mit loser Kopplung werden beispielsweise drei Methoden auf der Java-Seite eingesetzt, die jeweils durch das Präfix `rec_`, `can_` bzw. `try_` gekennzeichnet sind. Dementsprechend existieren auch drei funktionale Schnittstellen, die der Connector implementiert.

```
public interface Do_Rec {public Object  execute ();}
public interface Do_Can {public boolean execute ();}
public interface Do_Try {public Object  execute ();}
```

Die `Do_Rec`-Schnittstelle wird benötigt, wenn das Modell die entsprechende `rec_`-Methode aufruft. Dann wird aus dem zugeordneten Nachrichtenkanal für die lose Kopplung ein Datenwert gelesen. Dieser Aufruf blockiert, bis tatsächlich ein Wert vorliegt. Nicht blockierende Aufrufe sind möglich durch die Testmethode mit dem `can_`-Präfix und durch die `try_`-Methode, die mit einem `null`-Wert zurückkehrt, falls keine Daten vorliegen.

Das obige Kommunikationsmuster zeigt auch, dass Methodenaufrufe vom Modell ausgehen können. Da die Methodennamen der gerufenen Methoden natürlich noch nicht bekannt waren, als Cage implementiert wurde, kann der Code für die Vermittlung von der Java- zur CoSim-Seite nicht durch Vererbung in die Modellimplementation einfließen. Stattdessen erzeugt Cage eine *Unterklasse* je Modelldefinition, in die die notwendigen Aufrufe durch Überschreiben der Java-Methoden eingefügt werden. So kann ein Modellentwickler das Modell problemlos mit einem Java-Compiler übersetzen und in die Implementation der aufgerufenen Methoden eigenen Testcode einfügen. Die benutzerdefinierte Implementation der zu vermittelnden Methoden wird in der Cage-Umgebung nicht mehr aufgerufen.

Die Unterklasse wird als Java-Quellcode erzeugt und anschließend durch einen Java-Compiler in Objektcode übersetzt. Quell- und Objektcode werden nach den Cage-Ausgaberegeln (vgl. Abschnitt 7.1.7) in verschiedenen Verzeichnissen abgelegt, die sich aus dem Namen der Modellklasse und deren Klassenpfad ergeben. Damit ist auch ein Debugging des erzeugten Codes bei laufender Cage-Umgebung möglich. Für das obige Kommunikationsmuster entstünde beispielsweise folgender Quellcode, der den Zugriff auf einen Zugangspunkt „mvar“ des modellspezifischen Java-Datentyps `Check_Mapping` während eines Simulationslaufs ermöglicht:

```
public Check_Mapping rec_mvar_runtime () {
  Do_Rec        do_rec = (Do_Rec) connector.do_rec.get("rec_mvar_runtime");
  Object        object =  do_rec. execute ();
  Check_Mapping result = (Check_Mapping) _object;
  return        result;}

public Check_Mapping try_mvar_runtime () {
  Do_Try        do_try = (Do_Try) connector.do_try.get("try_mvar_runtime");
  Object        object =  do_try. execute ();
  Check_Mapping result = (Check_Mapping) _object;
  return        result;}

public boolean can_mvar_runtime () {
  Do_Can  do_can = (Do_Can) connector.do_can.get("can_mvar_runtime");
  boolean result =  do_can. execute ();
  return  result;}
```

Die Zuordnung von Exemplaren der Zugangspunkte zu ihren Vermittlungsfunktionen erfolgt wie bei der Modellbeschreibung auch durch `HashMaps`. Je nach Kommunikationsmuster werden dabei unterschiedliche Schlüsselwerte eingesetzt. Die Methoden in den neu erzeugten Unterklassen verwenden hier ihren eigenen Methodennamen als Schlüssel. Bei passiven Zugangspunkten, die eine enge Kopplung verwenden, erfolgt der Aufruf über die CoSim-Schnittstellen `get_connection_value` und `set_connection_value`. Hier wird die Beschreibung der Zugangspunkte aus dem Lookup-Repository als Parameter übergeben, deshalb bietet sich als Schlüssel die IOR-Codierung dieser Beschreibung an.

Die Verknüpfung zwischen Exemplaren der neuen Unterklasse und ihrem zugehörigen `Connector`-Exemplar wird durch Implementation der Methoden `set_connector` und `get_connector` aus der Schnittstelle `Model_Link` ebenfalls in den erzeugten Quellcode eingefügt. Nachfolgend sind diese Deklarationen für eine neue Unterklasse der Modelldefinition `Check_Wrapper` abgebildet.

```
public class Generated_Subclass_Of_Check_Wrapper
```

```
extends Check_Wrapper implements Model_Link {

public Generated_Subclass_Of_Check_Wrapper
  (PrintStream output, PrintStream errors) {super (output, errors);}

protected Connector     connector = null;
public    Connector get_connector ()     {return this. connector;}
public    void      set_connector (Connector c) {this. connector = c;}}
```

Ein `Connector`-Exemplar verwaltet außerdem die Nachrichtenkanäle, die bei der losen Kopplung verwendet werden. Bei der Benutzung eines Nachrichtenkanals erhält der Empfänger jedoch nur diejenigen Nachrichten, die nach dem Zeitpunkt versendet wurden, an dem eine Verbindung zwischen Empfänger und Nachrichtenkanal zustande kam. Nachrichten, die vor der Verbindungsaufnahme eines Empfängers versandt wurden, speichert der Nachrichtenkanal also nicht. Deshalb muss den Interessenten von Datenwerten eines Zugangspunktes die Möglichkeit gegeben werden, eine Verbindung zum zugehörigen Nachrichtenkanal aufzubauen, *bevor* das Modellexemplar in die Nutzungsphase wechselt, in der Datenwerte über den Kanal versendet werden. Cage erzeugt nun jeden notwendigen Nachrichtenkanal, sobald das `Connector`-Exemplar existiert, und stellt umgehend eine Verbindung her für Zugangspunkte, die als Empfänger deklariert sind (Eingabegrößen). Die Verbindung für Ausgabegrößen wird hingegen erst erstellt, wenn die Nutzungsphase tatsächlich erreicht ist. Ein Nachrichtenkanal wird geschlossen und neu erzeugt, wenn die Nutzungsphase des zugehörigen Zugangspunktes wieder verlassen wird. Damit ist sichergestellt, dass keine Werte aus einem vorangegangenen Zyklus der Nutzungsphasen übertragen werden.

8.3.4 Konvertierung von Kontext-Strukturen

Die Abbildung von CoSim-Kontextwerten (Abschnitt 6.6) in Java-Datenstrukturen erfolgt wie in Abschnitt 7.1.5 beschrieben durch ein Muster, das den benutzerdefinierten Java-Klassen die Deklaration bestimmter Datenfelder vorschreibt. Deshalb ist es Cage möglich, die betreffenden Datenfelder der Java-Klasse anhand ihres Namens zu erkennen und automatisch abzulesen bzw. zu belegen. Hier sei nur das Verfahren beschrieben, um angelieferte, komplexe CoSim-Datenwerte in die zugehörige Java-Datenstruktur zu konvertieren. Die Rückrichtung von Java-Datenstrukturen zu komplexen CoSim-Datenwerten erfolgt analog.

Die benutzerdefinierte Java-Klasse zur Aufnahme von Kontextwerten ist bei der Reflexion im Feld `Model_Connection.responded` abgelegt worden. Erhält Cage eine komplexe Cosim-Struktur (`Guided_Value`), wird ein neues Exemplar dieser Java-Klasse erzeugt und mit den zugehörigen Werten aus der Cosim-Struktur gefüllt. Der Hauptwert `Guided_Value.simple` gelangt dabei durch Verwendung der `java.lang.reflect.Field.set`-Methode immer in das Feld `value` des neuen Exemplars.

Die Reflexion der Modelldefinition erzeugt eine Abbildung (`HashMap`) von Zugangspunkten zu Feldern der Java-Klasse, wobei die Objektreferenz der Zugangspunkte aus dem Lookup-Repository als Schlüsselwert dient. Sie ist im Feld `Model_Connection.responded_fields` der reflektierten Zwischenrepräsentation abgelegt. Um die CoSim-Kontextwerte (`Guided_Value.contexts`) in die Java-Klasse zu übertragen, wandelt Cage den Schlüssel eines Kontextwertes, der als

`Lookup_Connection` gegeben ist, in seine IOR-Codierung um, und schlägt mit diesem Wert das zu belegende Feld in der `HashMap` nach.

Vor der Belegung mit Haupt- oder Kontextwerten muss das Unmarshalling der Datenwerte erfolgen. Ein CoSim-Datenwert ist als `::CORBA::Any` verpackt und soll in seine Java-Repräsentation überführt werden. Hier wird die Hilfsklasse `bachmann.tools.Any_Extractor` eingesetzt, die für die CORBA-Basistypen eine Fallunterscheidung durchführt und die zugehörige `Any`-Konvertierungsfunktion aufruft (für Boole'sche Werte beispielsweise `Any.extract_boolean`). Modellspezifische Datentypen müssen eine `Helper`-Klasse bereitstellen (Abschnitt 8.3.3), die auch das Unmarshalling erledigt. Hier ruft der `Any_Extractor` einfach deren `Helper.extract`-Methode auf.

8.3.5 Automatische Uhrzeit-Aufrufe

Die CoSim-Schnittstellen eines Modellexemplars erlauben zwar die Angabe mehrerer Zugangspunkte zur Simulationsuhr, je Modellexemplar soll jedoch die Simulationsuhr selbst nur ein einziges Mal vorhanden sein (vgl. Kommentare zur Definition `Model_Class.clocks` in der Datei *cosim_ class.idl*, Anhang A). Diese Uhr kann man als Variable eines Modellexemplars auffassen, die auf verschiedene Arten abgelesen oder verändert werden kann. Genau dieser Mechanismus lässt sich mit Cage automatisch realisieren. Die Methoden `get_time` und `set_time` aus der Java-Schnittstellen `Model_Support` (Anhang B) erlauben hier den synchronisierten Zugriff auf eine spezielle Variable der Modellrepräsentation, die Cage intern verwendet. Die Methode `sync_time` liefert zusätzlich noch das Java-Objekt, das tatsächlich für die zugehörigen Java-`synchronized`-Blöcke herangezogen wird. Werden mit der Methode `run_thread` modellspezifische Java-Threads gestartet, so lassen sie sich auch mit der Simulationsuhr synchronisieren.

Alle vier Methoden der Schnittstelle `Modell_Support` erzeugt Cage regelmäßig mit der neuen Unterklasse, die in Abschnitt 8.3.3 eingeführt wurde. Der nachfolgende Auszug aus einer erzeugten Unterklasse zeigt, dass dabei ein einfacher Delegationsmechanismus (Gamma u. a. 1996, S. 23ff.) verwendet wird, der die Aufrufe an die zuständige interne Repräsentation weiterleitet:

```
public Object sync_time (        ) {return connector. impl. time;}
public double  get_time (        ) {return connector. impl. get_time ( );}
public void    set_time (double t) {       connector. impl. set_time (t);}
public Thread run_thread (String name, Runnable runner) {
  return this. connector. impl. run_thread (name, runner);}
```

Auch für die acht Kommunikationsmuster nach Abbildung 7.2 lassen sich die Implementationen automatisch in der Unterklasse erzeugen. Dazu muss in jeder Deklaration eines Zugangspunktes, der diese Code-Erzeugung wünscht, eine statische Methode nach dem Muster `let_`*`name`*`_auto` den Boole'schen Wert `true` liefern. In der Datei Check_Declaration.java (Anhang B) sind alle acht Muster beispielhaft für den Zugang zur Simulationsuhr genau einmal deklariert. Als Datentyp steht hier jeweils `int`. Cage muss also für die Einbettung in den intern verwendeten Typ `double` sorgen.

Nachfolgend ist kurz der Code beschrieben, den Cage für die verschiedenen Kommunikationsmuster erzeugt. Da sich die Varianten für synchrone und asynchrone Muster kaum unterscheiden, wenn die Aktivität (aktiv/passiv) und die

Zugangsrichtung (Export/Import) gleich bleiben, ist nur die synchrone Variante aufgeführt.

Der einfachste Code ergibt sich aus den Mustern für passive Eingaben. Hier erfolgt der parametrisierte Methodenaufruf von außerhalb des Modellexemplars durch den Client. Der übergebene Parameter für den Ziel-Zeitpunkt wird direkt an die Schreib-Methode `set_time` weitergereicht:

```
public void set_c_set_clock (int param) {this. set_time (param);}
```

Eine passive Ausgabegröße liefert auf Anfrage die aktuelle Simulationszeit. Auch hier erfolgt ein direkter Aufruf der Lese-Methode `get_time`. Nun muss der Rückgabewert noch in den gewünschten numerischen Datentyp umgewandlet werden:

```
public int get_c_get_clock () {
    int    _result = (int) this. get_time ();
    return _result;}
```

Soll das Modellexemplar aktiv an der Übertragung der Simulationszeit mitwirken, erzeugt Cage eine zusätzliche Methode zur Uhrzeit je Zugangspunkt, für den eine `_auto`-Methode entsprechend deklariert wurde. Diese Methode führt die notwendige Kommunikation mit dem Client des Modellexemplars in regelmäßigen Abständen innerhalb eines separaten Java-Threads durch. Nach Abschnitt 8.3.3 werden die zugehörigen Kommunikationsmethoden ebenfalls in der Unterklasse bereitgestellt. Um die Anzahl dieser Aufrufe zu minimieren, speichert Cage den zuletzt übertragenen Wert in einer weiteren Variablen. Die Werte aktiver Eingabevariablen werden also nur an die interne Uhrzeitvariable weitergegeben, wenn tatsächlich ein neuer Wert empfangen wurde. Dabei nimmt die Simulationszeit nur zu und niemals ab:

```
protected double store_see_c_see_clock = Double . NEGATIVE_INFINITY;
protected void   cycle_see_c_see_clock () {
  ;        this. store_see_c_see_clock = the_c_see_initial ();
  while (this. connector. impl. state == Model_State . RUNNING) {
    try {
      int time = see_c_see_clock ();
      synchronized (this. sync_time ()) {
        if (           this. store_see_c_see_clock < time) {
          ;            this. store_see_c_see_clock = time;
          ; set_time  (this. store_see_c_see_clock);}}
        Thread . sleep (100);}
      catch (Exception e) {
        e. printStackTrace (this. connector. logger. err);}}}
```

Auch der Code für aktive Ausgabevariablen berücksichtigt den letzten Wert und ruft die Sendemethode nur auf, falls ein höherer Wert der Simulationszeit vorliegt:

```
protected double store_let_c_let_clock = Double . NEGATIVE_INFINITY;
protected void   cycle_let_c_let_clock () {
  ;        this. store_let_c_let_clock = the_c_let_initial ();
  Object        sync = this. sync_time ();
  synchronized (sync) {
```

```
    while (this. connector. impl. state == Model_State . RUNNING) {
      try {
        int time = (int) this. get_time ();
        if (this. store_let_c_let_clock < time) {
          ; this. store_let_c_let_clock = time;
          ; this.       let_c_let_clock  (time);}
        else {
          sync. wait ();}}
      catch (Exception e) {
        e. printStackTrace (this. connector. logger. err);}}}}
```

Cage schaltet der modellspezifischen Implementation für die Durchführung eines Simulationslaufs die eben beschriebenen Zyklen für alle aktive Zugänge vor, in dem die entsprechende Methode `execute` überschrieben wird. Weiterhin wird die Simulationszeit initialisiert und zwar mit dem Wert der ersten Deklaration im zugehörigen Java-Quellcode, die eine Initialisierungsmethode enthält:

```
public void execute () throws State_Exception {
  set_time (the_c_let_initial ());
  run_thread ("let_c_let_clock",
              new Runnable(){public void run(){cycle_let_c_let_clock();}});
  run_thread ("see_c_see_clock",
              new Runnable(){public void run(){cycle_see_c_see_clock();}});
  run_thread ("pub_c_pub_clock",
              new Runnable(){public void run(){cycle_pub_c_pub_clock();}});
  run_thread ("rec_c_rec_clock",
              new Runnable(){public void run(){cycle_rec_c_rec_clock();}});
  super. execute ();}
```

8.4 Der CoSim-Component-Connector

Zur Kopplung bestehender Modellkomponenten stellte Abschnitt 7.3 den CoSim-Component-Connector (CoCo) vor. CoCo enthält zwei Teile. Mit einem Grapheneditor lassen sich Modellkopplungen spezifizieren und in eine Zwischenrepräsentation überführen. Die Ausführungsumgebung sorgt dann für die Interpretation der Kopplungsspezifikationen.

8.4.1 Der Grapheneditor

Für den Grapheneditor stand bereits mit `JHotDraw` (www.jhotdraw.org) ein umfangreiches Rahmenwerk zur Verfügung, das die Kernfunktionalität zum Aufbau von Graphen aus Knotensymbolen und Linienverbindungen enthält. Da die CoCo-Symbolsprache jedoch mit den Linienverbindungen auch auf der konzeptuellen Ebene eigenständige, attributbehaftete Objekte (Datenpuffer) verbindet, mussten im `JHotDraw`-Quellcode Änderungen vorgenommen werden. Dabei wurde der Sichtbarkeitsbereich einiger Objektmethoden und -variablen von `private` auf `protected` erweitert, um in eigenen Unterklassen Erweiterungen zu realisieren, die auf die bestehenden Methoden und Variablen zugreifen. Ohne diese Änderungen wäre es nur mit viel Aufwand möglich gewesen, durch Doppelklicks Dialoge für Linienverbindungen zu öffnen oder mehrzeilige Knotenbeschriftungen in das Rahmenwerk einzubringen.

Weiterhin erlaubt der Graphheneditor, die erstellten Diagramme als Bilddatei zu speichern. Die verwendete Bibliothek `JIMI` (Java Image Manipulation Interface, http://java.sun.com/products/jimi) unterstützt dabei verschiedene Ausgabeformate (`PNG`, `JPEG` u.a.).

8.4.2 Die Zwischenrepräsentation

Die Zwischenrepräsentation als `.coco` Datei enthält den Kopplungsgraphen als serialisiertes Java-Objekt der Klasse `Checked_Graph` aus Anhang C. Abschnitt 7.3.7 stellte bereits Klassen vor, die für Knoten mit textueller Berechnungsdefinition erzeugt werden. Die erzeugten Klassen enthalten die benutzerdefinierten Texte der Methodenrümpfe zu den automatisch aus der CORBA-Typdeklaration ermittelten Methodengerüsten (Abschnitt 7.3.5). Hash-Codes im Klassen- und in den Methodennamen sorgen hier für die Vermeidung von Namenskonflikten.

Der Konstruktor dieser Klassen baut jeweils eine Zuordnungstabelle auf, in der zum Hauptwert und zu jedem Kontextwert die Namen der aufzurufenden Methoden sowie die zugehörigen Zugangspunkte notiert sind, um die entsprechenden Werteteile zu ermitteln. Die gewünschten Datenwerte lassen sich dann immer auf die gleiche Weise durch Aufruf der angegebenen Methoden per Java-Reflexion berechnen, so dass eine gemeinsame Oberklasse aller erzeugten Klassen ausreicht, um die geforderten Methoden der Schnittstellen `Calculation_Value` bzw. `Calculation_Context` zu implementieren.

Im Quellcode der CoCo-Realisierung unter `http://cosim.sf.net` enthält die Implementationsklasse `de.unihh.informatik.bachmann.coco.evaluation.Calculation_Evaluation` bereits ein einfaches Beispiel für das Muster zur Quellcode-Erzeugung. Hier sind ein Hauptwert (Methode `node__0815`) des Berechnungsknotens und ein Kontextwert (Methode `context__0007`) gefordert. In der Tabelle `methods` speichert der Konstruktor alle Methoden und in der Tabelle `lookups` die Zugangspunkte für Kontextwerte. Das Feld `name` hält den Namen des Berechnungsknotens fest. Der vom Benutzer eingestellte CORBA-Typ für den Hauptwert ist im Feld `typestring` enthalten. In diesem Fall handelt es sich um einen primitiven Datentyp, der unter der Ordnungszahl `21` im angegebenen CORBA Interface Repository zu finden ist.

```
public String calculate_0815 (Environment env) {return "";}
public double calculate_0007 (Environment env) {return -1;}

public Calculation_Evaluation () {
  this.name      ="node_0815";
  this.typestring="21";
  this.repository="corbaname::asipc6#Services.ctx/Repository.svc";
  this.lookups.put("context_0007","[...]#Model.ctx/0007.context");}
  this.methods.put("context_0007","calculate_0007");
  this.methods.put(   "node_0815","calculate_0815");
```

8.4.3 Die Ausführungsumgebung

Die Ausführungsumgebung liest die `.coco` Dateien ein und erzeugt daraus funktionsfähige Modellfabriken. Bevor ein Modellexemplar aus der Modellfabrik den Simulationsbetrieb aufnehmen kann, sind einige Vorbereitungen notwendig. Ein

Teil der Vorbereitungen kann bereits nach dem Einlesen der Kopplungsspezifikation erfolgen, ein anderer Teil erfolgt erst mit der Erzeugung jedes Modellexemplars. Zudem führt jede Änderung der Nutzungsphase zu weiteren Aktionen.

Einlesen der Kopplungsspezifikation

Die statischen Beziehungen aus der Kopplungsspezifikation führen beim Einlesen zum Aufbau verschiedener Zuordnungstabellen. Diese Tabellen ermöglichen den schnellen Zugriff auf regelmäßig benötigte Beziehungen, ohne die notwendigen Aufrufe auf entfernte Objekte erneut durchführen zu müssen. Die Tabellen umfassen Zuordnungen von

- Modellknoten zu Modellfabriken
- Modellknoten zu Modellbeschreibungen
- Zugangsknoten zu Zugangspunkten,
- Zugangspunkten zu Zugangsknoten,
- Zugangspunkten zu ihren Kategorien,
- Berechnungsknoten zu Berechnungsklassen,
- Klassenpfaden zu Klassenladern und
- Knotennamen zu Knoten.

Nach dem Tabellenaufbau erzeugt CoCo dann die Beschreibung des gekoppelten Modells und seiner Zugangspunkte in einem eigenen Unterkontext eines Namensdienstes. Die Struktur des Kopplungsgraphen wird dabei ausgewertet, um die Abhängigkeiten zwischen den externen Zugangspunkten anzugeben.

Erzeugen von Modellexemplaren

Jedes Modellexemplar, das mit Hilfe einer CoCo Modellfabrik gewonnen wurde, benötigt eigene Sende- und Empfangsobjekte, um die Kommunikation mit den benutzten, untergeordneten Modellexemplaren und mit seiner Außenwelt zu betreiben. Diese Kommunikationsobjekte entstehen zusammen mit dem Modellexemplar und werden dann den entsprechenden Nutzungsphasen zugeordnet. Die Ausführungsumgebung erzeugt auch die Datenpuffer aus Abschnitt 7.3.4 gemeinsam mit dem Modellexemplar. Ebenso werden alle eingebetteten Modellfabriken beauftragt, die untergeordneten Modellexemplare zu erzeugen.

Um die unterschiedlichen Kommunikationsarten zu überwinden, benutzt CoCo neben den als `Queue` implementierten Datenpuffern mehrere Schnittstellen, die die Datenübertragung zwischen den Zugangspunkten erledigen. Alle Zugangspunkte, die aktiv eine Datenübertragung veranlassen, greifen dabei direkt auf die zugehörigen `Queues` zu. Abbildung 8.4 zeigt die Schnittstellen `Export`, `Import` und `Runnable` für passive Zugangspunkte, die jeweils in mehreren Unterklassen an die verschiedenen Gegebenheiten angepasst werden. Über die Schnittstelle `Export` lassen sich mit der Methode `get` Daten aus Zugangspunkten abfragen. Dies kann entweder durch lose Kopplung mit Hilfe von Nachrichtenkanälen geschehen (Klasse `Loose`), oder mit Hilfe des CoSim-Rückrufmechanismus `Guided_Value_Supplier` bei enger Kopplung. Da die externen Zugangspunkte selbsttätig für den Datenabruf aus ggf. mehreren Sendern sorgen müssen, und interne Zugangspunkte nur einen einzigen Sender darstellen, unterscheidet CoCo hier die zwei Klassen `Tight_External`

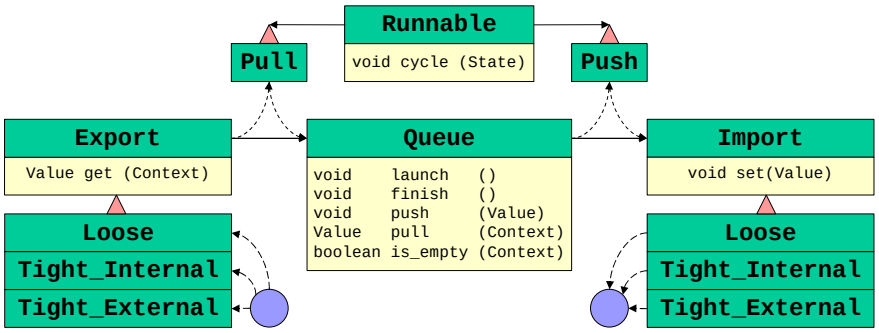


Abbildung 8.4: Queue

und `Tight_Internal`. Dieselben Unterscheidungen fallen auch auf der Seite des Datenversandes über die Schnittstelle `Import` an.

Regelmäßige Abfragen der Zugangspunkte oder der Datenpuffer auf neue Daten erledigt CoCo mit Hilfe der Schnittstelle `Runnable`. Die enthaltene Methode `cycle` ruft CoCo zyklisch auf, während die Nutzungsphase `State` andauert. Objekte der Klasse `Pull` greifen dabei auf Zugangspunkte zu und schreiben Daten in die `Queue`. Umgekehrt lesen Objekte der Klasse `Push` Daten aus der `Queue` und versenden sie über die zugehörigen Zugangspunkte.

Da die Erzeugung der notwendigen Kommunikationsobjekte viele Fallunterscheidungen auf der Basis von Klassen erfordert, bietet es sich an, je Fall eine eigene Methode zu implementieren. Dann können auch Ober-/Unterklassenbeziehungen ausgenutzt werden. Jedoch bietet die zur Implementation von CoCo verwendete Laufzeitumgebung der Programmiersprache Java einen Polymorphiemechanismus nur für das primäre Ziel eines Methodenaufrufs an. Die Argumente eines Methodenaufrufs erfasst dieser Mechanismus nicht. Bei der Auswahl (`Dispatch`) einer Methodenimplementation zieht die Java Laufzeitumgebung nur den Objekttyp des Objektes heran, auf dem die Methode aufgerufen wird. Die Objekt- oder Basisdatentypen der Argumente wurden bereits im Java Compiler verarbeitet, der auch für die Vorselektion aufrufbarer Methoden sorgt. Abhilfe schafft ein generischer Auswahlmechanismus für Methoden, der als Klasse `de.unihh.informatik.bachmann.tools.Generic_Dispatch` im Quellcode vorliegt und auch in einigen anderen Werkzeugen eingesetzt wird. Der neue Dispatcher verwendet intensiv die `java.lang.reflect` API, um sämtliche anwendbare Methoden eines Aufrufs zu ermitteln und aufzurufen, nicht nur die vom Compiler vorselektierten Methoden.

Wechsel der Nutzungsphase

Jeder Wechsel in der Nutzungsphase eines CoCo Modellexemplars muss sich auch in den untergeordneten Modellexemplaren fortsetzen. Die Reihenfolge der einzelnen Vor- und Nachbereitungsschritte ist dabei wesentlich, um einen geordneten Ablauf zu gewährleisten.

1. Direkt nachdem ein CoCo Modellexemplar einen Wechsel der Nutzungsphase akzeptiert hat, schließt es sämtliche Datenpuffer, die in der aktuellen Nutzungsphase aktiv waren. Damit werden weder neue Daten angeliefert noch versendet.

2. Die `Supplier` und `Consumer` Objekte melden sich bei ihrer Gegenstelle ab. Bei der losen Kopplung ist die Gegenstelle das Proxyobjekt im Nachrichtenkanal (vgl. Abbildung 6.9). Die enge Kopplung verwendet die untergeordneten Modellexemplare selbst als Gegenstellen.

3. Die Datenpuffer entfernen ggf. Restdaten ohne weitere Bearbeitung.

4. Nachrichtenkanäle der alten Nutzungsphase, die CoCo selbst erzeugt hat, werden vernichtet und neu erzeugt, damit auch hier keine Restdaten mehr auftauchen können.

5. CoCo beauftragt alle untergeordneten Modellexemplare, den Wechsel der Nutzungsphase zu vollziehen. Erst wenn alle beauftragten Modellexemplare den Vollzug gemeldet haben, ist dieser Schritt abgeschlossen.

6. Alle Datenanfragen sind über ein spezielles Verwaltungsobjekt synchronisiert und müssen warten, solange ein Wechsel der Nutzungsphase noch nicht abgeschlossen ist. Dieses Verwaltungsobjekt benachrichtigt alle wartenden Anfragen nach dem Aufruf der Methode `notifyAll`.

7. Der CORBA `TypedEventChannel` erhält die Wechselnachricht über die Methode `state_changed` mitgeteilt.

8. Die Datenpuffer werden wieder freigegeben. Bereits eingegangene Anfragen der untergeordneten Modellexemplare mussten bis zu diesem Zeitpunkt warten und lassen sich nun bearbeiten.

9. CoCo startet die regelmäßigen Anfragen an Datenpuffer und Zugangspunkte in eigenen Threads.

10. Alle Berechnungsknoten erhalten über die Methode `state_changed` eine Mitteilung über den Wechsel der Nutzungsphase.

11. CoCo ermittelt die Nachrichtenkanäle, die von untergeordneten Modellexemplaren neu erzeugt wurden, und ändert die Einträge in der zugehörigen Zuordnungstabelle.

9 Realisierung des Fässermodells

Dieses Kapitel beschreibt, wie die CoSim-Umgebung in einem speziellen Anwendungsfall eingesetzt werden kann: Für das Beispielmodell zur Fassbefüllung aus Kapitel 3 sind drei verschiedene Modellteile (Bestellung/Transport, Befüllung und Beladung) sowie ein gekoppeltes Gesamtmodell zu realisieren. Jeder Modellteil verwendet dabei unterschiedliche Werkzeuge zur Implementation. Der Transportteil steht als Java-Klasse zur Verfügung, für die das Werkzeug Cage eine Modellfabrik erzeugt. Die Realisierung des Befüllungsmodells erfolgt manuell in der Programmiersprache Common Lisp. Die beiden Modellteile Transport und Befüllung verbindet die Beladungskomponente, die als Berechnungsknoten mit Hilfe des Rahmenwerks aus dem Werkzeug CoCo in eine Kopplungsspezifikation eingebettet wird.

Da für jeden Modellteil unterschiedliche Werkzeuge eingesetzt wurden, müssen die Schnittstellen der Modellteile untereinander exakt definiert und für die jeweils anderen Modellteile offen gelegt sein. Die externe Repräsentation dieser Schnittstellen erfolgt jeweils durch die Modellbeschreibung eines Teilmodells und deren Zugangspunkte. Den Zugangspunkten sind teilweise modellspezifische Datentypen zugeordnet, die ebenfalls definiert und zugänglich sein müssen. Die Typdefinition erfolgt in der Beschreibungssprache CORBA-IDL. Ein CORBA Interface Repository enthält dann diese Definitionen (vgl. Abschnitte 6.1.4 und 7.2.3).

Die vollständige Realisierung der gekoppelten Modellkomponenten zeigt, dass die CoSim-Werkzeuge modellunabhängig eingesetzt werden können, denn keines der Werkzeuge verwendet direkt die neuen Datentypen, die erst mit dem Fässermodell entstanden sind. Weiterhin stößt die Experimentierumgebung die Ausführung von Simulationsläufen wie gewünscht an und erlaubt dabei ebenso eine typunabhängige Parametrisierung und Beobachtung von Zugangspunkten. Auch die Laufzeitumgebungen der Werkzeuge Cage und CoCo vollziehen sämtliche Datenübertragungen in der gewünschten Form. Sie unterstützen alle Arten der Datenübertragung gemäß Abschnitt 6.5.

9.1 Modellspezifische Datentypen

Für die problemangepasste Modellierung des Fässerbeispiels enthalten die Dateien `barrels.idl` und `desmoj.idl` aus Anhang A die notwendigen Typdefinitionen. Die `barrels` Definitionen wurden nur für dieses fiktive Beispiel entworfen. In einem realen Kontext erfolgt üblicherweise zunächst eine Analyse der Anwendungsdomäne, um die wiederverwendbaren Objekte und Funktionalitäten herauszuarbeiten. Die `desmoj` Definitionen stellen solche Objekte bereit. Sie beschreiben die Struktur von Statistikobjekten, die in vielen Anwendungsfällen benötigt werden.

Zentrale Gegenstände des Beispielmodells sind die Bierfässer. Damit einzelne

Fässer unterscheidbar sind, wurde ein CORBA-Objekttyp gewählt. Die zugehörige IDL Definition `interface Barrel` enthält genau eine Methode zum Ablesen der Fassgröße. In der Realität werden 20, 30 und 50 Liter als übliche Fassgrößen verwendet.

Die Abfüllstationen arbeiten je nach Maschinenbauweise mit unterschiedlichen Fülldrücken. Jede Station besitzt zwei Kenngrößen (maximaler und minimaler Fülldruck), zwischen denen der aktuelle Druck der Fassbefüllung linear variiert wird. Eine `struct Pressure` Definition fasst diese beiden Kenngrößen zusammen.

Um die Beladung eines Transportfahrzeugs zu beschreiben, wurde eine Ladeliste eingeführt. Jede Position der Ladeliste enthält Angaben über die zugehörigen Fässer dieser Position wie Fassgröße, Fassanzahl und eine Markierung, ob die Fässer gefüllt sind oder Leergut darstellen. Eine Position ist als zusammengesetzter Datentyp `struct Lot` definiert, die Ladeliste wird notiert als Liste aus Positionen (`sequence <Lot>`). Fahrzeuge sind genau wie Fässer einzeln unterscheidbar und als CORBA-Objekttyp `interface Vehicle` mit dem Nur-Lese-Attribut Ladeliste definiert.

Das Teilmodell Bestellung stellt die Verbrauchsereignisse als expontialverteilte Zufallsvariablen dar und verwendet dafür die zusammengesetzte Datenstruktur `struct Exponential_Distribution`, die den Startwert eines Zufallszahlengenerators und den Mittelwert der Verteilung umfasst. Die Lieferzeiten des Teilmodells benötigen normalverteilte Variablen (`struct Normal_Distribution`). Dieser Verteilungstyp enthält ein zusätzliches Attribut für die Standardabweichung der Verteilung. Die Datenstruktur der Kunden setzt sich dann zusammen aus den beiden Zufallsvariablen für Lieferzeit und Verbrauchszeit, sowie der Verbrauchseinheit, der Bestellgrenze, der Lagergröße und der verwendeten Fassgröße (`struct Consumer`). Jeder Kunde verwendet nur Fässer einer bestimmten Größe. Um bei stark unterschiedlichem Konsumverhalten ähnlich viele Konsumereignisse je Kunde im Bestellteil zu erhalten, dienen die Verbrauchseinheiten als Skalierungsmechanismus. Je Kunde wird also angegeben, wie viele Fässer ein einzelnes Konsumereignis betrifft. Zu Beginn der Simulation benötigt das Transportunternehmen die Liste aller Kunden, deshalb existiert auch noch die entsprechende `sequence <Consumer>`.

Die Beladungskomponente kann unterschiedliche Strategien für die Priorität bei der Befüllung von Fässern und bei der Beladung von Fahrzeugen verwenden. Die Menge jeweils möglicher Strategien geben zwei Aufzählungstypen an (`enum Vehicle_Strategy` und `enum Barrel_Strategy`, vgl. Abschnitt 9.4).

Die einzelnen Modellteile berechnen bereits Statistiken für einige interne Warteschlangen oder Zähler und geben sie in Form von Zugangspunkten für Simulationsergebnisse weiter. Die zusammengesetzte Datenstruktur `struct Tally_Statistics` für eine nicht zeitgewichtete Zählerstatistik enthält Werte für Minimum, Maximum, Mittelwert, Standardabweichung, letzten Zählerstand und Anzahl der Beobachtungen. Die entsprechende `struct Queue_Statistics` für zeitgewichtete Warteschlangenstatistik setzt sich zusammen aus minimaler, maximaler, mittlerer und letzter Warteschlangenlänge, maximaler und mittlerer Wartezeit sowie Anzahl der Einträge ohne jegliche Wartezeit und Anzahl der Beobachtungen insgesamt.

9.2 Modellkomponente „Transport“

9.2.1 Zugangspunkte des Transportteils

Die Transportkomponente sorgt für die Belieferung der Zwischenhändler mit Bierfässern. Dazu tauscht sie mit der Beladungskomponente über drei Zugangspunkte Daten aus. Der Zugangspunkt `order` sendet die Bestellungen der Zwischenhändler an die Beladungskomponente. Sobald ein Fahrzeug dort mit vollen Fässern beladen ist, erhält die Transportkomponente über den Zugangspunkt `delivery` die Vollzugsmeldung. Der Zugangspunkt `order` sendet ein Fahrzeug dann mit Leerfässern wieder an die Transportkomponente zurück.

Name	Phase	Richtung	Kopplung	Aktivität	Datentyp
`simend`	Parameter	Eingabe	eng	-	`double`
`vehicles_count`	Parameter	Eingabe	eng	-	`int`
`vehicles_capacity`	Parameter	Eingabe	eng	-	`long`
`consumers`	Parameter	Eingabe	eng	-	`Consumer[]`
`observe`	Uhrzeit	Ausgabe	eng	+	`double`
`control`	Uhrzeit	Eingabe	eng	-	`double`
`order`	Laufzeit	Ausgabe	eng	+	`Vehicle`
`delivery`	Laufzeit	Eingabe	eng	-	`Vehicle`
`return`	Laufzeit	Ausgabe	eng	+	`Vehicle`
`returns`	Ergebnis	Ausgabe	lose	-	`Tally_Statistics[]`
`deliveries`	Ergebnis	Ausgabe	lose	-	`Tally_Statistics[]`

Tabelle 9.1: Zugangspunkte der Transportkomponente

Jeder der drei Zugangspunkte `order`, `delivery` und `return` bekommt zusätzlich zum Hauptwert vom Typ Fahrzeug noch einen Zeitstempel als Kontextwert übergeben, damit der Ereigniszeitpunkt zwischen den beteiligten Modellkomponenten ausgetauscht wird. Als Schlüssel für den Zeitstempel dient der Zugangspunkt `observe`, durch den sich die aktuelle Simulationszeit der Transportkomponente auch direkt ablesen lässt. Als weitere Uhrzeitvariable ist `control` vorhanden. Hiermit kann der Fortschritt der Simulationszeit beschränkt und somit ein Simulationslauf interaktiv gesteuert werden. Die Transportkomponente wird nur solche Simulationsschritte berechnen, die nicht später als der letzte über `control` gesendete Zeitpunkt liegen.

Parametrisiert wird die Transportkomponente über die Zugangspunkte `simend`, `vehicles_count`, `vehicles_capacity` und `consumers`. Der Zugangspunkt `simend` gibt an, welches der letzte zu berechnende Zeitpunkt sein wird. Ist dieser Zeitpunkt erreicht, so stellt die Transportkomponente alle weiteren Laufzeitberechnungen ein und bereitet die Ermittlung von Simulationsergebnissen vor. Die Anzahl aller Fahrzeuge ist über den Parameter `vehicles_count` einstellbar. Alle Fahrzeuge besitzen die gleiche maximale Anzahl an Fässern, mit denen sie beladen werden können (Parameter `vehicles_capacity`). Der Parameter `consumers` enthält die Liste aller Bestell- und Verbrauchsdaten der Zwischenhändler.

Als Simulationsergebnisse bietet die Transportkomponente zwei Zählersta-

tistiken an. Der Zugangspunkt `returns` zählt die Anzahl zurückgegebener Leerfässer je Fahrt. Die ausgelieferten vollen Fässer je Fahrt sind über den Zugangspunkt `deliveries` erreichbar.

9.2.2 Kern des Transportteils

Die Realisierung der Kernfunktionalität im Transportteil erfolgte als prozessorientiertes Modell mit Hilfe des Rahmenwerks Desmo-J (Page u. a. 2000, Abschn. 4.3.6). Der Kern führt drei neue Klassen ein (`Consumer`, `Order` und `Vehicle`), die das Verhalten der beteiligten Prozesse beschreiben.

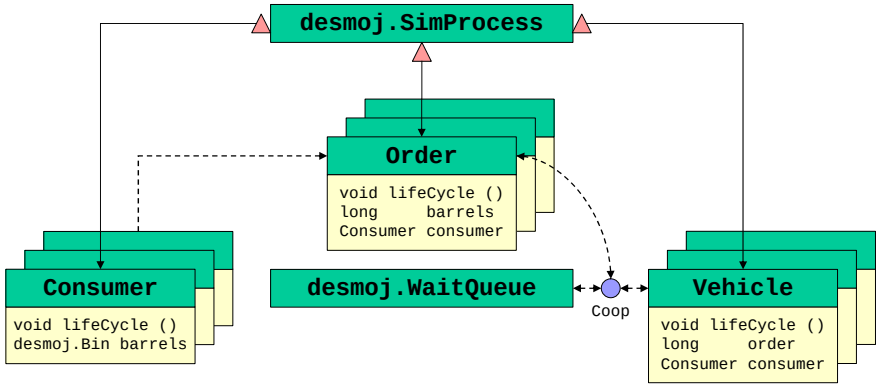


Abbildung 9.1: Kooperation der Transportprozesse

Je Kunde aus der Parameterliste des Transportteils existiert ein eigener `Consumer` Prozess, der das Lager des Kunden verwaltet und dafür ein höheres Modellierungskonstrukt für Produzenten-/Konsumentenbeziehungen aus dem Desmo-J-Rahmenwerk verwendet (`desmoj.Bin`, Page u. a. 2000, Abschn. 4.4.2). Der Lebenszyklus dieses Prozesses wiederholt sich bis zum voreingestellten Ende der Simulationszeit. Zunächst wird dazu geprüft, ob eine Bestellung auf Grund des Lagerbestandes notwendig ist. Falls der Lagerbestand unterhalb der Bestellgrenze liegt, entsteht ein neuer `Order` Prozess, der mit dem aktuellen Kunden verknüpft ist. Anschließend wartet der Kundenprozess für die Dauer eines Konsumereignisses.

Jeder `Order` Prozess existiert nur, um eine einzige Bestellung abzuarbeiten. Er tritt als Master einer zentralen Warteschlange auf (`desmoj.WaitQueue`, Page u. a. 2000, Abschn. 4.4.4) und kooperiert mit einem möglichen Fahrzeugprozess. In der zugehörigen Kooperation informiert der `Order` Prozess das Fahrzeug, welcher Kunde wie viele Fässer benötigt.

Ein `Vehicle` Prozess wickelt bis zum voreingestellten Ende der Simulationszeit nacheinander jeweils einen bestimmten Kundenauftrag ab. Das Fahrzeug startet auf dem Verladehof des Abfüllbetriebs. Zu Beginn jedes Zyklus wartet es in der zentralen Warteschlange als Slave, bis eine Bestellung beim Beladeteil eingegangen ist und es mit der bestellten Anzahl an Fässern beladen wurde. Nachdem die Fahrtzeit zum Kunden vergangen ist, lädt das Fahrzeug die Fässer in der zugehörigen `desmoj.Bin` ab. Es ermittelt gleichzeitig die beim Kunden vorhandenen Leerfässer, fährt damit zum Abfüllbetrieb zurück und lädt die Fässer zur Wiederbefüllung dort ab. Über eine `desmoj.Queue` führt das Fahrzeug

eine Statistik bzgl. der Lieferzeiten zum Kunden. Bei jeder Beladung trägt es sich in diese Warteschlange ein und nach der Hinfahrt zum Kunden wieder aus. Eine Statistik über die zurückgegebenen Fässer je Kunde führt das Fahrzeug ebenfalls fort.

Das Transportmodell verwendet eine Technik, die ähnlich zu Nicols (1988) Future Lists ist. Dabei werden die demnächst benötigten Zufallszahlen (Fahrzeiten, Konsumzeiten) im Voraus berechnet und gespeichert. Damit kann den logischen Prozessen aus anderen Modellteilen bei der Synchronisation ein höherer Zeitstempel garantiert werden.

9.2.3 CoSim-Anbindung mit Cage

Der CoSim Adapter Generator stellt die Kernfunktionalität des Transportmodells als CoSim-Modellfabrik zur Verfügung. Die Klasse `de.unihh.informatik.bachmann.barrels.wrapper.Transport_Wrapper` (unter http://cosim.sf.net) enthält die notwendigen Methoden gemäß Abschnitt 7.1 für die Anbindung an das Cage-Rahmenwerk. Diese Modellhülle verwaltet sämtliche CORBA-Objektreferenzen der Fahrzeuge, die mit Bestellungen oder mit Leergut-Rückgaben an die Beladekomponenten übergeben werden. Bei jeder Bestellung und bei jeder Rückgabe von Leergut sucht die Hülle das CORBA-Objekt zum Desmo-J-Fahrzeugprozess heraus und setzt die Ladeliste des Fahrzeugs auf die entsprechenden Werte aus dem Fahrzeugprozess.

Mit Eintritt in die Nutzungsphase `RUNNING` erzeugt die Modellhülle dann zusätzliche Desmo-J-Simulationsprozesse. Die Prozesse „Suspend“ und „Control“ tragen sich regelmäßig am Ende der Prozessliste wieder ein, nachdem sie sichergestellt haben, dass keine interaktive Unterbrechung vorliegt bzw. die Desmo-J-Simulationsuhr noch hinter dem letzten Wert des Zugangspunktes `control` liegt, der den Zeitfortschritt im Modell regelt. Der Prozess „Lower Bound Time Stamp - Order“ ermittelt regelmäßig den Lookahead für die Bestellungen. Dazu sucht der Prozess den nächsten Kunden auf der Desmo-J-Prozessliste und veröffentlicht dessen eingeplanten Simulationszeitpunkt, da kein anderer Kundenprozess vorher weitere Bestellungen tätigen kann. Ähnlich verfährt der Prozess „Lower Bound Time Stamp - Return“ mit dem Lookahead für mögliche Leergut-Rückgaben. Er addiert bei der Suche lediglich die vorher gespeicherte Dauer des nächsten Konsumereignisses hinzu. Damit die Desmo-J-Prozesse auch sämtliche Lieferungen voller Fässer berücksichtigen, sorgt der Prozess „Lower Bound Time Stamp - Delivery“ für eine Synchronisation der Lieferungen mit den anderen Prozessen.

Jede eingehende Lieferung bekommt die Hülle über eine zugehörige Cage-Rückrufmethode mitgeteilt. Sie aktiviert dann den Desmo-J-Fahrzeugprozess, der dem ankommenden Fahrzeug zugeordnet ist. Sämtliche Parameterwerte, die über die zugehörigen Zugangspunkte übertragen wurden, werden in Objektvariablen abspeichert und vor der Initialisierung jedes Simulationslaufs ausgewertet. Ergebnisanfragen nach der Statistik für Lieferzeiten und Leergutrückgaben ermittelt die Hülle direkt aus den Desmo-J-Statistiken, die jedes Fahrzeug einzeln führt.

9.2.4 Lokale Realisierung mit Desmo-J

Das Transportmodell lässt sich nicht nur als CoSim-Modellkomponente, sondern auch als eigenständiges Java-Gesamtmodell innerhalb einer einzigen virtuellen Maschine betreiben. Diese Variante entstand ursprünglich, um das Bestellverhalten im Transportteil lokal zu validieren, ohne die komplette CoSim-Umgebung zu starten. Einfache Geschwindigkeitsmessungen zeigten zudem die Performanzeinbußen auf, die bei Verwendung der CoSim-Umgebung auftreten. Sie belaufen sich etwa auf einen Geschwindigkeitsfaktor von 500 gegenüber der ausschließlichen Realisierung mit Java/Desmo-J. Selbstverständlich liefern die CoSim- und die Java-Variante identische Simulationsläufe, wobei die Java-Variante nur mit den festgelegten Strategien `Vehicle_First` und `Barrel_First` (vgl. Abschnitt 9.4.2) betrieben werden kann.

Dazu wurde die Beladekomponente mit fester Belade- und Befüllstrategie auf der Basis von Desmo-J nachgebildet und über zwei Java-Schnittstellen zum Beladen von Fahrzeugen (`Pick_Interface`) bzw. zur Rückgabe von Leergut (`Fill_Interface`) durch die Fahrzeugprozesse aus Abschnitt 9.2.2 mit dem Transportteil verbunden. Als zusätzliche Desmo-J-Prozesse tauchen in der Java-Variante die Füllstationen auf. Bis zum Ende der Simulationszeit entnimmt jeder Prozess für sich Leerfässer aus einer `desmoj.Bin` und berechnet die Befüllungszeit ähnlich wie im Abschnitt 9.3.2. Für gefüllte Fässer existiert je Fassgröße eine eigene `desmoj.Bin`. Nach einer Befüllung sortiert jeder Stationsprozess die vollen Fässer in die richtige Ablage ein.

Die lokale Variante realisiert die Schnittstelle `Fill_Interface` durch Einordnen von Leerfässern in `desmoj.Bin` Objekte anstatt durch Senden eines Fahrzeugs an die Beladekomponente. Für die Bedienung der Schnittstelle `Pick_Interface` muss ein Fahrzeugprozess nun auf volle Fässer in der Ablage warten anstatt auf Rückrufe durch die Beladekomponente.

9.3 Modellkomponente „Fill“

Eine Abfüllkomponente mit dem Namen „Fill“ sorgt für die Befüllung von leeren Bierfässern. Innerhalb des Gesamtmodells sind mehrere Abfüllstationen vorgesehen, deshalb werden auch mehrere Exemplare der Abfüllkomponente erzeugt.

9.3.1 Zugangspunkte im Abfüllteil

Die Abfüllkomponente benötigt den maximalen und minimalen Fülldruck als Eingabeparameter. Der Zugangspunkt `pressure` nimmt diese Werte als zusammengesetzte Datenstruktur entgegen. Der Simulationsfortschritt lässt sich wie im Transportmodell über den Zugangspunkt `control` aus der Kategorie „Uhrzeit“ steuern. Zur Laufzeit erhält die Abfüllkomponente die zu befüllenden Fässer über den Zugangspunkt `barrel_import`. Die befüllten Fässer gibt sie über `barrel_export` wieder zurück.

9.3.2 Kern des Abfüllteils

Die Abfüllkomponente wurde zunächst in der Programmiersprache Common Lisp realisiert, um zu demonstrieren, dass der Einsatz modellspezifischer Datenstrukturen auch unter verschiedenen Programmiersprachen funktioniert. Eine

Name	Phase	Richtung	Kopplung	Aktivität	Datentyp
`pressure`	Parameter	Eingabe	eng	-	`Pressure`
`control`	Uhrzeit	Eingabe	eng	-	`double`
`barrel_export`	Laufzeit	Ausgabe	eng	+	`Barrel`
`barrel_import`	Laufzeit	Eingabe	eng	-	`Barrel`

Tabelle 9.2: Zugangspunkte der Abfüllkomponente

geeignete CORBA-Umgebung für Common Lisp steht mit dem Produkt „LispWorks ORB$^{\text{TM}}$“ der Firma Xanalys auf verschiedenen Plattformen zur Verfügung. Da dieses kommerzielle Produkt nur wenig verbreitet ist, entstand zu Demonstrationszwecken des Fässermodells auch eine Umsetzung in Java mit der zugehörigen Cage-Hülle. Eine weitere Variante ermöglicht die vollständig lokale Ausführung des Fässermodells unabhängig von der CoSim-Umgebung (vgl. Abschnitt 9.2.4).

Für den Kern des Abfüllteils muss eine Differentialgleichung erster Ordnung gelöst werden, um den Befüllungsverlauf eines Fasses nachzubilden. Zur numerischen Lösung von Differentialgleichungen verwendet man Integrationsverfahren, die gesuchte Funktionswerte durch kleine Näherungsschritte berechnen. Hier fand ein Verfahren vom Runge-Kutta-Typ Anwendung (Bronstein und Semendjajew 1991, S. 770). Die Differentialgleichung

$$y'(x) = f(x, y(x)), y(a) = s$$

wird dazu angenähert durch

$$y_{j+1} := y_j + h\hat{f}(x_j, y_j); y_0 := s; j = 0, 1, ...$$

wobei

- h die Schrittweite des Integrationsverfahren angibt,
- $\hat{f} = \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)$,
- $K_1 = f(x_i, y_i)$,
- $K_2 = f(x_i + \frac{h}{2}, y_i + \frac{h}{2}K_1)$,
- $K_3 = f(x_i + \frac{h}{2}, y_i + \frac{h}{2}K_2)$ und
- $K_4 = f(x_i + h, y_i + hK_3)$.

Um die Fülldauer zu ermitteln, muss die Abfüllkomponente nun solange Integrationsschritte durchführen, bis ein Fass gefüllt ist. Nach dem letzten Schritt, der ein Fass üblicherweise schon überfüllt hat, interpoliert man noch mit dem Wert aus dem vorletzten Schritt, um die exakte Fülldauer zu erhalten. Unter Verwendung des Differentials aus Abbildung 3.4 ergibt sich dann folgender Lisp-Code:

```
(defun calculate (pressure size)
  (let* ((min          (op:minimum pressure))
         (max          (op:maximum pressure))
         (dt           (expt 0.5 4))
         (differential #'(lambda (x y)
```

```
                        (declare (ignore x))
                        (+ (* (- max min)
                              (/ (- size y) size))
                           min)))
      (integration  (make-instance 'runge-kutta-4
                                   :differential differential
                                   :dt           dt
                                   :x            0
                                   :y            0))
      (more         0.0)
      (less         size))

  (do () ((>= more size))
    (integrate integration)
    (setq less more)
    (setq more (y integration)))

  (let* ((fore (x integration))
         (back (- fore dt))
         (calc (+ (/ (* (- fore back)
                        (- size less))
                     (- more less))
                  back)))
    calc))
```

9.3.3 Umsetzung der Befüllung als Modellfabrik

Die Umsetzung der Abfüllkomponente als Modellfabrik in der Common Lisp Umgebung erfolgte manuell. Ein Rahmenwerk wie Cage für die Programmiersprache Java wurde hier nicht realisiert. Es entstanden also neue Klassen im Common Lisp Object System (CLOS), die dem CORBA-Lisp-Mapping (OMG 2000a) entsprechen und die notwendigen Schnittstellen einer Modellfabrik bzw. eines Modellexemplars umsetzen. Um die geforderten Schnittstellen-Methoden zu realisieren, enthalten diese Klassen hauptsächlich Deklarationen der Art `(corba:define-method op:`*`funktion`* `((self` *`klasse`*`) param1 ...))`.

Einen großen Teil der Befüllungskomponente nimmt der Aufbau der Modellbeschreibung und ihrer Zugangspunkte ein. Dazu konnte der Objekthüllendienst aus Abschnitt 6.4.2 benutzt werden. Die dort eingetragenen Datenstrukturen wurden direkt in Lisp codiert. Hier sei beispielhaft ein Ausschnitt für die Modellbeschreibung gezeigt:

```
(setq *lookup-class*
      (lookup-export cosim:_TC_Lookup_Class
                     "Fill" "class"
                     (cosim:Model_Class
                      :url "http://localhost/classes/fill.class"
                      :contexts   nil
                      :clocks     (list *lookup-control*)
                      :parameters (list *lookup-pressure*)
                      :runtimes   (list *lookup-import*
                                        *lookup-export*)))
```

Zur Laufzeit eines Modellexemplars erfolgen hauptsächlich Aufrufe der Methode `set_connection_value` durch den ORB, um Werte über Zugangspunkte zu

senden. Das Modellexemplar nimmt dann eine Fallunterscheidung nach dem angesprochenen Zugangspunkt vor (`control`, `pressure` oder `import`). In der einfach gehaltenen Umsetzung von Modellexemplaren wird die interaktive Laufsteuerung über den Zugangspunkt `control` gar nicht behandelt, da die Berechnung der Fülldauer umgehend in einem Schritt an die Beladekomponente weitergegeben wird. Werte, die über den Zugangspunkt `pressure` eingehen, geben eine Parametrisierung der Abfüllstation an und werden in einer Variablen gespeichert. Der Zugangspunkt `import` nimmt die eingehenden Leerfässer entgegen, berechnet die Fülldauer wie im vorigen Abschnitt angegeben und teilt sie allen Empfängern (also der Beladekomponente) direkt mit.

Der Wechsel der Nutzungsphasen macht auch in der Abfüllkomponente eine Synchronisation notwendig, damit beispielsweise nicht noch nach Ende der Laufzeitphase ausstehende Befüllungen abgearbeitet werden. Die LispWorks-Laufzeitumgebung stellt hierzu im Paket *Multiprocessing* geeignete `mp:lock` Strukturen zur Verfügung. Die Abfüllkomponente erzeugt dann für jedes eingehende Fass einen neuen Thread, der sich durch diesen Sperrmechanismus mit anderen nebenläufigen Aufrufen durch den ORB synchronisiert.

9.4 Modellkomponente „Hub“

Die Beladungskomponente „Hub“ verbindet eine Transportkomponente mit mehreren Exemplaren von Abfüllkomponenten. Dabei ist die Anzahl der verbundenen Abfüllkomponenten nicht festgelegt, sondern wird erst durch das Kopplungsdiagramm angegeben. Dies ist nur möglich, weil die Beladungskomponente als interne Komponente im Kopplungsdiagramm auftaucht. Interne Komponenten müssen ihre Modellbeschreibung nämlich erst offen legen, wenn aus dem Grapheneditor heraus versucht wird, eine Modellkopplung zu generieren. Vorher können die internen Komponenten ihre Umgebung in der Kopplungsspezifikation untersuchen und ihre Modellbeschreibung entsprechend anpassen.

9.4.1 Zugangspunkte im Beladungsteil

Als Vermittler zwischen Transport und Befüllung weist die Beladungskomponente die Gegenstücke der jeweiligen Laufzeitvariablen ihrer Kopplungspartner auf. So gibt es auf der Transportseite die Zugangspunkte `Order` und `Return` zum Entgegennehmen von Fahrzeugen mit Befüllungsaufträgen bzw. mit Leergut sowie den Zugangspunkt `Delivery` zum Abliefern beladener Fahrzeuge. Die Abfüllseite wird mit der entsprechenden Anzahl an `Import-` und `Export`-Zugängen bedient, über die leere bzw. gefüllte Fässer ausgetauscht werden.

Der Parameter `Strategy.Barrel` gibt die Strategie an, nach der die Befüllungsreihenfolge leerer Fässer ermittelt wird. Die Strategie, nach der die wartenden Fahrzeuge zu beladen sind, enthält der Parameter `Strategy.Vehicle`. Weiterhin kann die Menge vorhandener Fässer zum Simulationsstart über den Parameter `Initial` eingestellt werden. Der Parameter `Consumers` dient dazu, den größten gemeinsamen Teiler aller Skalierungseinheiten für Kundenlieferungen zu ermitteln. Der Teiler wird eingesetzt, um nicht jedes Fass einzeln nachbilden zu müssen. So kommt der Beladungsteil ohne Genauigkeitsverlust mit weniger Simulationsschritten aus.

Name	Phase	Richtung	Kopplung	Aktivität	Datentyp
`Initial`	Parameter	Eingabe	eng	+	`Lot[]`
`Strategy.Barrel`	Parameter	Eingabe	eng	+	`Barrel_Strategy`
`Strategy.Vehicle`	Parameter	Eingabe	eng	+	`Vehicle_Strategy`
`Consumers`	Parameter	Eingabe	eng	+	`Consumer[]`
`Transport.Order`	Laufzeit	Eingabe	eng	+	`Vehicle`
`Transport.Delivery`	Laufzeit	Ausgabe	eng	+	`Vehicle`
`Transport.Return`	Laufzeit	Eingabe	eng	+	`Vehicle`
`Fill.`*n*`.Import`	Laufzeit	Ausgabe	eng	+	`Barrel`
`Fill.`*n*`.Export`	Laufzeit	Eingabe	eng	+	`Barrel`
`Filling`	Ergebnis	Ausgabe	lose	+	`Queue_Statistics[]`
`Waiting`	Ergebnis	Ausgabe	lose	+	`Queue_Statistics[]`

Tabelle 9.3: Zugangspunkte der Beladungskomponente

Nach Simulationsende stehen zwei Warteschlangenstatistiken zur Verfügung. Der Zugangspunkt `Filling` liefert die Statistik über die Abfülldauer der Fässer, und über den Zugangspunkt `Waiting` sind die Wartezeiten der Fahrzeuge bis zum Beginn der Beladung erreichbar.

9.4.2 Umsetzung als interne Komponente

Jede interne Komponente, die als Berechnungsknoten in einem Kopplungsgraphen auftaucht, muss zunächst eine Beschreibung der möglichen Zugangspunkte angeben. Dazu zieht die Beladungskomponente die Namen der verbundenen Knoten heran und wertet sie gemäß Tabelle 9.3 aus. Die notwendige Typinformation muss durch Angabe von Objektreferenzen für Zugangspunkte im Initialisierungsfeld des Knotendialogs angegeben sein (vgl. Abbildung 7.24). Die Beladungskomponente bedient alle Datenpuffer aktiv. Anfragen für passive Partner per `calculate_value` müssen deshalb nicht realisiert sein.

Besondere Bedeutung kommt der Synchronisation im Beladungsteil zu, denn dieser Teil verwaltet selbst mehrere Threads, um Eingänge in den angeschlossenen Datenpuffern `Order`, `Return`, `Import.`*n* aktiv zu bearbeiten. Regelmäßig berechnen diese Threads eine interne Simulationszeit als Minimum aus Lookaheads und den Zeitstempeln unbearbeiteter Eingänge. Diese Zeit wird als Lookahead an alle Ausgänge der Laufzeitphase weitergereicht. Sie benutzen dabei ein zentrales Synchronisationsobjekt. Weitere Synchronisationsobjekte existieren für den Zugriff auf eine Bestellliste und auf eine Liste von Leerfässern.

Liegt eine Bestellung im Datenpuffer `Order` vor, so sorgt der zugehörige Thread dafür, dass die Bestellung in einer Liste gespeichert und ein Statistikvermerk für ein wartendes Fahrzeug angelegt wird. Leergut-Rückgaben im Datenpuffer `Return` führen zur Speicherung der Leerfässer in einer Warteschlange. Den Inhalt dieser Warteschlange arbeiten die Threads `Fill.`*n* ab, von denen einer je Abfüllstation existiert. Threads, die aktuell keine Fässer befüllen, werden im Round-Robin-Verfahren bedient. Sie entnehmen ein Fass der Warteschlange, senden es über den Datenpuffer `Fill.`*n*`.Export` an die Füllstation und warten auf die Rückkehr des Fasses im Datenpuffer `Fill.`*n*`.Import`. Da die Abfüllstati-

on Zeitstempel in der Einheit Sekunden interpretiert, die Beladungskomponente jedoch in Tagen rechnet, übernehmen diese Threads die nötigen Umrechnungen. Ein weiterer Thread `Delivery` prüft, ob von einer Liste vorliegender Bestellungen irgendein Auftrag bearbeitet und ein Fahrzeug abgefertigt werden kann.

Die Beladungskomponente erhält zwei Parameterwerte, die die Belade- und Befüllstrategie regeln. In beiden Fällen existiert jeweils eine abstrakte Klasse, die einen Vergleichsoperator deklariert. Dieser Operator vergleicht zwei Leerfässer bzw. zwei Ladelisten und entscheidet, welches Objekt Priorität bei der Bearbeitung erhält. Ein weiterer Operator gibt an, ob diese Entscheidung schon bei der Bestellung oder erst bei einer möglichen Auslieferung bzw. Befüllung getroffen werden kann. In beiden abstrakten Klassen stehen dann Klassenvariablen bereit, die die Operatoren realisieren und jeden der Punkte aus dem entsprechenden CORBA Aufzählungstyp abdecken. Jeder übergebene Parameterwert für eine Strategie führt zur Auswahl der zugehörigen Klassenvariablen, deren Wert in einer Objektvariablen des Modellexemplars als Strategieobjekt festgehalten wird. Mit dem Vergleichsoperator jedes Strategieobjektes lassen sich dann aus einer Liste von Fässern bzw. Ladelisten die zuerst zu bearbeitenden Objekte feststellen.

9.5 Das Kopplungsdiagramm

Die Abbildung 9.2 stellt das gekoppelte Modell aus dem Editor des Werkzeugs CoCo dar. Zentraler Bestandteil ist die mit `Hub` bezeichnete Beladekomponente. Das zugehörige sechseckige Symbol zeigt eine intern definierte Komponente an. Rechts davon befinden sich drei Abfüllstationen (`Fill.`*`n`*), links davon das Transportsystem (`Transport`). Diese externen Teilkomponenten sind als abgerundete Symbole dargestellt.

Der Benutzerzugang zum gekoppelten Modell erfolgt über die hell unterlegten rechteckigen Symbole. Dunkel unterlegte rechteckige Symbole spiegeln die Zugangspunkte der verwendeten inneren Modellkomponenten wider. Balken oberhalb einer Symbolbeschriftung stehen für Parameter, Balken unterhalb für Ergebnisgrößen. Zugangspunkte mit seitlichem Balken sind während eines Simulationslaufs erhältlich. Gepunktete seitliche Markierungen deuten die Zugangspunkte für die Simulationsuhr einer Modellkomponente an.

Zwischen der Uhrzeitsteuerung durch den Benutzer und einer möglichen Steuerung für die Abfüllstationen müssen noch die Zeiteinheiten von Tagen in Sekunden durch den Berechnungsknoten unten rechts umgerechnet werden.

Die Pfeilmarkierungen stellen den Datenfluss dar, wobei eine Linie mit zwei Pfeilen markiert sein kann. Je ein Pfeil steht für das Anfangs- bzw. Ende-Symbol der zugehörigen Verbindung. Helle Markierungen repräsentieren Aktivität bei der Datenübertragung, dunkle stehen für Passivität.

9.6 Experimentergebnisse

Das Beispielmodell ist einem Originalmodell mit Realitätsbezug nachempfunden, bleibt jedoch in der vorliegenden Form fiktiv. Trotzdem wurde versucht, bei der Parametrisierung auf reale Randbedingungen einzugehen. So befüllt eine moderne Maschine etwa 300 Bierfässer pro 8 Stundenschicht mit je 50 Liter In-

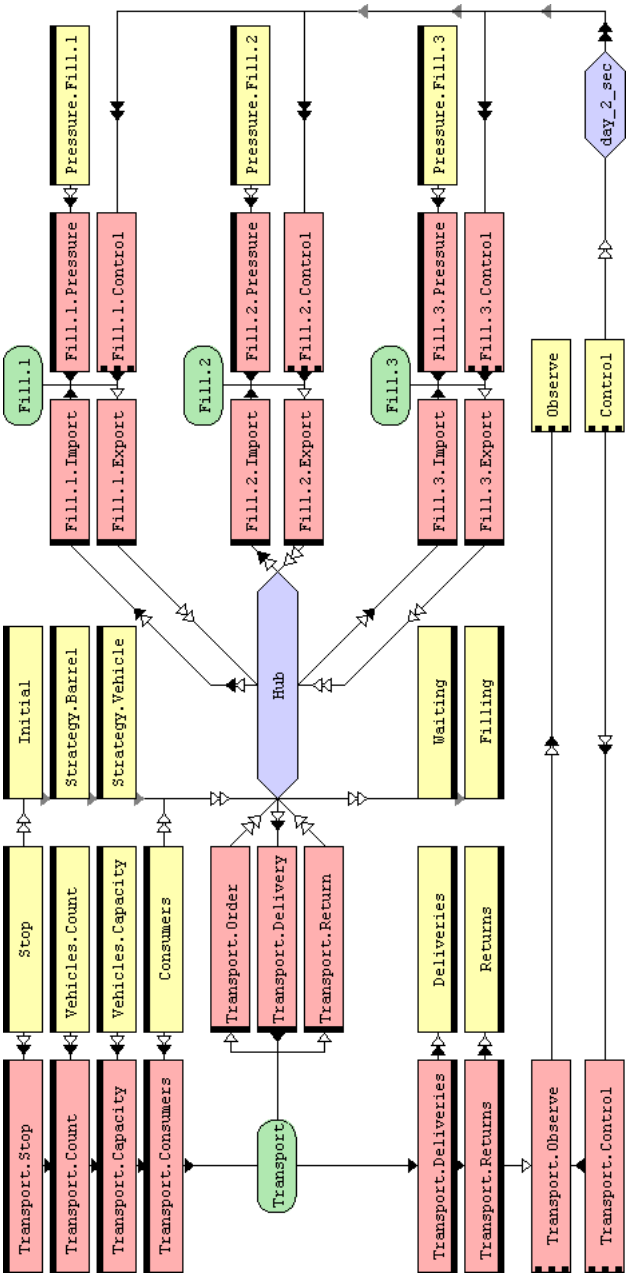


Abbildung 9.2: Kopplungsdiagramm

halt. Da das Abfüllmodell Reinigungs- und Wartungsarbeiten nicht betrachtet, legt man pauschal eine von drei Schichten als Ausfallzeiten fest. Damit ergibt sich eine Abfüllleistung von 600 Fässern pro Tag, was einem durchschnittlichen Fülldruck von 0,25 Litern pro Sekunde entspricht. Bei einem Minimaldruck von 0,05 Litern pro Sekunde erhält man so einen Maximaldruck von etwa 0,8333 Litern pro Sekunde. Die drei Abfüllstationen leisten zusammen eine Füllmenge von 1800 Fässern pro Tag.

Angenommen, ein Zwischenhändler beliefert etwa 300 Endkunden, und jeder Endkunde verbraucht im Mittel ein Fass pro Tag, dann ist bei tageweiser Bestellung eine Bestelleinheit von 100 Fässern gegenüber dem Abfüllbetrieb angebracht. Die Bestellungen der Zwischenhändler variieren zwischen 150 und 600 Fässern pro Tag. Dies entspricht einer Zeitspanne von 0,666 bzw. 0,166 Tagen zwischen zwei Bestelleinheiten. Der Abfüllbetrieb kann dann 6 Zwischenhändler mit dieser durchschnittlichen Bestellmenge bedienen.

Weiterhin wird angenommen, die Zwischenhändler lägen unterschiedlich weit entfernt vom Abfüllbetrieb mit einer variierenden LKW-Entfernung zwischen einer und 11 Stunden einfacher Fahrtzeit. Die mittlere Entfernung soll 6 Stunden betragen. Direktlieferungen an die Zwischenhändler ohne spezielle Routenplanung und ohne Rundtouren führen dann zu durchschnittlich zwei Lieferungen pro Tag und LKW, wenn man Ausfälle und Reparaturen der Fahrzeuge vernachlässigt. Eine ausreichende Anzahl an Fahrern sorgt dafür, dass die LKWs rund um die Uhr im Einsatz sind.

Das Ladevolumen eines großen LKW liegt bei etwa 20.000 Litern, es passen dann 400 Fässer mit je 50 Litern auf einen LKW. Für ein Transportvolumen von 1800 Fässern täglich genügen 3 eingesetzte LKW. Es ist günstig, die Lagergrößen und Bestellmengen so zu wählen, dass voll beladene LKW-Ladungen die Lager erreichen. Also sollten die Bestellmengen im Idealfall ein Vielfaches von 400 sein.

Die Lager der Zwischenhändler sind zum Simulationsstart maximal gefüllt. Der Abfüllbetrieb beginnt mit 1200 vollen Fässern. Die Anzahl der Fässer im Abfüllbetrieb ist so gewählt, dass zunächst kein Fahrzeug auf das Befüllen von Leergut warten muss.

Innerhalb eines Simulationsexperiments wurde das gekoppelte Beispielmodell mit verschiedenen Parametersätzen simuliert. Neben dem eben beschriebenen Grundszenario kamen vier Abwandlungen zum Einsatz:

- Halbierung der Zwischenwerte der Konsumzeiten
- Halbierung der Befüllungsdrücke
- Halbierung der gefüllten Fässer zum Simulationsstart
- Halbierung der Anzahl verfügbarer Fahrzeuge

Wie zu erwarten war, zeigten sich entsprechende Engpässe in jedem der abgewandelten Szenarien im Vergleich zur Grundeinstellung (vgl. Anhang E).

1. Halbiert man die Zwischenwerte der Konsumzeiten, so fordern die Zwischenhändler mehr Lieferungen an. Die Anzahl der beladenen Fahrzeuge (`waiting`) entspricht im Wesentlichen den Lieferungen. Der Anteil von Fahrzeugen, der ohne Wartezeit beladen werden konnte, bleibt jedoch gleich.

2. Bei halbem Befüllungsdruck sinkt zwar die Anzahl der befüllten Fässer auf etwa die Hälfte, die Anzahl der beladenen Fahrzeuge bleibt jedoch nahezu

unverändert. Es stehen also noch genügend Fässer aus der Anfangsphase zur Verfügung, damit der Disponent den Lieferanfragen nachkommen kann. Ein Engpass tritt erst auf, wenn die Fässer aus der Anfangsphase verbraucht sind.

3. Stehen zum Simulationsstart weniger gefüllte Fässer bereit, so steigt die Wartezeit der Fahrzeuge auf ihre Beladung deutlich an. Der anfängliche Fässervorrat wird dabei aufgebraucht, und die Fahrzeuge müssen auf den eigentlichen Befüllungsvorgang von Fässern warten.

4. Ist weniger Transportkapazität vorhanden, so sinkt die Menge an Lieferungen entsprechend. Die Fahrzeuge sind dabei jeweils voll ausgelastet.

10 Zusammenfassung und Ausblick

10.1 Zusammenfassende Betrachtung der Arbeit

Ziel dieser Arbeit war es, eine technische Basis zu schaffen, die es erlaubt, die Funktionalität vorhandener Simulationsmodelle als Komponenten in einer heterogenen Umgebung von Simulatoren bereitzustellen. Gelingt dieses Vorhaben, so ergeben sich für die Anwendung von Modellbildung und Simulation verschiedene Vorteile (Abschnitt 5.1). Da eine Modellkomponente das Wissen um die zu Grunde liegenden Systemzusammenhänge kapselt, lassen sich komplexe Modelle einfacher beherrschen. Simulationskomponenten ersparen Aufwand und damit verbundene Kosten, wenn sie für verschiedene Simulationsumgebungen zur Verfügung stehen und nur einmal entwickelt werden müssen. Zudem gehen viele Autoren davon aus, dass sich auf einem Markt für Simulationskomponenten nur qualitativ hochwertige, valide Modellteile durchsetzen. Das Vertrauen der Simulationsanwender in die erzielten Ergebnisse kann so gesteigert werden.

Damit ein komponentenorientierter Ansatz für die Simulation auch tatsächlich heterogene Simulationsumgebungen bedienen kann, müssen die in Abschnitt 5.2 herausgearbeiteten Anforderungen erfüllt sein. Einen zentralen Punkt stellt dabei die Unabhängigkeit von den Werkzeugen eines bestimmten Herstellers dar, denn nur so bleiben auch Anbindungen für wenig vertretene Umgebungen möglich. Manche Simulatoren lassen sich nur mit jeweils festgelegten Programmiersprachen anbinden. Ein komponentenorientierter Ansatz muss also sprachunabhängig sein, womit automatisch das Blackbox-Prinzip bei der Wiederverwendung zum Tragen kommt. Um gemeinsam Simulatoren zu nutzen, die auf verschiedenen Rechnern installiert sind, ist eine Kommunikation per Netzwerk vorzusehen. Die damit verbundene Nebenläufigkeit gekoppelter Simulationskomponenten schränken dann Mechanismen zur Synchronisation wieder ein, um die Kausalität innerhalb eines Gesamtmodells zu gewährleisten. Modell- oder domänenspezifische Datentypen erlauben eine problemangepasste Modellierung. Die Verwendung dieser Datentypen ist dann auch durchgängig von allen eingesetzten Werkzeugen zu unterstützen. Eine vollständige Unterstützung des dreischrittigen Experimentierprozesses (parametrisieren, berechnen, auswerten) kann ein Werkzeug gewährleisten, wenn die Komponentenbeschreibung entsprechende Benutzungsphasen vorsieht.

Bestehende Ansätze erfüllen die Anforderungen noch nicht (vgl. Abschnitt 5.3). Sogar die aktuelle Simulationsinfrastruktur HLA erreicht das Ziel der Wiederverwendung von Simulationskomponenten nicht, weil in den Bereichen Synchronisation, Introspektion und Aufbau von Simulationsverbünden eine deutliche Abhängigkeit vom jeweiligen Hersteller der notwendigen Laufzeitumgebung besteht. Der etablierte DEVS-Ansatz liefert mit dem Port-Konzept wichtige Grundlagen für die Kopplung von Simulationskomponenten und zur feingranularen Synchronisation. Er beschränkt sich jedoch auf wenige Grunddatentypen.

Der in dieser Arbeit konzipierte CoSim-Ansatz (Kapitel 6) geht auf alle genannten Anforderungen ein und bietet geeignete Lösungsmöglichkeiten an. Durch die Auswahl der Kommunikationsinfrastruktur CORBA erhält man direkt eine Lösung für den verteilten Zugang zu Objekten, der die Unabhängigkeit von Programmiersprachen und von den Werkzeugen eines bestimmten Herstellers garantiert. Das Blackbox-Prinzip wird mit Hilfe von Schnittstellenbeschreibungen in der Beschreibungssprache IDL umgesetzt, die auch modellspezifische Datentypen zulässt. Eine neue Art von Modellbeschreibungen erfasst die Zugangspunkte einer Komponente und ordnet sie in entsprechende Kategorien hinsichtlich ihrer Nutzungsphasen ein. Bei der Datenübertragung ist die Verwendung von Kontextwerten erlaubt, so dass die Simulationszeit als spezieller Kontext für die Synchronisation eingesetzt werden kann. Weiterhin sieht CoSim verschiedene Kommunikationsmechanismen vor. Dadurch können sich die Zugangspunkte bei der Datenübertragung hinsichtlich Aktivität und Kopplungsart unterscheiden.

Um zu zeigen, dass der CoSim-Ansatz auch umsetzbar ist, wurden verschiedene Werkzeuge entworfen und realisiert (Kapitel 7 und 8). Sie decken einen großen Teil der Aufgaben verschiedener Benutzerrollen ab, die in die komponentenorientierte Modellbildung und Simulation verwickelt sind. Dazu gehören Werkzeuge für die Komponentenerstellung (Werkzeug Cage), für die Komponentenkopplung (CoCo) und für die Experimentdurchführung (ExpU). Verzeichnisse nehmen die vorhandenen Modellkomponenten und ihre Beschreibungen auf. Sie werden ebenfalls durch eine komfortable grafische Benutzungsschnittstelle unterstützt. Sämtliche Werkzeuge wurden so realisiert, dass sie mit beliebigen modellspezifischen Datentypen umgehen können, auch wenn diese Typen zum Zeitpunkt der Kompilierung noch gar nicht existierten.

Anhand eines Beispielmodells zur Befüllung von Fässern wurde gezeigt, wie die einzelnen Werkzeuge zusammenspielen, um ein komplexes Gesamtmodell zu realisieren und damit Simulationsexperimente durchzuführen. Dabei kamen auch modellspezifische Datentypen und verschiedene Programmiersprachen zum Einsatz.

10.2 Offene Punkte und Ausblick

Die systematische Aufbereitung einzelner Anwendungsgebiete der Simulation durch Referenzmodelle (vgl. Abschnitt 5.2.8) liegt außerhalb der Zielsetzung dieser Arbeit. Diese Aufgabe verbleibt einerseits den Simulationsanwendern, die die charakteristischen Objekte ihrer Anwendungsdomäne und das zugehörige Verhalten bzw. deren Eigenschaften beschreiben. Auf der anderen Seite lohnt es sich für die Simulationsdienstleister, diese Beschreibungen in geeignete Daten- und Programmstrukturen umzusetzen, die dann zu einer umfangreichen Bibliothek wiederverwendbarer Komponenten beitragen können. Der CoSim-Ansatz dient in diesem Zusammenhang nur als mögliches technisches Rahmenwerk.

Auch im Bereich der Modellklassifikation und Komponentensuche sind noch keine etablierten Strukturen vorhanden. Ein Grund dafür ist sicherlich, dass bisher keine gemeinsame technische Basis existiert, die von vielen Benutzern eingesetzt wird. HLA trat mit dem Anspruch an, im militärischen Bereich als ausschließliche Lösung und im industriellen bzw. behördlichen Bereich als vorrangige Lösung für Simulationskomponenten zu dienen. Dieses Ziel hat HLA

bisher nicht erreicht, da noch immer Herstellerabhängigkeiten vorhanden sind und Modellentwickler die Schnittstellen wegen ihrer Komplexität nicht akzeptieren. Eine geeignete technische Basis jedoch, auf der eine kritische Masse von Simulationsanwendungen realisiert wird, kann dazu führen, dass auch Verzeichnisdienste mit komfortableren Suchmechanismen entstehen und intensiv genutzt werden.

Damit der CoSim-Ansatz diese Nutzung erfährt, müssen zunächst auch kommerziell verfügbare Simulatoren in die Umgebung eingebunden werden. Am Beispiel HLA hat Straßburger (2001) für die Simulatoren SLX und Simplex3 bereits gezeigt, dass eine Einbettung in übergreifende Umgebungen prinzipiell möglich ist, aber auch zusätzlicher Aufwand durch die Modellersteller entstehen kann. Hier sind einerseits die Hersteller der Simulatoren gefragt, ihre Werkzeuge hinsichtlich des Zugangs aus externen Applikationen heraus zu verbessern. Andererseits müssen Simulationsdienstleister diese Erweiterungen auch einfordern, um ihre Komponenten in einer offenen Umgebung anbieten zu können. Für die Einbettung weiterer Simulatoren wird es hilfreich sein, einen Adapter-Generator wie Cage in anderen Programmiersprachen als Java umzusetzen.

Die Grundstrukturen der Ansätze HLA, DEVS/CORBA und CoSim ähneln sich stark. Modellbeschreibungen, Zugangspunkte und spezielle Synchronisationsnachrichten stellen wesentliche Gemeinsamkeiten dar. So liegt der Versuch nahe, zwischen diesen Ansätzen Brücken zu schaffen, um die technischen Hürden zu überwinden. Mit DEVS/HLA (Zeigler u. a. 1999a) entstand bereits ein viel versprechender Übergang. Es bleibt also noch zu untersuchen, inwieweit sich Übergänge von CoSim zu HLA bzw. zu DEVS realisieren lassen und welche Probleme dabei auftreten können. Für alle Ansätze stehen bisher noch Umsetzungen aus, die den sicheren Zugang zu Modellkomponenten ermöglichen und vor unbefugter Benutzung oder Fälschungen schützen. Ebenso wichtig wird auf einem Markt für Komponenten die Abrechnung ihrer Benutzung sein. Hier könnten die CORBA-Dienste *Licensing* und *Security* Unterstützung leisten.

Weiterhin sind für die bestehenden Werkzeuge der CoSim-Umgebung Erweiterungen sinnvoll.

- Die Experimentierumgebung enthält beispielsweise nur einfache Visualisierungswerkzeuge. Hier bietet sich eine Exportfunktion in übliche Dateiformate an, damit Ergebnisdaten nach einem Simulationslauf durch externe Werkzeuge für Präsentationsgrafiken aufbereitet werden können. Laufbegleitende Darstellungen verlangen meist modellspezifische Anpassungen, etwa um in einem Transportmodell möglichen Routen der transportierten Objekte festzulegen. Man könnte deshalb für bestimmte Anwendungsdomänen geeignete Baukästen bereitstellen, mit denen sich komplexe Visualisierungen zusammenstellen lassen.

- Um die Palette vorhandener Datenfilter und Statistikfunktionen zu vergrößern, sieht die Experimentierumgebung bereits Schnittstellen vor. Bisher sind jedoch nur die einfachen Funktionen realisiert.

- Erweiterungen für die Kontrollstrukturen zum Start von Simulationsläufen wurden bereits kurz in Abschnitt 7.4.2 erwähnt.

- Die Werkzeuge Cage und CoCo werten bei Methodenaufrufen zur Datenübertragung mit enger Kopplung regelmäßig die Beschreibungen von

Zugangspunkten neu aus und erzeugen damit zusätzliche Aufrufe auf entfernten Objekten, da die Zugangspunkte durch den Objekthüllendienst verwaltet werden. Ein Caching-Mechanismus für die Zugangspunkte kann hier zu einer wesentlich schnelleren Abarbeitung dieser Anfragen führen.

- CoCo erlaubt noch keine direkte Datenübertragung zwischen Modellkomponenten. Sämtliche Datenwerte laufen zunächst über das gekoppelte Modell zur Laufzeitumgebung und werden dann zum Empfänger weitergeleitet. Bei der losen Kopplung ließen sich die beteiligten Nachrichtenkanäle unmittelbar verbinden. Für die enge Kopplung kann CoCo den passiven Kommunikationspartner beim aktiven Partner als Sender bzw. Empfänger anmelden.

- In Kopplungsdiagrammen müssen die Knotennamen bisher eindeutig sein. Bei großen Diagrammen kann dies dazu führen, dass Linienverbindungen über weite Strecken zu führen sind, um räumlich entfernte Knoten zu verbinden. Diese Randbedingung lässt sich auflösen, indem man Darstellungssymbole mit gleichem Namen als identische Knoten behandelt.

Abschließend kann man festhalten, dass der CoSim-Ansatz einen kleinen aber wichtigen Teil zur Förderung wiederverwendbarer Simulationskomponenten ausmacht. Damit diese Simulationskomponenten zukünftig effizient eingesetzt werden können, müssen Simulationsanwender, Dienstleister und Werkzeughersteller an einem Strang ziehen.

Anhang

A Schnittstellendefinitionen

Dieser Anhang führt die kommentierten IDL-Definitionen auf, über die Simulationskomponenten und ihre Beschreibungen zugänglich sind. Die Definitionen verteilen sich auf verschiedene Dateien. Tabelle A.1 gibt einen kurzen Überblick der Inhalte.

Datei	**Inhalt**
`cosim_basics.idl`	Nutzungsphasen, Modellabhängigkeiten, Strukturen zur Datenübertragung
`cosim_class.idl`	Modellbeschreibungen (Richtung, Kopplung, Aktivität), Kategorien
`cosim_model.idl`	Modellkomponente
`cosim_factory.idl`	Modellfabrik
`cosim_lookup.idl`	Objekthüllen
`cosim_datatransfer.idl`	Rückrufschnittstellen für enge Kopplungen
`aliases.idl`	Typinformationen für Objekthüllen
`cage.idl`	Scanner (CoSim Adapter Generator)
`coco.idl`	Scanner (CoSim Component Connector)
`desmoj.idl`	Statistik aus Desmo-J
`barrels.idl`	Beispielmodell

Tabelle A.1: Übersicht der IDL-Definitionen

```
#ifndef _ORB_IDL_
#include "orb.idl"
#define _ORB_IDL_
#endif
#ifndef COSNAMING
#include "CosNaming.idl"
#define COSNAMING
#endif
#ifndef COSEVENTCHANNELADMIN
#include "CosEventChannelAdmin.idl"
#define COSEVENTCHANNELADMIN
#endif
#ifndef COSTYPEDNOTIFYCHANNELADMIN
#include "CosTypedNotifyChannelAdmin.idl"
#define COSTYPEDNOTIFYCHANNELADMIN
#endif
#define COSIM_BASICS
#pragma prefix "bachmann.informatik.unihh.de"

// Definitionen für die Architektur CoSim (Component Based Distributed Simulation)
module cosim {

  interface Model;
  interface Model_Factory;
  interface Lookup_Repository;
  interface Lookup_Object;
  interface Lookup_Class;
  interface Lookup_Connection;
  interface Lookup_Value;

  // ================================================================================================
  // Eine Liste aus Lookup_Connection Objekten.
  typedef sequence<Lookup_Connection> Lookup_Connections;

  // ................................................................................................
  // Eine Liste aus Lookup_Class Objekten.
  typedef sequence<Lookup_Class> Lookup_Classes;

  // ================================================================================================
  // Beschreibt, welche semantischen Abhängigkeiten zwischen Modellvariablen bestehen.
  // Diese Abhängigkeiten geben das Ein-/Ausgabeverhalten eines Modells an,
  // d.h. welche Werte von Eingabevariablen benötigt werden,
  // um den Wert einer bestimmten Ausgabevariable zu ermitteln.
  struct Dependency {
    Lookup_Connection  connection; // Die Ausgabevariable, die von anderen Variablen abhängt
    Lookup_Connections uses;       // Die Eingabevariablen, die von der Ausgabevariablen abhängen
  };

  // Eine Liste aus Abhängigkeiten zwischen Modellvariablen.
  typedef sequence <Dependency> Dependencies;

  // ================================================================================================
  // Faßt eine Modellvariable und ihren aktuellen Wert zusammen.
  // Dies kann als Schlüssel/Wert-Paar angesehen werden,
  // wobei die Objektrefenz zur Modellvariablen als Schlüssel dient.
  // Paare dieser Art werden verwendet,
  // um bei der Werteübermittlung einen Kontext anzugeben
  // (also weitere Hilfsvariablen und deren Werte).
  // Kontextwerte sind notwendig, um weitere, beschreibende Daten
  // im Zuge einer einzigen Anfrage weiterzuleiten.
  // Häufig ist die aktuelle Simulationszeit beim Auslesen von Modellvariablen von Interesse.
  struct Context_Value {
    any               current; // Ein beliebiger (Kontext-)Wert.
                               // Der Datentyp dieses Wertes muß mit dem Datentyp
                               // aus der angegebenen Lookup_Connection übereinstimmen.
    Lookup_Connection guide;   // Variable, von der der aktuelle (Kontext-)Wert stammt
```

```
  };

  // Eine Liste aus Context_Values.
  typedef sequence <Context_Value> Context_Values;

  // ================================================================================================
  // Faßt den Wert einer Modellvariablen und ihre Kontextwerte zusammen.
  struct Guided_Value {
    any            simple;   // Der aktuelle Wert einer Modellvariablen
    Context_Values contexts; // Eine Menge von weiteren, beschreibenden Daten zum aktuellen Wert
    boolean        valid;    // Markierung für die Gültigkeit des Variablenwertes
    // Ist die Markierung als FALSE gesetzt,
    // so ist diese Nachricht als Zeitstempel zu behandeln und liefert keine Nutzdaten.
    // Stattdessen dient die Nachricht zur Synchronisation zwischen Modellkomponenten.
    // Sie stellt eine sogenannte Null-Nachricht dar.
    // Es sollte ein Kontextwert (Zeitstempel) für eine Uhrzeit-Variable angegeben sein,
    // der den kleinsten möglichen Zeitstempel der Modellvariablen angibt.
    // Alle nachfolgenden Werte für die Modellvariable werden nur solche Zeitstempel aufweisen,
    // deren Wert größer oder gleich dem vorliegenden Zeitstempel ist.
    // Markierte Nachrichten dieser Art versetzen eine Modellkomponente in die Lage,
    // die Nachrichten für eigene Modellvariablen soweit als möglich abzuarbeiten
    // und ihrerseits weitere markierte Nachrichten an andere Modellkomponente auszusenden,
    // damit die Simulationszeit eines gekoppelten Modells vorangetrieben wird.
  };

  // ================================================================================================
  // Nutzungsphase eines Modellexemplars.
  enum Model_State {CREATED, // Der Anfangszustand. Das Modell ist gerade erzeugt worden.
                    DELETED, // Der Endzustand.
                             // Das Modell wurde geschlossen und wird bald deaktiviert werden.
                             // Es wird nicht auf weitere Anfragen zur Zustandsänderung reagieren.
                             // Anfragen an Modellvariablen liefern keine sinnvollen Werte mehr.

                    SETTING, // Das Modell kann Werte zu Parametervariablen liefern oder empfangen.
                    RUNNING, // Das Modell kann Werte zu Laufzeitvariablen liefern oder empfangen.
                    GETTING, // Das Modell kann Werte zu Ergebnisvariablen liefern oder empfangen.

                    STOPPED  // Der Simulationslauf wurde angehalten.
                             // Das Modell kann keine Werte liefern oder empfangen.
  };
};
```

```
#define COSIM_CLASS
#ifndef COSIM_BASICS
#include "cosim_basics.idl"
#endif
#ifndef COSIM_DATATRANSFER
#include "cosim_datatransfer.idl"
#endif
#pragma prefix "bachmann.informatik.unihh.de"

module cosim {

  // ================================================================================================
  // Beschreibt, auf welche Art eine Modellvariable zugreifbar ist.
  // Eine Modellvariable wird durch die Datenstruktur Model_Connection beschrieben.
  // Das Feld "coupling" darin gibt die Kopplungsart an.
  enum Connection_Coupling {TIGHT, // Enge Kopplung.
                                   // Die Zugriffe auf die Variable erfolgen synchron.
                                   // Ist die Variable ein aktiver Sender, erfolgt ein Rückruf
                                   // über die Schnittstelle Guided_Value_Consumer.
                                   // Ist die Variable ein passiver Sender,
                                   // implementiert sie die Schnittstelle Guided_Value_Supplier.
                                   // Ist die Variable ein aktiver Empfänger, erfolgt ein Rückruf
                                   // über die Schnittstelle Guided_Value_Supplier.
                                   // Ist die Variable ein passiver Empfänger,
                                   // implementiert sie die Schnittstelle Guided_Value_Consumer.

                            LOOSE, // Lose Kopplung.
                                   // Die Zugriffe auf die Variable
                                   // erfolgen asynchron mit Hilfe eines EventChannels.
                                   // Über einen EventChannel können mehrere Sender
                                   // und auch mehrere Empfänger angeschlossen sein,
                                   // die ihre gegenseitgie Identität nicht im voraus kennen.
                                   // Aktive Sender rufen "push" Anfragen auf.
                                   // Passive Sender antworten auf "pull" und "try_pull" Anfragen.
                                   // Aktive Empfänger rufen "pull" Anfragen auf,
                                   // wenn sie auf einen Wert warten wollen.
                                   // Aktive Empfänger rufen "try_pull" Anfragen auf,
                                   // wenn sie prüfen wollen, ob ein Wert vorliegt.
                                   // Passive Empfänger antworten auf "push" Anfragen.

                            NONE   // Keine Kopplung vorhanden.
                                   // Die entsprechende Variable wird ausschließlich
                                   // innerhalb von Kontextwerten verwendet
                                   // und kann nicht direkt abgefragt werden.
  };
  // ================================================================================================
  // Beschreibt, ob eine Modellvariable selbständig Datenübertragungen anstüßt.
  // Eine Modellvariable wird durch die Datenstruktur Model_Connection beschrieben.
  // Das Feld "activity" darin gibt ihre Aktivität an.
  enum Connection_Activity {ACTIVE,  // Das Modellexemplar erzeugt die entsprechenden Aufrufe,
                                     // um Datenwerte für die Variable zu verschicken oder abzurufen.
                            PASSIVE  // Benutzer dieser Variablen müssen Datenwerte
                                     // für die Variable abfragen bzw. setzen.
  };
  // ................................................................................................
  // Beschreibt die Rolle einer Modellvariablen bei der Datenübertragung.
  enum Connection_Direction {IMPORT, // Die Variable empfängt Datenwerte.
                             EXPORT  // Die Variable versendet Datenwerte.
  };
  // ................................................................................................
  // Gültigkeitsbereich von Variablenwerten.
  // Eine Modellvariable kann mehrere Gültigkeitsbereiche zugeordnet bekommen.
  // Jeder Gültigkeitsbereiche stellt ein Intervall dar.
  struct Interval {
    Lookup_Value lower; // Untere Grenze des Gültigkeitsbereichs.
    Lookup_Value upper; // Obere  Grenze des Gültigkeitsbereichs.
```

```
};

// ..............................................................................
// Eine Menge von Gültigkeitsbereichen einer Modellvariablen.
// Die einzelnen Gültigkeitsbereiche sind als ODER verknüpft zu betrachten.
typedef sequence<Interval> Intervals;

// ..............................................................................
// Beschreibt eine Modellvariable syntaktisch.
// Alle Modellexemplare, die von derselben Model_Factory erzeugt sind,
// besitzen die gleichen Modellvariablen im Hinblick auf
// Namen, Typen, Kommunikationsmuster, Anfangswerte und Gültigkeitsbereiche.
// Eine Model_Connection kann als Objektreferenz über eine Lookup_Connection gespeichert sein.
struct Model_Connection {
  ::CORBA::IDLType     type;      // Datentyp der Modellvariablen.
  Connection_Coupling  coupling;  // Die benutzte Kopplungsart, um Datenwerte zu übertragen.
  Connection_Activity  activity;  // Die Aktivität der Modellvariablen bei einer Datenübertragung.
  Connection_Direction direction; // Die Rolle     der Modellvariablen bei einer Datenübertragung.

  Lookup_Connections   requested; // Benötigte Parameter einer Werteanfrage.
                                  // Um den Wert einer Variablen zu ermitteln, kann es notwendig sein,
                                  // den Kontext der entsprechenden Anfragen auszuwerten,
                                  // d.h. die Anfrage ist parametrisiert.
                                  // Dieses Feld gibt alle Hilfsvariablen an,
                                  // für die ein Kontextwert je Anfrage bereitszustellen ist.
                                  // Jede Hilfsvariablen dient als Schlüssel
                                  // in einer Liste aus Context_Value Strukturen.
                                  // Dieses Anfragemuster tritt nur bei synchronen Ausgabevariablen auf,
                                  // für alle anderen Variablen
                                  // sollte hier eine leere Liste angegeben sein.

  Lookup_Connections   responded; // Gelieferte Kontextwerten der Modellvariablen.
                                  // Um den Wert einer Modellvariablen zu interpretieren,
                                  // kann die Angabe von Kontextwerten notwendig sein.
                                  // Dieses Feld gibt alle Hilfsvariablen an,
                                  // für die eine Anfrage Kontextwerte bereitstellen wird.
                                  // Jede Hilfsvariablen dient als Schlüssel
                                  // in einer Liste aus Context_Value Strukturen.

  Lookup_Value         initial;   // Anfangswert der Modellvariablen.
                                  // Sobald ein Modellexemplar in den Zustand übergeht,
                                  // der für diese Modellvariable maßgebend ist,
                                  // bekommt die Modellvariable diesen Anfangswert zugeordnet.
                                  // Falls kein Anfangswert existiert,
                                  // so liefert die Methode "get_data" der Lookup_Content Schnittstelle
                                  // ein Any Objekt, das als ::CORBA::TypeCode
                                  // den Wert ::CORBA::TCKind.tk_void enthält.

  Intervals            valid;     // Gültigkeitsbereiche einer Modellvariablen.
                                  // Sie sind nur sinnvoll für Zahl- oder Aufzählungstypen.
};
// ==============================================================================================
// Beschreibt die Menge der Modellvariablen,
// die für jedes Modellexemplar einer gegebenen Modellfabrik existieren.
// Eine Model_Class kann als Objektreferenz über eine Lookup_Class gespeichert sein.
struct Model_Class {
  string             url;        // Verweis auf ein MIME Dokument,
                                 // das weitere Erläuterungen zu den Modellvariablen enthält.

  Lookup_Connections clocks;     // Externe Repräsentationen der internen Simulationsuhr.
                                 // Obwohl es mehrere Repräsentationen
                                 // der internen Simulationsuhr geben kann,
                                 // um z.B. mit verschiedenen Kommunikationsmustern zu arbeiten,
                                 // so ist doch immer dieselbe interne Uhr im Hintergrund gemeint.
                                 // Diese Modellvariablen sind nur gültig,
                                 // wenn sich ein Modellexemplar im Modellzustand RUNNING befindet.
```

```
                                // Jeder Simulationslauf läßt sich
                                // in einzelne Simulationsschritte unterteilen.
                                // Jeder Simulationsschritt schaltet in der Regel
                                // eine interne Simulationsuhr weiter.
                                // Diese Simulationszeit kann über Modellvariablen
                                // extern sichtbar gemacht werden.
                                // Ist die Modellvariable ein Empfänger von Datenwerten,
                                // so wird jeder übermittelte Datenwert als Zielwert betrachtet.
                                // Die interne Simulationsuhr darf solange
                                // einzelne Simulationsschritte durchführen,
                                // bis der Zielwert erreicht wurde.
                                // Ist die Modellvariable als Sender von Datenwerten deklariert,
                                // so wird die nächste mögliche Simulationszeit
                                // zum Zeitpunkt einer Anfrage zurückgegeben.
                                // Die Simulationszeit muß mit einem Zahltyp in Verbindung stehen.

    Lookup_Connections contexts;   // Modellvariablen, die nur als Hilfsvariablen (Kontext) dienen.
                                // Diese Modellvariablen können nicht direkt
                                // abgefragt oder gesetzt werden.

    Lookup_Connections parameters; // Modellvariablen, die als Parameter dienen.
                                // Diese Modellvariablen sind nur gültig,
                                // wenn sich ein Modellexemplar im Modellzustand SETTING befindet.
                                // Es sind üblicherweise Eingabevariablen,
                                // die das synchrone Kommunikationsmuster verwenden,
                                // denn der Sender von Parametern benötigt in der Regel
                                // eine Bestätigung über den Empfang.
                                // Falls noch nicht alle notwendigen Parameter
                                // eines Modellexemplars gesetzt sind,
                                // ist der Start eines neuen Simulationslauf nicht möglich.

    Lookup_Connections runtimes;   // Modellvariablen, die als Laufzeitgrößen dienen.
                                // Diese Modellvariablen sind nur gültig,
                                // wenn sich ein Modellexemplar im Modellzustand RUNNING befindet.

    Lookup_Connections results;    // Modellvariablen, die als Ergebnisgrößen dienen.
                                // Diese Modellvariablen sind nur gültig,
                                // wenn sich ein Modellexemplar im Modellzustand GETTING befindet.
  };
};
```

```
#define COSIM_MODEL
#ifndef COSIM_CLASS
#include "cosim_class.idl"
#endif
#pragma prefix "bachmann.informatik.unihh.de"

module cosim {

  // ============================================================================================
  // Die Vorbereitung einer Modellnutzungsphase kann nicht durchgeführt werden.
  // Für einige Modellvariablen fehlt ein Datenwert.
  // Die Modellnutzungsphase wurde nicht geändert.
  exception State_Denied {
    Model_State        current;     // Die aktuelle Modellnutzungsphase.
    Model_State        requested;   // Die angefragte Modellnutzungsphase.
    Lookup_Connections connections; // Die fehlenden Modellvariablen.
  };
  // ............................................................................................
  // Der Vorgängerzustand einer Anfrage zur Änderung der Modellnutzungsphase war nicht korrekt.
  // Die Modellnutzungsphase wurde nicht geändert.
  exception State_Cannot {
    Model_State current;   // Die aktuelle Modellnutzungsphase.
    Model_State requested; // Die angefragte Modellnutzungsphase.
  };
  // ............................................................................................
  // Interne Problem bei der Änderung der Modellnutzungsphase.
  exception State_Broken {
    Model_State current;   // Die aktuelle Modellnutzungsphase.
    Model_State requested; // Die angefragte Modellnutzungsphase.
  };
  // ============================================================================================
  // Das Modellexemplar als Kommunikationsendpunkt für Modellvariablen.
  // Um verwendete Ressourcen innerhalb einer Modellfabrik zu schonen,
  // kann ein Modellexemplar nach einiger Zeit ohne Kommunikationsaktivitäten
  // geschlossen werden.
  interface Model :
    Guided_Value_Supplier,
    Guided_Value_Consumer {

    // Liefert die Modellfabrik, die dieses Modellexemplar erzeugt hat.
    // Da die Modellfabrik die Definition der Modellklasse enthält,
    // ist diese Methode zur Introspektion notwendig.
    Model_Factory get_factory ();

    // Nachrichtenkanal für Ereignisse, die die Ausführungssteuerung betreffen.
    // Es gibt genau einen typisierten Nachrichtenkanal
    // je Modellexemplar für alle entsprechenden Ereignisse.
    // Die Implementation eines Modellexemplars ist verantwortlich für das Senden der Ereignisse.
    // Ein Modellexemplar versendet Ereignisse der Art Model_Event.
    ::CosTypedNotifyChannelAdmin::TypedEventChannel get_administration_channel ();

    // Nachrichtenkanal zur Übertragung von Datenwerten einer bestimmten Modellvariablen.
    // Je Modellvariable, die zur Datenübertragung die lose Kopplung verwendet,
    // gibt es einen eigenen Nachrichtenkanal.
    // Alle Datenwerte der Modellvariablen werden über den entsprechenden Nachrichtenkanal übertragen.
    // Jeder übertragene Datenwert ist eine Guided_Value Strukur,
    // auch falls keine Kontextwerte involviert sind.
    ::CosNotifyChannelAdmin::EventChannel get_connection_channel
    (in Lookup_Connection con // Die Objekthülle der Beschreibung einer Modellvariablen.
     ) raises (Connection_Not_Owned, // Die Variable existiert in diesem Modellexemplar nicht.
               Connection_Not_Loose  // Die Variable benutzt keine asynchrone Datenübertragung.
               );
    // ..........................................................................................
    // Fügt dem Modellexemplar einen Empfänger für Datenwerte einer bestimmten Modellvariablen hinzu.
    // Eine Ausgabevariable, die aktiv über synchrone Kommunikationswege zugreifbar ist,
    // benutzt die Rückruf-Schnittstelle Guided_Value_Consumer zur Datenübertragung.
```

```
// Die Modellvariable verschickt neue Datenwerte immer an alle Rückruf-Objekte.
// Die Reihenfolge, in der die Rückruf-Objekte aufgerufen werden, ist nicht festgelegt.
void append_connection_consumer
(in Lookup_Connection     con,     // Die Objekthülle der Beschreibung einer Modellvariablen.
 in Guided_Value_Consumer consumer // Der Empfänger für Datenwerte der Modellvariablen.
 ) raises (Connection_Not_Owned,   // Die Variable existiert in diesem Modellexemplar nicht.
           Connection_Not_Tight,   // Die Variable benutzt keine synchrone Datenübertragung.
           Connection_Not_Active,  // Die Variable ist bei der Datenübertragung nicht aktiv.
           Connection_Not_Export   // Die Variable ist keine Ausgabevariable.
           );

// Fügt dem Modellexemplar einen Sender für Datenwerte einer bestimmten Modellvariablen hinzu.
// Eine Eingabevariable, die aktiv über synchrone Kommunikationswege zugreifbar ist,
// benutzt die Rückruf-Schnittstelle Guided_Value_Supplier zur Datenübertragung.
// Die Modellvariable versucht, neue Datenwerte immer von allen hinzugefügten Rückruf-Objekten zu erhalten.
// Die Reihenfolge, in der die Rückruf-Objekte aufgerufen werden, ist nicht festgelegt.
void append_connection_supplier
(in Lookup_Connection     con,     // Die Objekthülle der Beschreibung einer Modellvariablen.
 in Guided_Value_Supplier supplier // Der Sender für Datenwerte der Modellvariablen.
 ) raises (Connection_Not_Owned,   // Die Variable existiert in diesem Modellexemplar nicht.
           Connection_Not_Tight,   // Die Variable benutzt keine synchrone Datenübertragung.
           Connection_Not_Active,  // Die Variable ist bei der Datenübertragung nicht aktiv.
           Connection_Not_Import   // Die Variable ist keine Eingabevariable.
           );

// Entfernt aus dem Modellexemplar einen Empfänger für Datenwerte einer bestimmten Modellvariablen.
// Eine Ausgabevariable, die aktiv über synchrone Kommunikationswege zugreifbar ist,
// benutzt die Rückruf-Schnittstelle Guided_Value_Consumer zur Datenübertragung.
// Die Modellvariable verschickt neue Datenwerte nur an hinzugefügte Rückruf-Objekte.
void remove_connection_consumer
(in Lookup_Connection     con,     // Die Objekthülle der Beschreibung einer Modellvariablen.
 in Guided_Value_Consumer consumer // Der Empfänger für Datenwerte der Modellvariablen.
 ) raises (Connection_Not_Owned,   // Die Variable existiert in diesem Modellexemplar nicht.
           Connection_Not_Tight,   // Die Variable benutzt keine synchrone Datenübertragung.
           Connection_Not_Active,  // Die Variable ist bei der Datenübertragung nicht aktiv.
           Connection_Not_Export   // Die Variable ist keine Ausgabevariable.
           );

// Entfernt aus dem Modellexemplar einen Sender für Datenwerte einer bestimmten Modellvariablen.
// Eine Eingabevariable, die aktiv über synchrone Kommunikationswege zugreifbar ist,
// benutzt die Rückruf-Schnittstelle Guided_Value_Supplier zur Datenübertragung.
// Die Modellvariable versucht, neue Datenwerte nur von hinzugefügten Rückruf-Objekten zu erhalten.
void remove_connection_supplier
(in Lookup_Connection     con,     // Die Objekthülle der Beschreibung einer Modellvariablen.
 in Guided_Value_Supplier supplier // Der Sender für Datenwerte der Modellvariablen.
 ) raises (Connection_Not_Owned,   // Die Variable existiert in diesem Modellexemplar nicht.
           Connection_Not_Tight,   // Die Variable benutzt keine synchrone Datenübertragung.
           Connection_Not_Active,  // Die Variable ist bei der Datenübertragung nicht aktiv.
           Connection_Not_Import   // Die Variable ist keine Eingabevariable.
           );
// ...........................................................................................
// Liest oder schätzt den minimalen Wert der internen Simulationsuhr.
// Für alle in der Modellklasse angegebenen Repräsentationen der Simulationsuhr
// wird der entsprechende Wert ermittelt.
// Die Liste der Werte ist leer, falls überhaupt keine Simulationsuhr verwendet wird.
Context_Values get_clock_minimum (); // Die Liste aller möglichen Repräsentationen.

// Liest oder schätzt den maximalen Wert der internen Simulationsuhr.
// Für alle in der Modellklasse angegebenen Repräsentationen der Simulationsuhr
// wird der entsprechende Wert ermittelt.
// Die Liste der Werte ist leer, falls überhaupt keine Simulationsuhr verwendet wird.
Context_Values get_clock_maximum (); // Die Liste aller möglichen Repräsentationen.

// ...........................................................................................
// Ändert das Zeitintervall, nach dem das Modellexemplar wegen Inaktivität geschlossen werden darf.
void set_purge_interval (in long long millis // Zeitintervall in Millisekunden.
```

```
                          );
    // Liest das Zeitintervall ab, nach dem das Modellexemplar wegen Inaktivität geschlossen werden darf.
    long long get_purge_interval (); // Zeitintervall in Millisekunden.

    // Liest die Restzeit ab, die noch verbleibt, um das Modellexemplar wegen Inaktivität zu schliessen.
    // Diese Methode setzt die Restzeit bis zum Schließen des Modellexemplars NICHT zurück.
    // Alle anderen Methoden des Modellexemplars,
    // sowie Kommunikationsaktivitäten über einen Nachrichtenkanal des Modellexemplars,
    // erneuern diese Restzeit.
    long long get_purge_schedule (); // Die Restzeit in Millisekunden.

    // ..........................................................................................
    // Aktuelle Nutzungsphase.
    Model_State get_state (); // Die Nutzungsphase als Model_State.

    // Vollzieht den Übergang in eine neue Modellnutzungsphase.
    // Die Nachrichtenkanäle der aktuellen Modellnutzungsphase werden dabei geschlossen.
    // Im Anschluß enstehen neue Exemplare der Nachrichtenkanäle für die betroffenen Modellvariablen.
    // Sobald diese Anfrage ohne Fehlermeldungen zurückkehrt ,
    // ist das Modellexemplar in die neue Modellnutzungsphase übergegangen.
    void change_state
    (in Model_State target   // Die neue Modellnutzungsphase.
     ) raises (State_Cannot, // Die angegebene Modellnutzungsphase
                             // ist aus der aktuellen Modellnutzungsphase nicht erreichbar.
               State_Broken, // Der Übergang schlug fehl.
               State_Denied  // Es fehlen Datenwerte in der aktuellen Modellnutzungsphase,
                             // um in die angegebene Modellnutzungsphase überzugehen.
               );

    // Führt die Berechnungsschritte der Modellnutzungsphase RUNNING aus.
    // Diese Anfrage kann erst dann zum Erfolg führen,
    // wenn der Übergang in die Modellnutzungsphase RUNNING beendet ist.
    // Die entsprechende change_state Anfrage ist also bereits ohne Fehler zum Abschluß gekommen.
    void execute () raises (State_Cannot, // Die aktuelle Modellnutzungsphase ist nicht RUNNING
                                          // oder eine execute Anfrage ist bereits erfolgt.
                            State_Broken  // Die Ausführung schlug fehl.
                            );
  };
};
```

```
#define COSIM_FACTORY
#ifndef COSIM_LOOKUP
#include "cosim_lookup.idl"
#endif
#ifndef COSIM_MODEL
#include "cosim_model.idl"
#endif
#pragma prefix "bachmann.informatik.unihh.de"

module cosim {

  // ============================================================================================
  // Steuert den Lebenszyklus von Modellexemplaren.
  // Dies ist die Einstiegsschnittstelle für die Nutzung von Simulationsmodellen in CoSim.
  // Üblicherweise ist eine Model_Factory bei einem Suchdienst eingetragen,
  // wie z.B. CORBA NamingService, CORBA TradingService oder CoSim Factory_Repository.
  // Mit einem Modellexemplar kann nur ein Simulationslauf zur Zeit durchgeführt werden.
  // Um mehrere, nebenläufige Simulationsläufe durchzuführen,
  // können aus einer Modellfabrik mehrere Modellexemplare erzeugt und bedient werden.
  interface Model_Factory : Lookup_Object {

    // Ein Verweis auf ein MIME Dokument,
    // das weitere Erläuterungen zu den inneren Abläufen der erzeugten Modellexemplare enthält.
    string get_url (); // Ein URL als Text.

    // Eine Objekthülle für die Beschreibungen der Modellvariablen.
    Lookup_Class get_class (); // Eine umhüllte Modellklasse.

    // Eine Objekthülle für die Beschreibungen der Abhängigkeiten zwischen Modellvariablen.
    Lookup_Dependencies get_dependencies (); // Eine umhüllte Menge von Modellabhängigkeiten.

    // Erzeugt ein neues Modellexemplar.
    // Die Variablen des neuen Modellexemplars sind durch die Modellklasse gegeben.
    Model create_model (); // Ein Modellexemplar.

    // Schließt ein Modellexemplar und gibt benutzte Ressourcen wieder frei.
    // Clients sollten die Methode aufrufen,
    // wenn sie ein Modellexemplar nicht mehr benötigen.
    void delete_model (in Model mdl); // Ein nicht mehr benötigtes Modellexemplar.
  };
  // Eine Liste aus Model_Factory Objekten.
  typedef sequence<Model_Factory> Model_Factories;

  // ............................................................................................
  // Ein Suchdienst für Modellfabriken.
  // Dieser Dienst stellt die Umkehrabbildung (Lookup_Class -> Model_Factory)
  // des Fabrikattributes "Modellklasse" bereit.
  // Es werden also Modellfabriken gesucht, die durch eine bestimmte Modellklasse beschrieben sind.
  interface Factory_Repository {

    // Trägt eine Modellfabrik beim Suchdienst ein.
    void export (in Model_Factory fact // Das einzutragende Model_Factory Objekt.
                 );
    // Trägt eine Modellfabrik beim Suchdienst aus.
    void resign (in Model_Factory fact // Das auszutragende Model_Factory Objekt.
                 );
    // Listet alle eingetragenen Modellfabriken auf.
    Model_Factories list_factories ();

    // Listet alle Modellklassen auf, zu denen Einträge von Modellfabriken existieren.
    Lookup_Classes list_classes ();

    // Sucht diejenigen Modellfabriken heraus, die durch eine Modellklasse beschrieben werden.
    Model_Factories lookup (// Eine Liste alle gefundenen Modellfabriken.
                            in Lookup_Class cls // Objekthülle der gesuchten Modellklasse
                                                // Der Suchdienst prüft auf Objektidentität
```

```
                                                // (und benutzt dazu "_is_equivalent")
                            );
  };
};
```

```
#define COSIM_LOOKUP
#ifndef COSIM_CLASS
#include "cosim_class.idl"
#endif
#ifndef COSIM_MODEL
#include "cosim_model.idl"
#endif
#pragma prefix "bachmann.informatik.unihh.de"

module cosim {

  // ============================================================================================
  // Beschreibt, wo ein benanntes Objekt innerhalb eines CORBA NamingService aufzufinden ist.
  // Ein benanntes Objekt ist in einem ::CosNaming::NamingContext eingetragen.
  // Der CORBA NamingService stellt eine Möglichkeit dar,
  // Objekte mit Namen zu versehen und wiederaufzufinden.
  // Der Name eingetragener Objekte ist für den Menschen leichter lesbar,
  // als eine Objektreferenz in IOR-Zahlencodierung.
  struct Lookup_Name {
    string                      context_url; // Objektreferenz als Text (in URL Form)
                                             // zu einem ::CosNaming::NamingContext Verzeichnis
                                             // unter dem ein benanntes Objekt abgelegt ist.
    ::CosNaming::NameComponent component;    // Der Namenseintrag im Verzeichnis.
                                             // Dieser Teil ist abgetrennt, weil er oft benutzt wird
                                             // und so eine wiederholte Aufbereitung
                                             // der Gesamtreferenz vermieden werden kann.
  };
  // ............................................................................................
  // Eine Liste aus Lookup_Name Objekten.
  typedef sequence<Lookup_Name> Lookup_Names;
  // ............................................................................................
  // Ein Objekt, das innerhalb eines NamingService wiederauffindbar ist.
  interface Lookup_Object {

    // Der Eintrag im NamingService, unter dem das Objekt wiederzufinden ist.
    // Somit besitzt das Objekt eine namentliche Selbstbeschreibung.
    Lookup_Name get_name (); // Unterteilung des Eintrags in Verzeichnis und nachschlagbaren Namen.
  };
  // ............................................................................................
  // Eine Objekthülle für Datenstrukturen.
  // Diese Schnittstelle ermöglicht dem Persistenz-Dienst Lookup_Repository,
  // den Datentyp umhüllter Objekte zu entdecken.
  // Jede Schnittstelle, die von der Lookup_Content Schnittstelle erbt
  // und eine Methode "get_data" ohne Parameter deklariert,
  // kann mit dem Persistenz-Dienst Lookup_Repository zusammenarbeiten.
  // Die Methode "get_data" extrahiert dabei die zu speichernden Datenstrukturen.
  interface Lookup_Content : Lookup_Object {

    // Interface Definition der realen, erbenden Schnittstelle.
    // Als erbende Schnittstellen sind bisher vorgesehen:
    // Lookup_Class, Lookup_Dependencies, Lookup_Connection, Lookup_Value
    ::CORBA::IDLType get_type ();
  };
  // ............................................................................................
  // Eine Objekthülle für Model_Class Strukturen.
  interface Lookup_Class : Lookup_Content {
    Model_Class get_data (); // Datenstruktur, die durch diese Hülle verpackt ist..
  };
  // ............................................................................................
  // Eine Objekthülle für Dependencies Strukturen.
  interface Lookup_Dependencies : Lookup_Content {
    Dependencies get_data (); // Datenstruktur, die durch diese Hülle verpackt ist.
  };
  // ............................................................................................
  // Eine Objekthülle für Model_Connection Strukturen.
  interface Lookup_Connection : Lookup_Content {
```

```
  Model_Connection get_data (); // Datenstruktur, die durch diese Hülle verpackt ist.
};
// ..............................................................................
// Eine Objekthülle für beliebige Datenstrukturen.
// Die eigentliche Datenstruktur ist nochmals als Any verpackt.
// Diese Objekthülle ist notwendig,
// damit Model_Class Strukturen andere Datenstrukturen referenzieren können
// (z.B. als Anfangswerte für Modellvariablen),
// die zur Compile-Zeit der CoSim-Anwendungen noch nicht bekannt waren.
interface Lookup_Value : Lookup_Content {
  any get_data (); // Datenstruktur, die durch diese Hülle verpackt ist.
};
// ..............................................................................
// Ein Lookup_Name ist in einem Lookup_Repository nicht aufzufinden.
exception Lookup_Content_Not_Found {string message;};
// ..............................................................................
// Der Typ eines Any Wertes,
// der in einem Lookup_Repository einzutragen ist,
// entspricht nicht dem deklarierten ::CORBA::TypeCode.
exception Lookup_Content_Not_Valid {string message;};
// ..............................................................................
// Ein Lookup_Name ist in einem Lookup_Repository nicht enthalten.
exception Lookup_Content_Not_Owned {string message;};
// ..............................................................................
// Verzeichnis zur Speicherung von Datenstrukturen,
// die als benannte Lookup_Content Objekte verpackt sind.
// Ein Lookup_Repository ist selbst ein benanntes Objekt,
// also in einem NamingService Verzeichnis eingetragen.
interface Lookup_Repository : Lookup_Object {

  // Speichert eine beliebige Datenstruktur und verknüpft sie mit einem Namen.
  Lookup_Content export // Ein neu erzeugtes Lookup_Content Objekt,
                       // dessen ::CORBA::IDLType     dem Parameter "type"  entspricht,
                       // dessen ::cosim::Lookup_Name dem Parameter "name"  entspricht und
                       // dessen "get_data" Wert      dem Parameter "value" entspricht.
  (in Lookup_Name      name,  // Der Name, unter dem die neue Objekthülle zu speichern ist.
                              // Falls für diesen Namen schon ein Eintrag existiert,
                              // werden die vorher gespeicherten Daten überschrieben.
                              // Das Verzeichnis trägt die neue Objekthülle
                              // selbsttätig beim NamingService ein.
   in any              value, // Eine als Any verpackte Datenstrukur,
                              // deren Typ zum angegebenen Typ Parameter paßt.
                              // Falls Lookup_Value Hüllen erzeugt werden sollen,
                              // muß der hier angegebene Wert ein Any Objekt sein,
                              // das eine weiteres Any Objekt enthält.
   in ::CORBA::IDLType type   // Ein ::CORBA::IDLType, d.h. eine Typdefinition,
                              // die mit Hilfe eines ::CORBA::Repository gewonnen wurde.
                              // Der Typ muß von Lookup_Content abgeleitet sein
                              // und eine parameterlose Methode "get_data" besitzen.
                              // Der Ergebnistyp dieser Methode muß mit dem Typ
                              // des übergebenen Wertes übereinstimmen.
                              // Z.Zt gibt es vier CoSim Schnittstellen,
                              // die gültige Schnittstellentypen darstellen:
                              // Lookup_Class, Lookup_Connection, Lookup_Value, Lookup_Dependencies.
   ) raises (Lookup_Content_Not_Valid // Der Datentyp des Parameters "value"
                                      // paßt nicht zum Parameter "type",
                                      // oder der Parameter "type" ist kein Typ,
                                      // der von der Schnitstelle Lookup_Content erbt
                                      // und eine parameterlose Methode "get_data" besitzt.
          );
  // Löscht einen gespeicherten Wert.
  // Das Verzeichnis trägt die neue Objekthülle selbsttätig beim NamingService aus.
  void resign (in Lookup_Name name // Der Name, unter dem ein Eintrag im NamingService erfolgt ist.
               )
    raises (Lookup_Content_Not_Owned // Dieses Verzeichnis enthält den gesuchten Wert nicht
          );
```

```
    // Sucht einen gespeicherten Wert.
    Lookup_Content lookup // Die Objekthülle, die unter dem angegebenen Namen verzeichnet ist.
    (in Lookup_Name name // Der Name, unter dem ein Eintrag im NamingService erfolgt ist.
     ) raises (Lookup_Content_Not_Found // Kein Wert unter dem angegebenen Namen verzeichnet.
               );
    // Listet die Namen alle gespeicherten Objekthüllen auf.
    Lookup_Names report (); // Eine Liste aller gespeicherten Objekthüllen.
  };
};
```

```
#define COSIM_EVENT
#ifndef COSIM_CLASS
#include "cosim_class.idl"
#endif
#pragma prefix "bachmann.informatik.unihh.de"

module cosim {

  // ============================================================================================
  // Ereignisse, die den Lebenszyklus von Modellexemplaren betreffen.
  // Diese Ereignisse werden asynchron weitergeleitet über einen typisierten Nachrichtenkanal
  // (meist an ein übergeordnetes gekoppeltes Modell oder eine Experimentierumgebung).
  // Je Modellexemplar existiert ein solcher Kanal.
  interface Model_Event {

    // Ein neues Modellexemplar wurde erzeugt.
    void model_created (in Model mdl // Das betroffene Modellexemplar
                        );
    // Ein Modellexemplar wurde geschlossen.
    void model_deleted (in Model mdl // Das betroffene Modellexemplar
                        );
    // ....................................................................................
    // Ein Modellexemplar ist ungültig geworden.
    // Modellfabriken billigen jedem Modellexemplar nur ein begrenzten Zeitraum zu,
    // in dem es ungenutzt bleiben darf.
    // Sobald irgendeine Kommunikation mit dem Modellexemplar stattfindet,
    // wird dieser Zeitraum erneuert.
    void model_purged (in Model       mdl,      // Das betroffene Modellexemplar.
                       in Model_State current, // Die aktuelle Modellnutzungsphase. (meist DELETED)
                       in Model_State previous // Die vorangegangene Modellnutzungsphase.
                       ));

    // ....................................................................................
    // Bei der Berechnung von Modellvariablen trat ein Fehler auf.
    void model_failed (in Model       mdl,     // Das betroffene Modellexemplar.
                       in Model_State current, // Die aktuelle Modellnutzungsphase.
                       in Model_State previous // Die vorangegangene Modellnutzungsphase.
                       );
    // ....................................................................................
    // Ein Modellexemplar ist in eine neue Modellnutzungsphase eingetreten.
    void state_changed (in Model       mdl,     // Das betroffene Modellexemplar.
                        in Model_State current, // Die aktuelle Modellnutzungsphase.
                        in Model_State previous // Die vorangegangene Modellnutzungsphase.
                        );
    // Die Ausführung der Modellnutzungsphase RUNNING hat begonnen.
    void execution_started (in Model mdl // Das betroffene Modellexemplar.
                            );
    // Die Ausführung der Modellnutzungsphase RUNNING ist beendet.
    void execution_stopped (in Model mdl // Das betroffene Modellexemplar.
                            );
  };
};
```

```
#define COSIM_DATATRANSFER
#ifndef COSIM_BASICS
#include "cosim_basics.idl"
#endif
#pragma prefix "bachmann.informatik.unihh.de"

module cosim {

  // ============================================================================================

  exception Connection_Not_Owned   {}; // Die Variable ist nicht Teil des Modellexemplars.
  exception Connection_Not_Import  {}; // Die Variable ist keine Eingabevariable.
  exception Connection_Not_Export  {}; // Die Variable ist keine Ausgabevariable.
  exception Connection_Not_Active  {}; // Die Variable ist bei der Datenübertragung nicht aktiv.
  exception Connection_Not_Passive {}; // Die Variable ist bei der Datenübertragung nicht passiv.
  exception Connection_Not_Tight   {}; // Die Variable benutzt keine  synchrone Datenübertragung.
  exception Connection_Not_Loose   {}; // Die Variable benutzt keine asynchrone Datenübertragung

  // ....................................................................................
  // Die Modellvariable ist in der aktuellen Modellnutzungsphase nicht verfügbar.
  exception Connection_State {
    Model_State current; // Die aktuelle Modellnutzungsphase.
  };
  // ....................................................................................
  // Die Modellvariable verlangt nach einem Kontextwert für eine bestimmte Hilfsvariable.
  exception Context_Missing {
    Lookup_Connection connection; // Die verlangte Hilfsvariable.
  };
  // ....................................................................................
  // Die Modellvariable weist den angegbenen Kontextwert für eine bestimmte Hilfsvariable zurück.
  exception Context_Unknown {
    Lookup_Connection connection; // Die zurückgewiesene Hilfsvariable.
  };
  // ============================================================================================
  // Ermöglicht das Lesen von Datenwerten einer eng gekoppelten Modellvariablen.
  interface Guided_Value_Supplier {

    // Liest den aktuellen Datenwert einer bestimmten Modellvariablen.
    // Nur die Datenwerte eng gekoppelter Variablen werden über diesen Methodenaufruf abgelesen.
    Guided_Value get_connection_value // Der Variablenwert zusammen mit den zugehörigen Kontextwerten.
    (in Lookup_Connection con, // Die Objekthülle der Beschreibung einer Modellvariablen.
     in Context_Values    ctx  // Die Liste notwendiger Kontextwerte als Schlüssel/Wert Paare.
     )
      raises (Connection_Not_Owned,   // Die Variable existiert in diesem Modellexemplar nicht.
              Connection_Not_Tight,   // Die Variable benutzt keine synchrone Datenübertragung.
              Connection_Not_Passive, // Die Variable ist bei der Datenübertragung nicht passiv.
              Connection_Not_Export,  // Die Variable ist keine Ausgabevariable.
              Connection_State,       // In der akutellen Phase ist die Variable nicht nutzbar.
              Context_Missing,        // Ein zusätzlicher Kontextwert wird benötigt.
              Context_Unknown         // Ein angegebener  Kontextwert wird nicht benötigt.
              );
  };
  // ============================================================================================
  // Ermöglicht das Schreiben von Datenwerten für eng gekoppelten Modellvariablen.
  interface Guided_Value_Consumer {

    // Ändert den aktuellen Datenwert einer bestimmten Modellvariablen.
    // Nur die Datenwerte eng gekoppelter Variablen werden über diesen Methodenaufruf geändert.
    void set_connection_value
    (in Lookup_Connection con, // Die Objekthülle der Beschreibung einer Modellvariablen.
     in Guided_Value      val  // Der neue Variablenwert zusammen mit den zugehörigen Kontextwerten.
     ) raises (Connection_Not_Owned,   // Die Variable existiert in diesem Modellexemplar nicht.
               Connection_Not_Tight,   // Die Variable benutzt keine synchrone Datenübertragung.
               Connection_Not_Passive, // Die Variable ist bei der Datenübertragung nicht passiv.
               Connection_Not_Import,  // Die Variable ist eine Ausgabevariable.
               Connection_State,       // In der akutellen Phase ist die Variable nicht nutzbar.
```

```
                Context_Missing,        // Ein zusätzlicher Kontextwert wird benötigt.
                Context_Unknown         // Ein angegebener Kontextwert wird nicht benötigt.
                );
  };
};
```

```
#ifndef _ORB_IDL
#include <orb.idl>
#define _ORB_IDL
#endif
#define LOOKUP_ALIASES
#pragma prefix "bachmann.informatik.unihh.de"

// Definitionen für die Realisierung des CoSim Lookup_Repository
module aliases {

// Einige Interface Repository Realisierungen (z.B. VisiBroker)
// liefert keine persistenten IRObject Objektreferenzen.
// Deshalb existiert eine Erweiterung,
// die einen Typ so beschreibt,
// daß er im Interface Repository wiedergefunden werden kann.
  interface Persistent_Type : ::CORBA::IDLType {
    ::CORBA::Repository get_repository ();           // Ein Typ ist im Interface Repository enthalten
    string              get_repository_id ();        // und hat entweder eine Kennung
    long                get_repository_primitive (); // oder ist primitiv (d.h. ein Basistyp).
  };
};
```

```
#define CAGE
#ifndef COSNAMING
#include "CosNaming.idl"
#define COSNAMING
#endif
#ifndef COSIM_FACTORY
#include "cosim_factory.idl"
#endif
#pragma prefix "bachmann.informatik.unihh.de"

// Definitionen für den Scanner des CoSim Adapter Generators
module cage {

  // ================================================================================================
  // Steuert die Aufnahme neuer Java Hüllen als Modellfabrik.
  // Details zur Bedeutung der Parameter sind zu finden in den Regeln für
  // <a href="http://cosim.sf.net/cage4j/location_rules.html">Verzeichnisse und Namen</a>

  interface Scanner {

    // Verzeichnis im CORBA NamingService, unter dem alle erzeugten Modellfabriken eingetragen sind.
    ::CosNaming::NamingContext get_root_context ();

    // Verzeichnis des Dateisystems, unter dem zu erzeugende Modellfabriken zu suchen sind.
    string get_root_directory ();

    // Durchsucht das Dateisystem nach Modellfabriken.
    ::cosim::Model_Factories scan_all ();

    // Übernimm eine bestimmten Java Hülle als Modellfabrik.
    ::cosim::Model_Factories include_directory
    (in string directory); // Das Dateiverzeichnis mit der Java Hülle.

    // Schließe die Modellfabrik einer bestimmten Java Hülle.
    void exclude_directory
    (in string directory); // Das Dateiverzeichnis mit der Java Hülle.
  };
};
```

```
#define COCO
#ifndef COSNAMING
#include "CosNaming.idl"
#define COSNAMING
#endif
#ifndef COSIM_FACTORY
#include "cosim_factory.idl"
#endif
#pragma prefix "bachmann.informatik.unihh.de"

// Definitionen für den Scanner des CoSim Component Connectors
module coco {

  // ============================================================================================
  // Steuert die Aufnahme neuer Kopplungssgraphen als Modellfabrik
  // Details zur Bedeutung der Parameter sind zu finden in den Regeln für
  // <a href="http://cosim.sf.net/coco4j/locations.html">Verzeichnisse und Namen</a>
  interface Scanner {

    // Verzeichnis im CORBA NamingService, unter dem alle erzeugten Modellfabriken eingetragen sind.
    ::CosNaming::NamingContext get_root_context   ();

    // Verzeichnis des Dateisystems, unter dem zu erzeugende Modellfabriken zu suchen sind.
    string get_root_directory ();

    // Durchsucht das Dateisystem nach Modellfabriken.
    ::cosim::Model_Factories scan_all ();

    // Übernimm einen bestimmten Kopplungsgraphen als Modellfabrik.
    ::cosim::Model_Factory include_graph
    (in string file); // Die ".coco"-Datei mit dem Kopplungsgraphen.

    // Schließe die Modellfabrik eines bestimmten Kopplungsgraphen.
    void exclude_graph
    (in string file); // Die ".coco"-Datei mit dem Kopplungsgraphen.

  };
};
```

```
#define DESMOJ
#pragma prefix "bachmann.informatik.unihh.de"

// Definitionen für das Simulationspaket DesmoJ als Statistiklieferanten
module desmoj {

  struct Queue_Statistics { // Zusammenstellung von statistischen Kennwerten einer Warteschlange
    long long observations; // Zahl der berücksichtigten Einträge
    long long cur_length;   // aktuelle Warteschlangenlänge
    long long min_length;   // minimale Warteschlangenlänge
    long long max_length;   // maximale Warteschlangenlänge
    double    avg_length;   // gewichtetes Mittel der Warteschlangenlänge
    long long non_wait;     // Zahl der Einträge ohne Wartezeit
    double    max_wait;     // maximale Wartezeit
    double    avg_wait;     // mittlere Wartezeit
  };
  typedef sequence <Queue_Statistics> Queue_Statistics_Sequence;

  // ============================================================================================
  struct Tally_Statistics { // Zusammenstellung von statistischen Kennwerten einer Zählers
    long long observations; // Zahl der berücksichtigten Werte
    double    current;      // akuteller Zählerstand
    double    minimum;      // minimaler Zählerstand
    double    maximum;      // maximaler Zählerstand
    double    mean;         // mittlerer Zählerstand
    double    deviation;    // Standardabweichung
  };
  typedef sequence <Tally_Statistics> Tally_Statistics_Sequence;

  // ============================================================================================
  struct Normal_Distribution_Settings { // Kennwerte einer Normalverteilung
    long long seed; // Startwert des Zufallszahlengenerators
    double    mean; // Mittelwert der Verteilung
    double    devt; // Standardabweichung der Verteilung
  };
  typedef sequence <Normal_Distribution_Settings> Normal_Distribution_Settings_Sequence;

  // ............................................................................................
  struct Exponential_Distribution_Settings { // Kennwerte einer Exponentialverteilung
    long long seed; // Startwert des Zufallszahlengenerators
    double    mean; // Mittelwert der Verteilung
  };
  typedef sequence <Exponential_Distribution_Settings> Exponential_Distribution_Settings_Sequence;
};
```

```
#define BARRELS
#ifndef DESMOJ
#include "desmoj.idl"
#endif
#pragma prefix "bachmann.informatik.unihh.de"

// Definitionen für das Beispielmodell zur Faßbefüllung.
module barrels {

  // ================================================================================================
  struct Consumer { // Parameter eines einzelnen Konsumten

                                    long long consume_unit; // konsumierte Fässer per Ereignis
    ::desmoj::Exponential_Distribution_Settings consume_time; // Zeit pro Konsumereignis
         ::desmoj::Normal_Distribution_Settings  travel_time; // Zeit für Lieferung / Rückgabe

    long long order_threshold; // untere Bestellgrenze
    long long order_maximum;   // obere Speichergrenze

    double barrel_size; // Faßgröße je Lieferung; alle Fässer haben die gleiche Füllmenge
  };
  // ................................................................................................
  typedef sequence <Consumer> Consumers;

  // ================================================================================================
  struct Lot { // Teilladung eines Fahrzeugs

    boolean   barrel_empty; // Sind die Fässer dieser Teilladung leer ?
    long long barrel_count; // Wieviele Fässer umfaßt diese Teilladung ?
    double    barrel_size;  // Alle Fässer dieser Teilladung haben die gleiche Füllmenge
  };
  // ................................................................................................
  typedef sequence <Lot> Lots;

  // ================================================================================================
  interface Vehicle { // Fahrzeug als Objekt, damit es eine Identität hat
    Lots get_load (); // eine Fahrzeugladung besteht aus mehreren Teilladungen
  };
  // ................................................................................................
  enum Vehicle_Strategy {FIRST_LOAD,  // FCFS aller Bestellungen
                         LEAST_LOAD,  // kleinere Bestellmengen haben Priorität
                         MATCH_LOAD}; // lieferbare Bestellmengen haben Priorität

  // ================================================================================================
  interface Barrel { // Faß als Objekt, damit es eine Identität hat
    double get_size (); // Füllmenge
  };
  // ................................................................................................
  enum  Barrel_Strategy {FIRST_BARREL,  // FCFS aller Fässer
                         SMALL_BARREL,  //  große Fässer haben Priorität
                         LARGE_BARREL,  // kleine Fässer haben Priorität
                         LEAST_BARREL}; // am wenigsten vorhandene Fässer haben Priorität

  // ================================================================================================
  struct Pressure { // Fülldrücke einer Abfüllstation
    double minimum;
    double maximum;
  };
};
```

B Deklarationen zur Einbettung von Java-Modellen

Dieser Anhang enthält die Schnittstellen zum Rahmenwerk des CoSim Adapter Generators, mit dem sich vorhandenene Modellrealisierung in Java in die CoSim-Architektur als Modellfabrik einbetten lassen. Tabelle B.1 gibt einen kurzen Überblick der vorhandenen Dateien.

Datei	**Inhalt**
`Dependencies.java`	Modellabhängigkeiten
`Denied_Exception.java`	Fehler bei der Parametrisierung
`State_Exception.java`	Fehler beim Zustandsübergang
`Value_Exception.java`	Fehler bei der losen Kopplung
`Context_Mapping.java`	Markierung für Kontextwerte als Parameter bei enger Kopplung
`Value_Mapping.java`	Hauptwert mit Kontextwerten
`Model_Wrapper.java`	Zustandsübergänge
`Model_Support.java`	Synchronisierter Zugriff auf eine zentrale Simulationsuhr
`Model_Implementation.java`	Variablen für den Ein- und Ausgabestrom
`Check_Declaration.java`	Beispiele für Übertragungsarten
`Check_Mapping.java`	Beispiel für Kontextwerte

Tabelle B.1: Übersicht der Cage-Definitionen

```
package de.unihh.informatik.bachmann.cage.wrapper;

public class Dependencies {

  public String   variable = "";
  public String[] requires = new String[] {};

  public Dependencies () {super ();}
  public Dependencies (String   variable,
                       String[] requires) {
    super ();
    this. variable = variable;
    this. requires = requires;}

}
```

```
package de.unihh.informatik.bachmann.cage.wrapper;

public class Denied_Exception
  extends Exception {

  protected String[] names = null;

  // ========================================================================

  // if switching from one state to another cannot succeed due to missing variable values,
  // please list the variable names in this constructor's parameter

  public Denied_Exception (String[] names) {
    this. names = names;}

  // ========================================================================

  public String[] get_names () {
    return names;}

}
```

```
package de.unihh.informatik.bachmann.cage.wrapper;

public class State_Exception
  extends Exception {

  // if executing your method implementations did not succeed,
  // please fill the field Exception.message with a detailed description

}
```

```
package de.unihh.informatik.bachmann.cage.wrapper;

public class Value_Exception
  extends RuntimeException {

  /* if executing any pull supplier method
   * (one starting with "has_", "snd_" or "prb_")
   * a Value_Exception declares that no more values will be available
   * for the associated simulation model variable.
   *
   * The associated CORBA event channel will receive an
   * org.omg.EventComm.Disconnected user exception
   * on any following pull or try_pull invocation.
   */

}
```

```
package de.unihh.informatik.bachmann.cage.wrapper;

public interface Context_Mapping {

}
```

```
package de.unihh.informatik.bachmann.cage.wrapper;

public class Value_Mapping
  implements Context_Mapping {

  // ========================================================================

  public boolean valid = true;

  // ========================================================================

  public Value_Mapping () {
    super ();}

  // ........................................................................

  public Value_Mapping (boolean valid) {
    super ();
    this. valid = valid;}

}
```

```
package de.unihh.informatik.bachmann.cage.wrapper;

public interface Model_Wrapper {

  // ========================================================================
  // destructor: clean up resources held by this object

  public void destroy ();

  // ========================================================================
  // setup actions                                                  // actions when switching

  public void prepare_launch  () throws State_Exception, Denied_Exception; // from CREATED to SETTING
  public void prepare_finish  () throws State_Exception, Denied_Exception; // from GETTING to CREATED

  public void prepare_set     () throws State_Exception, Denied_Exception; // from GETTING to SETTING
  public void prepare_run     () throws State_Exception, Denied_Exception; // from SETTING to RUNNING
  public void prepare_get     () throws State_Exception, Denied_Exception; // from RUNNING to GETTING

  public void prepare_suspend () throws State_Exception;                   // from RUNNING to STOPPED
  public void prepare_resume  () throws State_Exception;                   // from STOPPED to RUNNING
  public void prepare_abort   () throws State_Exception;                   // from any     to CREATED

  public void execute         () throws State_Exception; // actions to calculate values while RUNNING

}
```

```
package de.unihh.informatik.bachmann.cage.wrapper;

public interface Model_Support {

  // ========================================================================
  // Model_Wrapper implementations are allowed to call these methods.
  // They are implemented automatically by a generated subclass

  public Object sync_time (); // use this object in synchronized() blocks
  public double  get_time ();
  public void    set_time (double time);
```

```
  public Thread run_thread (String name, Runnable runnable);
  // creates an thread called "name" executing "runnable"

}
```

```
package de.unihh.informatik.bachmann.cage.wrapper;

import  java.io.PrintStream;

public class Model_Implementation
  implements Model_Support {

  // ============================================================================

  protected PrintStream cage_stream_output = null;
  protected PrintStream cage_stream_errors = null;

  // ============================================================================

  public Model_Implementation () {}
  public Model_Implementation (PrintStream output,
                               PrintStream errors) {
    this. cage_stream_output = output;
    this. cage_stream_output = errors;}

  // ============================================================================
  // Dummy implementations overwritten automatically by a generated subclass

  public Object sync_time () {return null;}
  public double  get_time () {return Double . NEGATIVE_INFINITY;}
  public void    set_time (double time) {};

  public Thread run_thread (String name, Runnable runnable) {return null;}

}
```

```
package de.unihh.informatik.schoellhammer.expu.check;

import  de.unihh.informatik.bachmann.cage.wrapper.Dependencies;
import  de.unihh.informatik.bachmann.cage.wrapper.Model_Wrapper;
import  de.unihh.informatik.bachmann.cage.wrapper.Model_Implementation;
import  de.unihh.informatik.bachmann.cage.wrapper.State_Exception;
import  de.unihh.informatik.bachmann.cage.wrapper.Denied_Exception;
import  de.unihh.informatik.bachmann.desmoj.Tally_Statistics;
import  java.io.PrintStream;

public class Check_Declaration
  extends    Model_Implementation
  implements Model_Wrapper {

  // ========================================================================
  // description

  public static String         the_factory_name () {return "Check";}
  public static String         the_factory_url  () {return "http://localhost/factories/check.html";}
  public static String         the_class_name   () {return "Check";}
  public static String         the_class_url    () {return "http://localhost/classes/check.html";}
  public static Dependencies[] the_dependencies () {return new Dependencies[] {};}

  // ========================================================================
  // initialization, destruction

  public Check_Declaration (PrintStream output,
                            PrintStream errors) {super (output, errors);}
  public void destroy () {}

  // ========================================================================
  //                          Activity,Coupling,Direction

  public static String           the_p_1_name       ()                  {return "p_1st";}
  public static long[][]         the_p_1_valids     ()                  {return    null;}
  public static long             the_p_1_initial    ()                  {return      -1;}
  public        long             see_p_1_parameter  ()                  {return      -1;}
  //                             ACTIVE,TIGHT,IMPORT

  public static String           the_p_2_name       ()                  {return "p_2nd";}
  public static long[][]         the_p_2_valids     ()                  {return    null;}
  public static long             the_p_2_initial    ()                  {return      -1;}
  public        long             rec_p_2_parameter  ()                  {return      -1;}
  public        long             try_p_2_parameter  ()                  {return      -1;}
  public        boolean          can_p_2_parameter  ()                  {return   false;}
  //                             ACTIVE,LOOSE,IMPORT

  public static String           the_p_3_name       ()                  {return "p_3rd";}
  public static long[][]         the_p_3_valids     ()                  {return    null;}
  public static long             the_p_3_initial    ()                  {return      -1;}
  public        void             set_p_3_parameter  (long p)            {               }
  //                            PASSIVE,TIGHT,IMPORT

  public static String           the_p_4_name       ()                  {return "p_4th";}
  public static long[][]         the_p_4_valids     ()                  {return    null;}
  public static long             the_p_4_initial    ()                  {return      -1;}
  public        void             sub_p_4_parameter  (long p)            {               }
  //                            PASSIVE,LOOSE,IMPORT

  // ........................................................................

  public static String           the_c_let_name     ()                  {return "c_let";}
  public static boolean          the_c_let_auto     ()                  {return    true;}
  public        int              see_c_let_initial  ()                  {return      -1;}
  public        void             let_c_let_clock    (int c)             {               }
  //                             ACTIVE,TIGHT,EXPORT

  public static String           the_c_see_name     ()                  {return "c_see";}
  public static boolean          the_c_see_auto     ()                  {return    true;}
  public        int              see_c_see_initial  ()                  {return      -1;}
  public        int              see_c_see_clock    ()                  {return      -1;}
  //                             ACTIVE,TIGHT,IMPORT

  public static String           the_c_get_name     ()                  {return "c_get";}
```

```
public static boolean          the_c_get_auto     ()                  {return    true;}
public        int              see_c_get_initial  ()                  {return      -1;}
public        int              get_c_get_clock    (int c)             {return      -1;}
//                            PASSIVE,TIGHT,EXPORT

public static String           the_c_set_name     ()                  {return "c_set";}
public static boolean          the_c_set_auto     ()                  {return    true;}
public        int              see_c_set_initial  ()                  {return      -1;}
public        void             set_c_set_clock    (int c)             {               }
//                            PASSIVE,TIGHT,IMPORT

public static String           the_c_pub_name     ()                  {return "c_pub";}
public static boolean          the_c_pub_auto     ()                  {return    true;}
public        int              see_c_pub_initial  ()                  {return      -1;}
public        void             pub_c_pub_clock    (int c)             {               }
//                             ACTIVE,LOOSE,EXPORT

public static String           the_c_rec_name     ()                  {return "c_rec";}
public static boolean          the_c_rec_auto     ()                  {return    true;}
public        int              see_c_rec_initial  ()                  {return      -1;}
public        int              rec_c_rec_clock    ()                  {return      -1;}
public        int              try_c_rec_clock    ()                  {return      -1;}
public        boolean          can_c_rec_clock    ()                  {return   false;}
//                             ACTIVE,LOOSE,IMPORT

public static String           the_c_snd_name     ()                  {return "c_snd";}
public static boolean          the_c_snd_auto     ()                  {return    true;}
public        int              see_c_snd_initial  ()                  {return      -1;}
public int                     snd_c_snd_clock    ()                  {return      -1;}
public int                     prb_c_snd_clock    ()                  {return      -1;}
public boolean                 has_c_snd_clock    ()                  {return   false;}
//                            PASSIVE,LOOSE,EXPORT

public static String           the_c_sub_name     ()                  {return "c_sub";}
public static boolean          the_c_sub_auto     ()                  {return    true;}
public        int              see_c_sub_initial  ()                  {return      -1;}
public void                    sub_c_sub_clock    (int c)             {               }
//                            PASSIVE,LOOSE,IMPORT

// ..................................................................

public static String           the_t_1_name       ()                  {return "t_1st";}
public        Check_Mapping    rec_t_1_runtime    ()                  {return    null;}
public        Check_Mapping    try_t_1_runtime    ()                  {return    null;}
public        boolean          can_t_1_runtime    ()                  {return   false;}
//                             ACTIVE,LOOSE,IMPORT

public static String           the_t_2_name       ()                  {return "t_2nd";}
public        void             sub_t_2_runtime    (Check_Mapping m)   {               }
//                            PASSIVE,LOOSE,IMPORT

public static String           the_t_3_name       ()                  {return "t_3rd";}
public        void             pub_t_3_runtime    (Check_Mapping m)   {               }
//                             ACTIVE,LOOSE,EXPORT

public static String           the_t_4_name       ()                  {return "t_4th";}
public        Check_Mapping    snd_t_4_runtime    ()                  {return    null;}
public        Check_Mapping    prb_t_4_runtime    ()                  {return    null;}
public        boolean          has_t_4_runtime    ()                  {return   false;}
//                            PASSIVE,LOOSE,EXPORT

// ..................................................................

public static String           the_r_1_name       ()                  {return "r_1st";}
public static String           the_r_1_repository ()                  {return    null;}
public         void            let_r_1_result     (Tally_Statistics t) {               }
//                             ACTIVE,TIGHT,EXPORT

public static String           the_r_2_name       ()                  {return "r_2nd";}
public static String           the_r_2_repository ()                  {return    null;}
public         void            pub_r_2_result     (Tally_Statistics t) {               }
//                             ACTIVE,LOOSE,EXPORT

public static String           the_r_3_name       ()                  {return "r_3rd";}
```

```
  public static String           the_r_3_repository ()                  {return    null;}
  public        Tally_Statistics get_r_3_result     ()                  {return    null;}
  //                            PASSIVE,TIGHT,EXPORT

  public static String           the_r_4_name       ()                  {return "r_4th";}
  public static String           the_r_4_repository ()                  {return    null;}
  public        Tally_Statistics snd_r_4_result     ()                  {return    null;}
  public        Tally_Statistics prb_r_4_result     ()                  {return    null;}
  public        boolean          has_r_4_result     ()                  {return   false;}
  //                            PASSIVE,LOOSE,EXPORT

  // ========================================================================
  // run
                                                                         // actions when switching
  public void prepare_launch  () throws State_Exception, Denied_Exception {}// from CREATED to SETTING
  public void prepare_finish  () throws State_Exception, Denied_Exception {}// from GETTING to CREATED
  public void prepare_set     () throws State_Exception, Denied_Exception {}// from GETTING to SETTING
  public void prepare_run     () throws State_Exception, Denied_Exception {}// from SETTING to RUNNING
  public void prepare_get     () throws State_Exception, Denied_Exception {}// from RUNNING to GETTING
  public void prepare_suspend () throws State_Exception                   {}// from RUNNING to STOPPED
  public void prepare_resume  () throws State_Exception                   {}// from STOPPED to RUNNING
  public void prepare_abort   () throws State_Exception                   {}// from any     to CREATED

  public void execute         () throws State_Exception {}

}
```

```
package de.unihh.informatik.schoellhammer.expu.check;

import  de.unihh.informatik.bachmann.cage.wrapper.Value_Mapping;

public class Check_Mapping
  extends    Value_Mapping {

  // ============================================================================
  // reflectable declarations

  public static String  time_name   = Check_Wrapper . the_c_let_name ();
  public        int     time_value  = -1;
  public        long    value       = -1;

  // ============================================================================
  // initialization

  public Check_Mapping () {
    super ();}

  // ............................................................................

  public Check_Mapping (boolean valid) {
    super (valid);}

  // ............................................................................

  public Check_Mapping (long value, int time_value) {
    this (true);
    this. time_value = time_value;
    this.      value = value;}

  // ............................................................................

  public Check_Mapping (boolean valid, long value, int time_value) {
    this (valid);
    this. time_value = time_value;
    this.      value = value;}

}
```

C Deklarationen für Berechnungsknoten in Modellkopplungen

Dieser Anhang enthält die Schnittstellen zum Rahmenwerk des CoSim Component Connectors. Das Rahmenwerk wird in Berechnungsknoten benötigt, die Zugriff auf die Datenwerte anderer Knoten oder auf die statische Struktur des Kopplungsgraphen benötigen. Tabelle C.1 gibt einen kurzen Überblick der vorhandenen Dateien.

Datei	**Inhalt**
`Calculation_Context.java`	Deklaration von Kontexten (vgl. Abb. 7.16)
`Calculation_Description.java`	Ausgabebeschreibung
`Calculation_Setup.java`	Syntaxprüfung und Initialisierung
`Calculation_Value.java`	Deklaration von Hauptwerten
`Checked_Calculation.java`	Berechnungsknoten
`Checked_Graph.java`	Kopplungsgraph
`Checked_Holder.java`	Zugangspunkt
`Checked_Model.java`	Modellexemplar
`Checked_Node.java`	beliebiger Knoten
`Checked_Publish.java`	externer Zugangspunkt
`Checked_Queue.java`	Datenpuffer
`Checked_Subscribe.java`	interner Zugangspunkt
`Environment.java`	abfragbare Datenwerte
`Finished_Environment_Exception.java`	Änderung der Nutzungsphase während Datenabfrage
`Instance.java`	Thread-Erzeugung, Zugriff auf ORB/POA
`Logger.java`	Zugriff auf Ein- und Ausgabestrom
`Neighbourhood.java`	Zugriff auf Kopplungsgraph und Datenpuffer
`Requestable.java`	erlaubte Schlüsselwerte für Datenabfragen
`Settings_Reply.java`	Antwortkonfiguration eines Datenpuffers
`Settings_Request.java`	Abfragekonfiguration eines Datenpuffers

Tabelle C.1: Übersicht der CoCo-Definitionen

```
package de.unihh.informatik.bachmann.coco.calculation;

import  de.unihh.informatik.bachmann.cosim.*;

public interface Calculation_Context
  extends        Calculation_Description {

  // calculate the set of all context values
  // by using the dependent values available in "env"
  public Context_Value[] calculate_context (String node, Environment env);

}
```

```
package de.unihh.informatik.bachmann.coco.calculation;

import  de.unihh.informatik.bachmann.cosim.*;

public interface Calculation_Description
  extends        Calculation_Setup {

  // get object references for context variables concerning "node".
  //
  // if there is a connection from itself into "node",
  // this will be the set of responded contexts
  //
  // if there is a connection from "node" into itself,
  // this will be the set of requested contexts
  //
  public Lookup_Connection[] describe_context (String node);

}
```

```
package de.unihh.informatik.bachmann.coco.calculation;

import  de.unihh.informatik.bachmann.coco.reader.Logger;
import  de.unihh.informatik.bachmann.coco.graph.Checked_Calculation;
import  de.unihh.informatik.bachmann.cosim.*;

public interface Calculation_Setup {

  // is this calculation an active part for a given node name ...
  public boolean is_active_from (String node); // ... of a connection from itself into "node"
  public boolean is_active_into (String node); // ... of a connection from "node" into itself

  // executed once after the calculation has been read from the ".coco" file
  public void setup_calculation (Logger              l,
                                 Instance            i,
                                 Neighbourhood       m,
                                 Checked_Calculation self,
                                 String              user);

  // executed each time when consistency checks are done on creating a ".coco" file
  public boolean check_calculation (Logger        l,
                                    Requestable[] r);

  // executed each time when the model instance changes its state
  public void    state_changed     (Model_State old_state,
                                    Model_State new_state);

}
```

```
package de.unihh.informatik.bachmann.coco.calculation;

import  de.unihh.informatik.bachmann.cosim.*;

import  org.omg.CORBA.IDLType;
```

```
public interface Calculation_Value
  extends        Calculation_Description {

  // get the CORBA type definition when "node" is requesting a value.
  // this method is called for compile time type checking
  // each time creating a ".coco" file
  public IDLType describe_value (String node);

  // create the complex value requested by "node".
  // use "env" for retreiving dependent values
  public Guided_Value calculate_value (String node, Environment env);

}
```

```
package de.unihh.informatik.bachmann.coco.graph;

import  java.io.Serializable;


public class Checked_Calculation
  extends    Checked_Node {

  public String   class_name;
  public String[] class_path;
  public String   setup_args;
}
```

```
package de.unihh.informatik.bachmann.coco.graph;

import  de.unihh.informatik.bachmann.cosim.*;

import  com.objectspace.jgl.HashMap;

import  java.io.Serializable;

public class Checked_Graph
  implements Serializable {

  // static description
  public String factory_name;
  public String factory_url;
  public String   class_name;
  public String   class_url;

  // partion of all graph elements
  // edges := queues,
  // nodes := others
  public Checked_Model       [] models;
  public Checked_Subscribe   [] subscribed;
  public Checked_Publish     [] published;
  public Checked_Calculation [] calculations;
  public Checked_Queue       [] queues;

  public Checked_Calculation    clock; // null, if no "<coco:simulation clock>"
}
```

```
package de.unihh.informatik.bachmann.coco.graph;

import  de.unihh.informatik.bachmann.cosim.*;

import  java.io.Serializable;

public class Checked_Holder
  extends    Checked_Node {
  // representation a model variable

  public Lookup_Name connection;
```

```
  public int         holder;
  // 1: CLOCK;
  // 2: CONTEXT;
  // 3: PARAMETER;
  // 4: RUNTIME;
  // 5: RESULT;
}
```

```
package de.unihh.informatik.bachmann.coco.graph;

import  de.unihh.informatik.bachmann.cosim.*;

import  java.io.Serializable;

public class Checked_Model
  extends Checked_Node {

  public Lookup_Name factory;

}
```

```
package de.unihh.informatik.bachmann.coco.graph;

import  de.unihh.informatik.bachmann.cosim.*;

import  java.io.Serializable;

public class Checked_Node
  implements Serializable {

  public String          name;

  public Checked_Queue[] from; // leaving  connections (receivers)
  public Checked_Queue[] into; // incoming connections (senders)
}
```

```
package de.unihh.informatik.bachmann.coco.graph;

import  de.unihh.informatik.bachmann.cosim.*;

import  java.io.Serializable;

public class Checked_Publish
  extends    Checked_Holder {
  // available to users of the coupled model

}
```

```
package de.unihh.informatik.bachmann.coco.graph;

import  de.unihh.informatik.bachmann.coco.editor.Settings_Request;
import  de.unihh.informatik.bachmann.coco.editor.Settings_Reply;

import  java.io.Serializable;


public class Checked_Queue
  implements Serializable {

  public Checked_Node from; // the connections' starting point
  public Checked_Node into; // the connections'   ending point

  public Settings_Request settings_request;
  public Settings_Reply   settings_reply;
}
```

```
package de.unihh.informatik.bachmann.coco.graph;

import  de.unihh.informatik.bachmann.cosim.*;

import  java.io.Serializable;

public class Checked_Subscribe
  extends    Checked_Holder {
  // variable of an embedded model

  public Checked_Model model;
}
```

```
package de.unihh.informatik.bachmann.coco.calculation;

import  org.omg.CORBA.Any;

public interface Environment {

  // this is some kind of a hierarchical hashtable of depth two
  // with "variable" being the key of the first level
  // and  "part"     being the key of the second level
  // for retrieving dependent values.
  //
  // use "null" for "part" to request "variable"s main value.
  //
  // a Finished_Environment_Exception runtime alert will be thrown,
  // if a state change occurs during requests
  public Any get (String variable, String part);

}
```

```
package de.unihh.informatik.bachmann.coco.calculation;

import  org.omg.CORBA.Any;

public class Finished_Environment_Exception
  extends RuntimeException {

  // ========================================================================
  // thrown if a state change occurs during environment requests

  public Finished_Environment_Exception () {
    super ();}

  public Finished_Environment_Exception (String message) {
    super (message);}
}
```

```
package de.unihh.informatik.bachmann.coco.calculation;

import  de.unihh.informatik.bachmann.coco.graph.Checked_Calculation;
import  de.unihh.informatik.bachmann.cosim.*;

import  org.omg.CORBA.ORB;
import  org.omg.PortableServer.POA;

public interface Instance {
  // there is one Instance objects for each model instance

  public ORB orb  (); // there should be only one ORB serving all requests
  public POA root (); // the ORBs RootPOA

  // call "runner"s run method once within a separate thread.
  // "name" is used as the threads name,
```

```
  // "originator" is used to determine a thread group.
  public Thread run_single (Checked_Calculation originator,
                            Runnable            runner,
                            String              name);

  // call "runner"s run method repeatedly within a separate thread
  // as long as the model instance stays in one of the given states.
  // execution will not start until one of the states is reached.
  // "name" is used as the threads name,
  // "originator" is used to determine a thread group.
  public Thread run_cycles (Checked_Calculation originator,
                            Runnable            runner,
                            String              name,
                            Model_State[]       synchronized_states);


}
```

```
package de.unihh.informatik.bachmann.coco.reader;

import  java.io.File;
import  java.io.PrintStream;
import  java.io.FileOutputStream;

public class Logger {
  // managing logging streams "err" and "out"

  public static String log_standard_property = "de.unihh.informatik.bachmann.cage.log_standard";
  public static String log_standard_default  = "true";

  // ..................................................................

  public static String  log_data_property = "de.unihh.informatik.bachmann.coco.log_data";
  public static String  log_data_default  = "false";
  public        boolean log_data          = false;

  // ..................................................................

  public PrintStream out = null;
  public PrintStream err = null;

  // ==================================================================

  public Logger (                                ) {super ();}
  public Logger (PrintStream out, PrintStream err) {super (); this. err = err; this. out = out;}

  // ==================================================================

  public void setup (String data_directory,
                     String path) {

    try {
      String log_path = data_directory + File . separatorChar + "logs";
      File   log_file = new File (log_path);
      log_file. mkdirs ();

      String log_out  = log_path + File . separatorChar + path + ".out";
      String log_err  = log_path + File . separatorChar + path + ".err";
      String log_std  = System. getProperty (log_standard_property, log_standard_default);

      if (log_std. equalsIgnoreCase (log_standard_default)) {
        this. out = System . out;
        this. err = System . err;}
      else {
        this. out = new PrintStream (new FileOutputStream (log_out), true);
        this. err = new PrintStream (new FileOutputStream (log_err), true);}

      String text_log_data = System . getProperty (log_data_property     , log_data_default);
      this.       log_data = text_log_data     . equalsIgnoreCase ("true");}
    catch (Exception e) {
      e. printStackTrace ();}}
```

```
  // ========================================================================

  public void close () {
    try {
      err. flush ();
      err. close ();
      out. flush ();
      out. close ();}
    catch (Exception e) {
      e. printStackTrace ();}}

}
```

```
package de.unihh.informatik.bachmann.coco.calculation;

import  de.unihh.informatik.bachmann.coco.graph.Checked_Graph;
import  de.unihh.informatik.bachmann.coco.graph.Checked_Queue;
import  de.unihh.informatik.bachmann.cosim.*;

public interface Neighbourhood {
  // if the calculation performs somesteps actively,
  // these methods will help to detect other nodes in the neighbourhood
  // and to communicate values with them.

  // the complete serializable graph representation.
  // the Checked_Calculation "self"
  // given within the Calculation_Setup.setup_calculation method
  // is the starting point within the graph.
  public Checked_Graph graph ();

  // request communication with available value queues
  public void         push  (Checked_Queue queue, Context_Value[] contexts, Guided_Value value);
  public Guided_Value pull  (Checked_Queue queue, Context_Value[] contexts);
  public boolean      empty (Checked_Queue queue, Context_Value[] contexts);

}
```

```
package de.unihh.informatik.bachmann.coco.calculation;

import  de.unihh.informatik.bachmann.cosim.*;

import  org.omg.CORBA.IDLType;

public class Requestable {
  // variables and their parts
  // available when a compile time check request is issued.
  // "variable" and "part" will be valid keys for Environment requests.

  public String  variable = null;
  public String  part     = null;
  public IDLType type     = null;

  // ========================================================================

  public Requestable () {
    super ();}

  // ........................................................................

  public Requestable (String  variable,
                      String  part,
                      IDLType type) {
    super ();

    this. variable = variable;
    this. part     = part;
    this. type     = type;}

}
```

```
package de.unihh.informatik.bachmann.coco.editor;

import  java.io.Serializable;

public class Settings_Reply
  implements Serializable {
  // user input by dialog

  public boolean separate = false;

  public boolean hold     = false;

  public boolean limit    = false;
  public int     size     = -1;
  public boolean shift    = false;


}
```

```
package de.unihh.informatik.bachmann.coco.editor;

import  java.io.Serializable;

public class Settings_Request
  implements Serializable {
  // user input by dialog

  public boolean generate = false;
  public int     rate     = -1;

  public boolean filter   = false;
  public int     step     = -1;

}
```

D Herleitungen für das Teilmodell „Befüllung“

D.1 Lösung der linearen Differentialgleichung

Gegeben sind der Wert $size$ für die Fassgröße in Litern, der Wert max für den maximalen Fülldruck in Litern je Sekunde und der Wert min für den minimalen Fülldruck in Litern je Sekunde, wobei $0 < min < max$ gelte. Gesucht ist die Funktion y, die die Füllhöhe eines Fasses in Abhängigkeit von der Zeit t angibt. Dabei steht t für die Zeitspanne in Sekunden, die seit Beginn des Füllvorgangs verstrichen ist. Die erste Ableitung y' gibt also den Fülldruck an. Die Hilfsgröße sei s die Zeitspanne, nach der der Sollwert $size$ der Füllhöhe erreicht wird. Die Randbedingungen der Funktion y ergeben sich nun nach Abschnitt 3.3. Zu Beginn des Füllvorgangs ist ein Fass leer, also $y(0) = 0$, und der maximale Fülldruck liegt an, also $y'(0) = max$. Zum Ende des Füllvorgangs herrscht nur noch der minimale Fülldruck, also $y'(s) = min$. Der Fülldruck nimmt zwischen den angegebenen Extrempunkten linear ab, also

$$y'(t) = \frac{y(s) - y(t)}{y(s)}(max - min) + min \tag{D.1}$$

Durch Einsetzen von 0 bzw. s für t in D.1 ergeben sich wieder die eben besprochen Randbedingungen. Da ein Fass zum Zeitpunkt s voll ist, also $y(s) = size$ gilt, kann man auch schreiben

$$y'(t) = \frac{size - y(t)}{size}(max - min) + min \tag{D.2}$$

Die Anwendung der allgemeinen Lösung für die inhomogene lineare Differentialgleichung aus (Bronstein und Semendjajew 1991, S. 420) benötigt eine Gleichung der Form $y' + f(x)y = g(x)$. Entsprechend umgeformt stellt sich D.2 dar als

$$y'(x) + \frac{max - min}{size}y(x) = max \tag{D.3}$$

Zunächst ist der integrierende Faktor $M(x)$ durch $e^{\int f(x)dx}$ zu berechnen. Mit dem Koeffizienten aus D.3 ergibt sich

$$M(x) = e^{\int \frac{max-min}{size}dx} = e^{x\frac{max-min}{size}} \tag{D.4}$$

Die allgemeine Lösung für die Füllhöhe y erhält man nun aus der Formel $y(x) = \frac{1}{M(x)}\int g(x)M(x)dx + C$, wobei $y(x_0) = C$. Für das Füllmodell gilt mit

D.4 also

$$\begin{aligned} y(x) &= \frac{1}{e^{x\frac{max-min}{size}}} \int max \cdot e^{x\frac{max-min}{size}} dx + 0 \\ &= e^{x\frac{min-max}{size}} [max \frac{size}{max-min} e^{x\frac{max-min}{size}}]_0^x \\ &= \frac{max \cdot size}{max-min}(1 - e^{x\frac{min-max}{size}}) \end{aligned} \tag{D.5}$$

Die Lösungsfunktion D.5 für die Füllhöhe war schon bei der Beschreibung des Teilmodells in Abbildung 3.5 dargestellt. Zur Probe sei noch einmal die Ableitung ausgerechnet als

$$\begin{aligned} y'(x) &= \frac{size \cdot max}{max-min} \frac{min-max}{size}(1 - e^{x\frac{min-max}{size}}) \\ &= max \cdot e^{x\frac{min-max}{size}} \end{aligned} \tag{D.6}$$

Einsetzen von D.6 in D.3 ergibt

$$\begin{aligned} max &= y'(x) + \frac{max-min}{size} y(x) \\ &= max \cdot e^{x\frac{min-max}{size}} + \frac{max-min}{size} \frac{size \cdot max}{max-min}(1 - e^{x\frac{min-max}{size}}) \\ &= max \cdot e^{x\frac{min-max}{size}} + max \cdot (1 - e^{x\frac{min-max}{size}}) \\ &= max \end{aligned}$$

D.2 Auflösung der Fülldauer

Um die Fülldauer s direkt zu berechnen, setzt man

$$\begin{aligned} size &= y(s) = \frac{size \cdot max}{max-min}(1 - e^{s\frac{min-max}{size}}) \\ \frac{max-min}{max} &= 1 - e^{s\frac{min-max}{size}} \\ e^{s\frac{min-max}{size}} &= 1 - \frac{max-min}{max} = \frac{min}{max} \\ s\frac{min-max}{size} &= \ln \frac{min}{max} \\ s &= \frac{min-max}{size} \ln \frac{min}{max} \\ s &= \frac{max-min}{size} \ln \frac{max}{min} \end{aligned} \tag{D.7}$$

D.7 ergibt für das Grundszenario `Barrels` aus Abschnitt E.1 mit $min = 0,05000$, $max = 0,83333$ und $size = 50,0$ beispielsweise eine Fülldauer von $179,58$ Sekunden, entsprechend $0,0020785$ Tagen, pro Fass. Der Wert für die Befüllung von 100 Fässern ($0,20785$ Tage) findet sich auch in der Warteschlangenstatistik E.8 der Füllstationen als durchschnittliche Wartezeit wieder.

E Experimentergebnisse

Dieser Anhang enthält die Experimenteinstellungen und -Ergebnisse verschiedene Szenarien, unter denen das Beispielmodell ausgeführt wurde. Tabelle E.1 gibt einen kurzen Überblick der Szenarien.

	Szenario	**Beschreibung**
E.1	Barrels	Grundszenario aus Abschnitt 9.6
E.2	Consume	Halbierung der Zwischenwerte der Konsumzeiten
E.3	Filling	Halbierung der Befüllungsdrücke
E.4	Initial	Halbierung der gefüllten Fässer zum Simulationsstart
E.5	Vehicle	Halbierung der Anzahl verfügbarer Fahrzeuge

Tabelle E.1: Übersicht der Szenarien für das Beispielmodell

E.1 Szenario Barrels

consumer	barrel_size	order_threshold	order_maximum	consume_unit	consume_time.seed	consume_time.mean	travel_time.seed	travel_time.mean	travel_time.devt
0	50.00000	1200	1600	100	101	0.66666	201	0.04166	0.01041
1	50.00000	1200	2000	100	102	0.41666	202	0.12500	0.03125
2	50.00000	1600	2000	100	103	0.33333	203	0.16666	0.04166
3	50.00000	1600	2400	100	104	0.25000	204	0.33333	0.08333
4	50.00000	2000	2400	100	105	0.16666	205	0.37500	0.09375
5	50.00000	2000	2800	100	106	0.16666	206	0.45833	0.11458

Tabelle E.2: Parameterbelegung des Feldes `consumer` im Szenario `barrels`

initial	barrel_empty	barrel_count	barrel_size
0	true	0	50.00000
1	false	1200	50.00000

Tabelle E.3: Parameterbelegung des Feldes `initial` im Szenario `barrels`

Stop	14.00000
Vehicles.Capacity	400
Vehicles.Count	3
Strategy.Barrel	FIRST_BARREL
Strategy.Vehicle	FIRST_LOAD
Pressure.Fill.*[1,2,3]*.minimum	0.05000
Pressure.Fill.*[1,2,3]*.maximum	0.83333

Tabelle E.4: Belegung der Einzelparameter im Szenario `barrels`

Anfang	Ende
0.00000	...
0.21309	13.82732
0.21325	13.83476
0.22036	13.90368
0.31972	13.92711
0.34053	13.97077
...	13.99863

Tabelle E.5: Ereigniszeitpunkte im Szenario `barrels`

deliveries	observations	current	minimum	maximum	mean	deviation
0	2	0.02343	0.02343	0.04385	0.03364	0.01444
1	8	0.09105	0.08350	0.17288	0.12583	0.03201
2	7	0.16388	0.07424	0.30399	0.17192	0.07685
3	4	0.29622	0.22794	0.46638	0.30619	0.11115
4	17	0.33252	0.32143	0.55394	0.42520	0.07410
5	20	0.42025	0.22264	0.64586	0.47130	0.11404

Tabelle E.6: Feld der Zählerstatistiken `deliveries` im Szenario `barrels`

returns	observations	current	minimum	maximum	mean	deviation
0	2	400.00000	400.00000	400.00000	400.00000	0.00000
1	8	400.00000	400.00000	400.00000	400.00000	0.00000
2	7	400.00000	400.00000	400.00000	400.00000	0.00000
3	4	400.00000	400.00000	400.00000	400.00000	0.00000
4	16	100.00000	0.00000	400.00000	287.50000	145.48769
5	20	100.00000	0.00000	400.00000	265.00000	149.64871

Tabelle E.7: Feld der Zählerstatistiken `returns` im Szenario `barrels`

Warteschlange:	*waiting*	*filling* [1]	*filling* [2]	*filling* [3]
observations	60	50	50	50
cur_length	0	1	1	1
min_length	0	0	0	0
max_length	1	1	1	1
avg_length	0.04566	0.75173	0.75173	0.75173
non_wait	39	0	0	0
max_wait	0.08520	0.20785	0.20785	0.20785
avg_wait	0.01065	0.20785	0.20785	0.20785

Tabelle E.8: Statistiken der Warteschlangen im Szenario `barrels`

E.2 Szenario Consume

consumer	barrel_size	order_threshold	order_maximum	consume_unit	consume_time.seed	consume_time.mean	travel_time.seed	travel_time.mean	travel_time.devt
0	50.00000	1200	1600	100	101	0.33333	201	0.04166	0.01041
1	50.00000	1200	2000	100	102	0.20833	202	0.12500	0.03125
2	50.00000	1600	2000	100	103	0.16666	203	0.16666	0.04166
3	50.00000	1600	2400	100	104	0.12500	204	0.33333	0.08333
4	50.00000	2000	2400	100	105	0.08333	205	0.37500	0.09375
5	50.00000	2000	2800	100	106	0.08333	206	0.45833	0.11458

Tabelle E.9: Parameterbelegung des Feldes `consumer` im Szenario `consume`

initial	barrel_empty	barrel_count	barrel_size
0	true	0	50.00000
1	false	1200	50.00000

Tabelle E.10: Parameterbelegung des Feldes `initial` im Szenario `consume`

Stop	14.00000
Vehicles.Capacity	400
Vehicles.Count	3
Strategy.Barrel	FIRST_BARREL
Strategy.Vehicle	FIRST_LOAD
Pressure.Fill.*[1,2,3]*.minimum	0.05000
Pressure.Fill.*[1,2,3]*.maximum	0.83333

Tabelle E.11: Belegung der Einzelparameter im Szenario `consume`

Anfang	*Ende*
0.00000	...
0.10654	13.60802
0.10662	13.66652
0.11018	13.73238
0.15986	13.90288
0.17026	13.96260
...	

Tabelle E.12: Ereigniszeitpunkte im Szenario `consume`

deliveries	*observations*	*current*	*minimum*	*maximum*	*mean*	*deviation*
0	1	0.04385	0.04385	0.04385	0.04385	—
1	8	0.09105	0.08350	0.17288	0.12583	0.03201
2	7	0.16388	0.07424	0.23724	0.16178	0.05918
3	4	0.29622	0.22794	0.46638	0.30619	0.11115
4	20	0.51034	0.30616	0.57168	0.42663	0.08191
5	20	0.42025	0.22264	0.63296	0.45016	0.10915

Tabelle E.13: Feld der Zählerstatistiken `deliveries` im Szenario `consume`

returns	*observations*	*current*	*minimum*	*maximum*	*mean*	*deviation*
0	1	400.00000	400.00000	400.00000	400.00000	—
1	8	400.00000	400.00000	400.00000	400.00000	0.00000
2	7	400.00000	400.00000	400.00000	400.00000	0.00000
3	4	400.00000	400.00000	400.00000	400.00000	0.00000
4	19	100.00000	0.00000	400.00000	236.84211	146.09938
5	19	100.00000	100.00000	400.00000	273.68421	152.17718

Tabelle E.14: Feld der Zählerstatistiken `returns` im Szenario `consume`

Warteschlange:	*waiting*	*filling* [1]	*filling* [2]	*filling* [3]
observations	61	55	55	55
cur_length	0	1	1	1
min_length	0	0	0	0
max_length	1	1	1	1
avg_length	0.01241	0.82244	0.81810	0.81810
non_wait	55	0	0	0
max_wait	0.05893	0.20785	0.20785	0.20785
avg_wait	0.00284	0.20785	0.20785	0.20785

Tabelle E.15: Statistiken der Warteschlangen im Szenario `consume`

E.3 Szenario Filling

consumer	barrel_size	order_threshold	order_maximum	consume_unit	consume_time.seed	consume_time.mean	travel_time.seed	travel_time.mean	travel_time.devt
0	50.00000	1200	1600	100	101	0.66666	201	0.04166	0.01041
1	50.00000	1200	2000	100	102	0.41666	202	0.12500	0.03125
2	50.00000	1600	2000	100	103	0.33333	203	0.16666	0.04166
3	50.00000	1600	2400	100	104	0.25000	204	0.33333	0.08333
4	50.00000	2000	2400	100	105	0.16666	205	0.37500	0.09375
5	50.00000	2000	2800	100	106	0.16666	206	0.45833	0.11458

Tabelle E.16: Parameterbelegung des Feldes `consumer` im Szenario `filling`

initial	barrel_empty	barrel_count	barrel_size
0	true	0	50.00000
1	false	1200	50.00000

Tabelle E.17: Parameterbelegung des Feldes `initial` im Szenario `filling`

Stop	14.00000
Vehicles.Capacity	400
Vehicles.Count	3
Strategy.Barrel	FIRST_BARREL
Strategy.Vehicle	FIRST_LOAD
Pressure.Fill.[1,2,3].minimum	0.02500
Pressure.Fill.[1,2,3].maximum	0.41666

Tabelle E.18: Belegung der Einzelparameter im Szenario `filling`

Anfang	*Ende*
0.00000	...
0.21309	13.82732
0.21325	13.83476
0.22036	13.90368
0.31972	13.92711
0.34053	13.97077
...	13.99863

Tabelle E.19: Ereigniszeitpunkte im Szenario `filling`

deliveries	*observations*	current	minimum	*maximum*	*mean*	*deviation*
0	2	0.02343	0.02343	0.04385	0.03364	0.01444
1	8	0.09105	0.08350	0.17288	0.12583	0.03201
2	7	0.16388	0.07424	0.30399	0.17192	0.07685
3	4	0.29622	0.22794	0.46638	0.30619	0.11115
4	17	0.33252	0.32143	0.60564	0.42956	0.08244
5	20	0.42025	0.22264	0.65004	0.46760	0.11461

Tabelle E.20: Feld der Zählerstatistiken `deliveries` im Szenario `filling`

returns	observations	current	minimum	maximum	mean	deviation
0	2	400.00000	400.00000	400.00000	400.00000	0.00000
1	8	400.00000	400.00000	400.00000	400.00000	0.00000
2	7	400.00000	400.00000	400.00000	400.00000	0.00000
3	4	400.00000	400.00000	400.00000	400.00000	0.00000
4	16	100.00000	0.00000	400.00000	287.50000	145.48769
5	20	100.00000	0.00000	400.00000	265.00000	149.64871

Tabelle E.21: Feld der Zählerstatistiken `returns` im Szenario `filling`

Warteschlange:	*waiting*	*filling* [1]	*filling* [2]	*filling* [3]
observations	60	29	29	29
cur_length	0	1	1	1
min_length	0	0	0	0
max_length	1	1	1	1
avg_length	0.04566	0.88615	0.87111	0.87111
non_wait	39	0	0	0
max_wait	0.08773	0.41569	0.41569	0.41569
avg_wait	0.01065	0.41569	0.41569	0.41569

Tabelle E.22: Statistiken der Warteschlangen im Szenario `filling`

E.4 Szenario Initial

consumer	barrel_size	order_threshold	order_maximum	consume_unit	consume_time.seed	consume_time.mean	travel_time.seed	travel_time.mean	travel_time.devt
0	50.00000	1200	1600	100	101	0.66666	201	0.04166	0.01041
1	50.00000	1200	2000	100	102	0.41666	202	0.12500	0.03125
2	50.00000	1600	2000	100	103	0.33333	203	0.16666	0.04166
3	50.00000	1600	2400	100	104	0.25000	204	0.33333	0.08333
4	50.00000	2000	2400	100	105	0.16666	205	0.37500	0.09375
5	50.00000	2000	2800	100	106	0.16666	206	0.45833	0.11458

Tabelle E.23: Parameterbelegung des Feldes `consumer` im Szenario `initial`

initial	barrel_empty	*barrel_count*	barrel_size
0	true	0	50.00000
1	false	*600*	50.00000

Tabelle E.24: Parameterbelegung des Feldes `initial` im Szenario `initial`

Stop	14.00000
Vehicles.Capacity	400
Vehicles.Count	3
Strategy.Barrel	FIRST_BARREL
Strategy.Vehicle	FIRST_LOAD
Pressure.Fill.*[1,2,3]*.minimum	0.05000
Pressure.Fill.*[1,2,3]*.maximum	0.83333

Tabelle E.25: Belegung der Einzelparameter im Szenario `initial`

Anfang	*Ende*
0.00000	...
0.21309	13.82022
0.21325	13.84475
0.22036	13.89027
0.31972	13.93078
0.34053	13.96690
...	13.98132

Tabelle E.26: Ereigniszeitpunkte im Szenario `initial`

deliveries	observations current	*minimum*	*maximum*	*mean*	*deviation*
0	1	0.04385	0.04385	0.04385	0.04385	—
1	8	0.09105	0.08350	0.17288	0.12583	0.03201
2	7	0.16388	0.07424	0.26189	0.16591	0.06562
3	4	0.29622	0.22794	0.46638	0.30619	0.11115
4	16	0.37174	0.32143	0.78853	0.51817	0.15041
5	19	0.30055	0.22264	0.74317	0.49092	0.14410

Tabelle E.27: Feld der Zählerstatistiken `deliveries` im Szenario `initial`

returns	observations current	*minimum*	*maximum*	*mean*	*deviation*
0	1	400.00000	400.00000	400.00000	400.00000	—
1	7	400.00000	400.00000	400.00000	400.00000	0.00000
2	7	400.00000	400.00000	400.00000	400.00000	0.00000
3	4	400.00000	400.00000	400.00000	400.00000	0.00000
4	15	100.00000	0.00000	400.00000	293.33333	138.70146
5	19	100.00000	100.00000	400.00000	273.68421	144.69165

Tabelle E.28: Feld der Zählerstatistiken `returns` im Szenario `initial`

Warteschlange:	*waiting*	*filling* [1]	*filling* [2]	*filling* [3]
observations	56	50	49	49
cur_length	0	1	1	1
min_length	0	0	0	0
max_length	3	1	1	1
avg_length	0.16527	0.74579	0.74058	0.73094
non_wait	35	0	0	0
max_wait	0.41570	0.20785	0.20785	0.20785
avg_wait	0.04131	0.20785	0.20785	0.20785

Tabelle E.29: Statistiken der Warteschlangen im Szenario `initial`

E.5 Szenario Vehicle

consumer	barrel_size	order_threshold	order_maximum	consume_unit	consume_time.seed	consume_time.mean	travel_time.seed	travel_time.mean	travel_time.devt
0	50.00000	1200	1600	100	101	0.66666	201	0.04166	0.01041
1	50.00000	1200	2000	100	102	0.41666	202	0.12500	0.03125
2	50.00000	1600	2000	100	103	0.33333	203	0.16666	0.04166
3	50.00000	1600	2400	100	104	0.25000	204	0.33333	0.08333
4	50.00000	2000	2400	100	105	0.16666	205	0.37500	0.09375
5	50.00000	2000	2800	100	106	0.16666	206	0.45833	0.11458

Tabelle E.30: Parameterbelegung des Feldes `consumer` im Szenario `vehicle`

initial	barrel_empty	barrel_count	barrel_size
0	true	0	50.00000
1	false	1200	50.00000

Tabelle E.31: Parameterbelegung des Feldes `initial` im Szenario `vehicle`

Stop	14.00000
Vehicles.Capacity	200
Vehicles.Count	2
Strategy.Barrel	FIRST_BARREL
Strategy.Vehicle	FIRST_LOAD
Pressure.Fill.*[1,2,3]*.minimum	0.05000
Pressure.Fill.*[1,2,3]*.maximum	0.83333

Tabelle E.32: Belegung der Einzelparameter im Szenario `vehicle`

Anfang	*Ende*
0.00000	...
0.21309	13.65052
0.21325	13.82515
0.22036	13.88756
0.31972	13.88811
0.34053	13.94674
...	13.95106

Tabelle E.33: Ereigniszeitpunkte im Szenario `vehicle`

deliveries	*observations*	*current*	*minimum*	*maximum*	*mean*	*deviation*
0	0	0.00000	0.00000	0.00000	—	—
1	0	0.00000	0.00000	0.00000	—	—
2	2	0.16650	0.16650	0.23302	0.19976	0.04703
3	4	0.29622	0.22794	0.46638	0.30619	0.11115
4	14	0.32143	0.32143	0.58746	0.44816	0.07460
5	14	0.22263	0.22264	0.72899	0.47859	0.14222

Tabelle E.34: Feld der Zählerstatistiken `deliveries` im Szenario `vehicle`

returns	*observations*	*current*	*minimum*	*maximum*	*mean*	*deviation*
0	0	0.00000	0.00000	0.00000	—	—
1	0	0.00000	0.00000	0.00000	—	—
2	2	200.00000	200.00000	200.00000	200.00000	0.00000
3	2	200.00000	200.00000	200.00000	200.00000	0.00000
4	14	200.00000	200.00000	200.00000	200.00000	0.00000
5	14	200.00000	200.00000	200.00000	200.00000	0.00000

Tabelle E.35: Feld der Zählerstatistiken `returns` im Szenario `vehicle`

Warteschlange:	*waiting*	*filling* [1]	*filling* [2]	*filling* [3]
observations	34	21	21	21
cur_length	0	1	0	0
min_length	0	0	0	0
max_length	1	1	1	1
avg_length	0.04219	0.32615	0.31177	0.31177
non_wait	20	0	0	0
max_wait	0.13583	0.20785	0.20785	0.20785
avg_wait	0.01737	0.20785	0.20785	0.20785

Tabelle E.36: Statistiken der Warteschlangen im Szenario `vehicle`

Literaturverzeichnis

Andradóttir u. a. 1997 Andradóttir, S. (Hrsg.) ; Healy, K. J. (Hrsg.) ; Withers, D. H. (Hrsg.) ; Nelson, B. L. (Hrsg.): *Winter Simulation Conference*. SCS, 1997

Bachmann 2000 Bachmann, Ralf: HLA und CORBA: Partner oder Konkurrenten? In: (Möller 2000), S. 141–146

Bachmann 2001 Bachmann, Ralf: *HLA und CORBA: Ansätze zur verbesserten Integration*. Vortrag im HLA-Forum der Magdeburger Tagung Simulation und Visualisierung. 2001. – http://www.kompetenzzentrum-hla.de/documents/hlaforum/2001/HLAForum2001DOC-Bachmann.pdf

Bachmann u. a. 1999 Bachmann, Ralf ; Gehlsen, Björn ; Lamersdorf, Winfried ; Page, Bernd: Nutzerführung und Dienste für heterogene, verteilte Umweltinformation / Universität Hamburg, Fachbereich Informatik, ASI / VSYS. 1999. – Abschlußbericht des Forschungsprojekts TIDE.

Becken und Bosselmann 1998 Becken, Olaf ; Bosselmann, Michael: *Realisierung der Netzwerkkomponente eines verteilten Modellierungs- und Simulationssystems mittels OSF-DCE*, Universität Hamburg, Fachbereich Informatik, Diplomarbeit, 1998

Boger 2001 Boger, Marko: *Entwicklung verteilter Softwaresysteme: Integration von Verteilung, Nebenläufigkeit und Persistenz*, Universität Hamburg, Fachbereich Informatik, Dissertation, 2001

Box 1998 Box, Don: *Essential COM*. Addison-Wesley, 1998

Bronstein und Semendjajew 1991 Bronstein, I. N. ; Semendjajew, K. A. ; Grosche, G. (Hrsg.) ; Ziegler, V. (Hrsg.) ; Ziegler, D. (Hrsg.): *Taschenbuch der Mathematik*. 25. Teubner, 1991

Buss 2000 Buss, Arnold H.: Component-Based Simulation Modeling. In: (Joines u. a. 2000), S. 964–971

Cardelli 1991 Cardelli, Luca: Typeful Programming. In: Neuhold, E. J. (Hrsg.) ; Paul, M. (Hrsg.): *Formal Description of Programming Concepts*. Springer, 1991, S. 431–507

Chan und Spracklen 1999 Chan, Alvin ; Spracklen, Tim: Web-Based Distributed Object Simulation Framework. In: Obaidat, Mohammad S. (Hrsg.) ; Nisanci, Abe (Hrsg.) ; Sadoun, Balqies (Hrsg.): *Summer Simulation Conference*. Chicago, Illinois : SCS, July 1999, S. 9–14

Chandry und Misra 1979 Chandry, K. M. ; Misra, Jayadev: Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. In: *IEEE Transactions on Software Engineering* 5 (1979), Nr. 5, S. 440–452

Cho 2001 Cho, Young K.: *RTDEVS/CORBA: A Distributed Object Computing Environment for Simulation-Based Design of Real-Time Discrete Event Systems*, University of Arizona, Department of Electrical and Computer Engineering, Dissertation, 2001

Cholkar und Koopman 1999 Cholkar, Arjun ; Koopman, Philip: A widely deployable Web-based network simulation framework using CORBA IDL-based APIs. In: (Farrington u. a. 1999), S. 1587–1594

Coulouris u. a. 1994 Coulouris, George ; Dollimore, Jean ; Kindberg, Tim: *Distributed Systems: Concepts and Design*. 2. Addison-Wesley, 1994

Crane 1997 Crane, John S.: *Dynamic Binding for Distributed Systems*, University of London, Imperial College of Science, Department of Computing, Dissertation, 1997

Cubert und Fishwick 1997 Cubert, Robert M. ; Fishwick, Paul A.: A Framework for Distributed Object-Oriented Multimodeling and Simulation. In: (Andradóttir u. a. 1997), S. 1315–1322

Cubert und Fishwick 1998a Cubert, Robert M. ; Fishwick, Paul A.: OOPM: An Object-Oriented Multimodeling and Simulation Application Framework. In: *Simulation* 70 (1998), Nr. 6, S. 379–395

Cubert und Fishwick 1998b Cubert, Robert M. ; Fishwick, Paul A.: Software Architecture for Distributed Simulation. In: Sisti, Alex. F. (Hrsg.): *Enabling Technology for Simulation Science*. Orlando, FL : The International Society for Optical Engineering, April 1998, S. 154–163. – SPIE Vol. 3369

Dahmann u. a. 1997 Dahmann, Judith S. ; Fujimoto, Richard M. ; Weatherly, Richard M.: The Department of Defense High Level Architecture. In: (Andradóttir u. a. 1997), S. 142–149

Davis und Moeller 1999 Davis, Wayne J. ; Moeller, Gerald L.: The High Level Architecture: Is There a Better Way? In: (Farrington u. a. 1999), S. 1595–1601

Deussen u. a. 1999 Deussen, Oliver (Hrsg.) ; Hinz, Volker (Hrsg.) ; Lorenz, Peter (Hrsg.): *Simulation und Visualisierung*. SCS Europe, März 1999

DMSO 2000 *High Level Architecture Run-Time Infrastructure: RTI Programmer's Guide*. RTI 1.3 - Next Generation Version 3. Department of Defense, Defense Modeling and Simulation Office, April 2000

Dorwarth u. a. 1997 Dorwarth, Heiko ; Schumann, Marco ; Seibt, Frank: Komponenten und Konzepte für Simulationsumgebungen im WWW. In: Kuhn, A. (Hrsg.) ; Wenzel, S. (Hrsg.): *11. Symposium Simulationstechnik*. Dortmund : Vieweg, November 1997, S. 593–598

Dörnhöfer 1991 Dörnhöfer, Klaus: *Die Benutzerumgebung von Simplex II*, Universität Erlangen-Nürnberg, Institut für Mathematische Maschinen und Datenverarbeitung, Dissertation, 1991

Faden u. a. 1999 Faden, Michael ; Fischer, Jochen ; Forkert, Tomas ; Koudriavski, Michael ; Schreiber, Andreas ; Strietzel, Martin ; Wins, Michael: Parallele Kommunikation zwischen verteilten Applikationen. Deutsches Zentrum für Luft- und Raumfahrt, Simulations- und Softwaretechnik, April 1999. – Technische Dokumentation. – URL `http://www.sistec.dlr.de/tent/docs/pdf/Parallele_Kommunikation.pdf`.

Farrington u. a. 1999 Farrington, P. A. (Hrsg.) ; Nembhard, H. B. (Hrsg.) ; Sturrock, D. T. (Hrsg.) ; Evans, G. W. (Hrsg.): *Winter Simulation Conference*. SCS, 1999

Ferris und Farrell 2003 Ferris, Christopher ; Farrell, Joel: What are Web Services? In: *Communications of the ACM* 46 (2003), June, Nr. 6, S. 31

Fishwick u. a. 1998 Fishwick, P. A. (Hrsg.) ; Hill, D. R. C. (Hrsg.) ; Smith, R. (Hrsg.): *Web-Based Modeling and Simulation*. SCS, 1998

Forrester 1972 Forrester, J. W.: *Grundzüge einer Systemtheorie*. Wiesbaden : Gabler, 1972

Foster u. a. 2001 Foster, Ian ; Kesselman, Carl ; Tuecke, Steven: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In: *The International Journal of High Performance Computing Applications* 15 (2001), Nr. 3, S. 200–222

Freese und Seidel 1994 Freese, Henning ; Seidel, Jürgen: *Entwicklung einer graphischen, tabellenbasierten Simulationsmethodik zur Emissionsmodellierung*, Universität Hamburg, Fachbereich Informatik, Diplomarbeit, 1994

Fujimoto und Weatherly 1996 Fujimoto, Richard ; Weatherly, Richard M.: Time Management in the DoD High Level Architecture. In: (Louchs und Preiss 1996), S. 60–67

Fujimoto 2000 Fujimoto, Richard M.: *Parallel and Distributed Simulation Systems*. Wiley-Interscience Publication, 2000

Gamma u. a. 1996 Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster*. Addison-Wesley, 1996

Gehlsen und Page 2001 Gehlsen, Björn ; Page, Bernd: A Framework for Distributed Simulation Optimization. In: (Peters u. a. 2001), S. 508–514

Gouache und Priol 2000 Gouache, Stéphane ; Priol, Thierry: JACO3: A CORBA Software Infrastructure for Distributed Numerical Simulation. In: Engquist, B. (Hrsg.) ; Johnsson, L. (Hrsg.) ; Hammill, M. (Hrsg.) ; Short, F. (Hrsg.): *Simulation and Visualization on the Grid*. Paralleldatorcentrum Kungl Tekniska Högskolan Seventh Annual Conference, Stockholm, Sweden, December 1999 : Springer, 2000, S. 33–45. – Lecture Notes in Computational Science and Engineering, Vol. 13

Griffel 1998 Griffel, Frank: *Componentware: Konzepte und Techniken eines Softwareparadigmas.* dpunkt-Verlag, 1998

Griffel 2001 Griffel, Frank: *Verteilte Anwendungssysteme als Komposition klassifizierter Softwarebausteine: Ein Komponenten-basierter Ansatz zur Generativen Softwarekonstruktion*, Universität Hamburg, Fachbereich Informatik, Dissertation, 2001

Hamilton 1997 Hamilton, Graham: Java Beans™ API Specification. Sun Microsystems, August 1997. – Technical Document. – URL `http://java.sun.com/products/javabeans/docs/spec.html`.

Heim 1997 Heim, Joseph A.: Integrating Distributed Simulation Objects. In: (Andradóttir u. a. 1997), S. 532–538

Herren u. a. 1997 Herren, L. T. ; Fink, Pamela K. ; Moehle, Christoper J.: A User Interface to Support Experimental Design and Data Exploration of Complex, Deterministic Simulations. In: (Andradóttir u. a. 1997), S. 319–325

Herzog u. a. 2001 Herzog, Reinhard ; Mulder, John ; Pixius, Kay ; Menzler, Hans-Peter: HLA on top of CORBA Common Object Services. In: *European Simulation Interoperability Workshop.* London : University of Westminster, June 2001, S. 01E–SIW–009

Hilty u. a. 1998 Hilty, Lorenz M. u. a.: Instrumente für die ökologische Bewertung und Gestaltung von Verkehrs- und Logistiksystemen / Universität Hamburg, Fachbereich Informatik, ASI und Universität Ulm, Forschungsinstitut für anwendungsorientierte Wissensverarbeitung. 1998. – Abschlußbericht des Forschungsprojekts MOBILE.

MPI-Forum 1994 MPI-Forum: MPI: A Message-Passing Interface Standard. In: *International Journal of Supercomputer Applications* 8 (1994), Nr. 3-4, S. 165–414

Häuslein 1993 Häuslein, Andreas: *Wissensbasierte Unterstützung der Modellbildung und Simulation im Umweltbereich: Konzeption und prototypische Realisierung eines Simulationssystems.* Europäische Hochschulschriften, Reihe XLI, Informatik, Band 12; zugl. Dissertation, Universität Hamburg, Fachbereich Informatik. Peter Lang, 1993

Iazeolla und D'Ambrogio 1998 Iazeolla, G. ; D'Ambrogio, A.: A Web-based Environment for the Reuse of Simulation Models. In: *Western Multiconference on Computer Simulation.* San Diego, CA : SCS, January 1998, S. 37–42

IEEE 2000a IEEE: *1516-2000: Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules.* Institute of Electrical and Electronics Engineers, 2000

IEEE 2000b IEEE: *1516.1-2000: Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification.* Institute of Electrical and Electronics Engineers, 2000

IEEE 2000c IEEE: *1516.2-2000: Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Object Model Template (OMT) Specification*. Institute of Electrical and Electronics Engineers, 2000

ISI 1999 *Cosimulation Using Distributed Objects (CUDOs) Architecture*. pLUG&SIM^{TM} 11.6 Online Documentation. Integrated Systems, Inc., 1999. – URL `http://www.isi.com`. – 000-0173-001

Jefferson 1985 JEFFERSON, David R.: Virtual Time. In: *ACM Transactions on Programming Languages and Computer Systems* 7 (1985), July, Nr. 3, S. 404–425

Jessen und Valk 1987 JESSEN, Eike ; VALK, Rüdiger: *Rechensysteme: Grundlagen der Modellbildung*. Springer, 1987

Joines u. a. 2000 JOINES, J. A. (Hrsg.) ; BARTON, R. R. (Hrsg.) ; KANG, K. (Hrsg.) ; FISHWICK, P. A. (Hrsg.): *Winter Simulation Conference*. SCS, 2000

Kilgore und Healy 1998 KILGORE, R. ; HEALY, K.: Java, enterprise simulation and the Silk simulation language. In: (Fishwick u. a. 1998), S. 195–200

Kreger 2003 KREGER, Heather: Fulfilling the Web Services Promise. In: *Communications of the ACM* 46 (2003), June, Nr. 3, S. 29–34

Kriebisch 1996 KRIEBISCH, Ralf: *DYNAMIS III: Ein prototypisches Simulationssystem - Konzeption und Realisierung von Komponenten zur Modellerstellung und Experimentdurchführung*, Universität Hamburg, Fachbereich Informatik, Diplomarbeit, 1996

Kuljis und Paul 2000 KULJIS, Jasna ; PAUL, Ray J.: A Review of Web Based Simulation: Wither We Wander? In: (Joines u. a. 2000), S. 1872–1881

Lamersdorf 1994 LAMERSDORF, Winfried: *Datenbanken in verteilten Systemen: Konzepte, Lösungen, Standards*. Vieweg, 1994

Langer 1989 LANGER, Klaus-Jürgen: *Entwurf und Implementierung eines Simulationssystems mit integrierter Modellbank- und Experimentdatenverwaltung*, Universität Erlangen-Nürnberg, Institut für Mathematische Maschinen und Datenverarbeitung, Dissertation, 1989

Lantzsch u. a. 1999 LANTZSCH, Gunther ; STRASSBURGER, Steffen ; URBAN, Christoph: HLA-basierte Kopplung der Simulationssysteme SIMPLEX III und SLX. In: (Deussen u. a. 1999), S. 153–166

Lechler 1996 LECHLER, Tim: *Klassifikation von Simulationsmodellen im Bereich Verkehr und Logistik*. Studienarbeit, Universität Hamburg, Fachbereich Informatik. 1996

Lindholm und Yellin 1999 LINDHOLM, Tim ; YELLIN, Frank: *The Java Virtual Machine Specification*. Second Edition. Addison-Wesley, 1999

Liu und Nicol 2001 LIU, Jason ; NICOL, David: Learning not to share. In: BAGRODIA, Rajive L. (Hrsg.) ; DEELMAN, Ewa (Hrsg.) ; WILSEY, Philip A. (Hrsg.): *Parallel and Distributed Simulation*. Los Alamitos, CA : IEEE Computer Society Press, 2001, S. 46–55

Louchs und Preiss 1996 Louchs, Wayne M. (Hrsg.) ; Preiss, Bruno R. (Hrsg.): *Workshop on Parallel and Distributed Simulation*. Los Alamitos, CA : IEEE Computer Society Press, 1996

Löwy 2001 Löwy, Juval: *COM and .NET Component Services*. Second Edition. O'Reilly, 2001

Medeirors u. a. 1998 Medeirors, D. J. (Hrsg.) ; Watson, E. F. (Hrsg.) ; Carson, J. S. (Hrsg.) ; Manivannan, M. S. (Hrsg.): *Winter Simulation Conference*. SCS, 1998

Mehl 1994 Mehl, Horst: *Methoden verteilter Simulation*. Vieweg, 1994 (Programm angewandte Informatik)

Merz u. a. 1994 Merz, M. ; Müller, K. ; Lamersdorf, W.: Service Trading and Mediation in Distributed Computing Environments. In: Svobodova, L. (Hrsg.): *International Conference on Distributed Computing Systems*. Poznan, Poland : IEEE Computer Society Press, 1994, S. 440–457

Mügge und Meyer 1996 Mügge, Holger ; Meyer, Ruth: MSL - Eine Modell- und Experimentbeschreibungssprache. In: Ranze, C. (Hrsg.) ; Tuma, A. (Hrsg.) ; Hilty, L. M. (Hrsg.) ; Haasis, H.-D. (Hrsg.) ; Herzog, O. (Hrsg.): *Intelligente Methoden zur Verarbeitung von Umweltinformationen*. Marburg : Metropolis (Umweltinformatik aktuell; 10), 1996, S. 165–180

Miller u. a. 2001 Miller, John A. ; Fishwick, Paul A. ; Taylor, Simon J. ; Benjamin, Perakath ; Szymanski, Boleslaw: Research and Commercial Opportunities in Web-Based Simulation. In: *Simulation Practice and Theory* 9 (2001), October, Nr. 1-2, S. 55–72

Miller u. a. 2000a Miller, John A. ; Seila, Andrew F. ; Tao, Junxiu: Finding a Substrate for Federated Components on the Web. In: (Joines u. a. 2000), S. 1849–1854

Miller u. a. 2000b Miller, John A. ; Seila, Andrew F. ; Xiang, Xuewei: The JSIM Web-Based Simulation Environment. In: *Future Generation Computer Systems* 17 (2000), October, Nr. 2, S. 119–133

Möller 2000 Möller, Dietmar P. F. (Hrsg.): *14. Symposium Simulationstechnik*. Hamburg : SCS Europe, September 2000

Müller u. a. 1996 Müller, S. ; Müller-Jones, K. ; Lamersdorf, W. ; Tu, T.: Global Trader Cooperation in Open Service Markets. In: Spaniol, O. (Hrsg.) ; Meyer, B. (Hrsg.) ; Linnhoff-Popien, C. (Hrsg.): *Trends in distributed systems: CORBA and beyond*, Springer, October 1996, S. 214–227. – Lecture Notes in Computer Science, Issue 1161

Murphy und Aswegan 1998 Murphy, William S. ; Aswegan, Galen D.: High Level Architecture Remote Data Filtering. In: (Medeirors u. a. 1998), S. 835–840

Nicol 1988 Nicol, David M.: Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. In: Wexelblat, Richard L. (Hrsg.): *ACM/SIGPLAN conference on Parallel programming*. New York, NY : ACM Press, 1988, S. 124–137

Nicol u. a. 1997 Nicol, David M. ; Johnson, Michael M. ; Yoshimura, Ann S.: The IDES Framework: A Case Study in Development of a Parallel Discrete-Event Simulation System. In: (Andradóttir u. a. 1997), S. 93–99

Oestereich 1998 Oestereich, Bernd: *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language*. 4., aktualisierte Auflage. Oldenbourg, 1998

OMG 1999 OMG: The Common Object Request Broker: Architecture and Specification, Version 2.3.1. Object Management Group, October 1999. – Standard. formal/99-10-07

OMG 2000a OMG: Lisp Mapping Specification. Object Management Group, May 2000. – Standard. formal/00-06-02

OMG 2000b OMG: Notification Service Specification, Version 1.0. Object Management Group, June 2000. – Standard. formal/00-06-20

OMG 2000c OMG: Trading Object Service Specification, Version 1.0. Object Management Group, May 2000. – Standard. Manufacturing Domain: formal/00-06-27

OMG 2001a OMG: Event Service Specification, Version 1.1. Object Management Group, March 2001. – Standard. formal/01-03-01

OMG 2001b OMG: IDL to Java™ Language Mapping Specification, Version 1.1. Object Management Group, May 2001. – Standard. formal/01-05-03

OMG 2002a OMG: CORBA Components, Version 3.0. Object Management Group, June 2002. – Standard. formal/02-06-65

OMG 2002b OMG: Distributed Simulation Systems Specification, Version 2.0. Object Management Group, November 2002. – Standard. Manufacturing Domain: formal/02-11-11

OMG 2002c OMG: Naming Service Specification, Version 1.2. Object Management Group, September 2002. – Standard. formal/02-09-02

OMG 2002d OMG: Persistent State Service Specification, Version 2.0. Object Management Group, September 2002. – Standard. formal/02-09-06

OMG 2002e OMG: The Common Object Request Broker Architecture: Core Specification, Version 3.0.2. Object Management Group, December 2002. – Standard. formal/02-12-02

Orfali und Harkey 1998 Orfali, Robert ; Harkey, Dan: *Client/Server Programming with Java and CORBA*. Second Edition. John Wiley & Sons, 1998

Oses 2002 Oses, Noelia: *Component-based Simulation*, Lancaster University, UK, Department of Management Science, Dissertation, 2002

Oses u. a. 2002 Oses, Noelia ; Pidd, Michael ; Brooks, Roger J.: Component-based Simulation. In: *Component-based Simulation*. (Oses 2002), S. 248–259. – Appendix B: Paper of the 2002 Simulation Study Group Workshop

OSF 1995 Open Software Foundation: *Introduction to OSF DCE: Release 1.1.* Third Edition. Prentice Hall, 1995

Padur und Schreiber 2001 Padur, Volker ; Schreiber, Andreas: TENT - eine Integrations- und Testumgebung für Simulation und Visualisierung. In: Schulze, Thomas (Hrsg.) ; Schlechtweg, Stefan (Hrsg.) ; Hinz, Volker (Hrsg.): *Simulation und Visualisierung*, SCS Europe, März 2001, S. 305–314

Page 1991 Page, Bernd: *Diskrete Simulation: Eine Einführung mit Modula-2.* Springer, 1991

Page u. a. 2000 Page, Bernd ; Lechler, Tim ; Claassen, Sönke: *Objektorientierte Simulation in Java mit dem Framework Desmo-J.* Libri Books on Demand, 2000

Page u. a. 1998 Page, Ernest H. ; Buss, Arnold ; Fishwick, Paul A. ; Healy, Kevin ; Nance, Richard E. ; Paul, Ray J.: Web-Based Simulation: Revolution or Evolution? In: (Fishwick u. a. 1998), S. 3–17

Page u. a. 1997 Page, Ernest H. ; Moose, Robert L. ; Griffin, Sean P.: Web-based Simulation in Simjava Using Remote Method Invocation. In: (Andradóttir u. a. 1997), S. 468–474

Peters u. a. 2001 Peters, B. A. (Hrsg.) ; Smith, J. S. (Hrsg.) ; Medeiros, D. J. (Hrsg.) ; Rohrer, M. W. (Hrsg.): *Winter Simulation Conference.* SCS, 2001

Pidd und Castro 1998 Pidd, M. ; Castro, R. B.: Hierarchical Modular Modelling in Discrete Simulation. In: (Medeirors u. a. 1998), S. 383–390

Pitch 2003 Pitch: *Pitch Kunskapsutveckling AB, Nygatan 35, S-582 19 Linköping, Sweden.* http://www.pitch.de. März 2003

von Polier 2002 Polier von: *Persönliche Gespräche mit Mitarbeitern der Öffentlichkeitsabteilung im Rahmen einer Brauereibesichtigung.* Holsten-Brauerei AG, Holstenstrasse 224, 22765 Hamburg. April 2002

Praehofer u. a. 2000 Praehofer, H. ; Sametinger, J. ; Stritzinger, A.: Architektur eines Simulationsbaukastens basierend auf dem Komponentenmodell JavaBeans. In: *HMD-Praxis der Wirtschaftsinformatik* 212 (2000), April, Nr. 37, S. 99–111

Praehofer und Reisinger 2000 Praehofer, Herbert ; Reisinger, Gernot: Komponentenbasierte parallele Simulation unter objektorientierter Realisierung. In: Szczerbicka, Helena (Hrsg.) ; Uthmann, Thomas (Hrsg.): *Modellierung, Simulation und künstliche Intelligenz*, SCS Europe, 2000, S. 75–109

PrismTech 2002 *OpenFusion® CORBA Services: Notification Service Guide.* Version 3.0. Prism Technologies, June 2002. – URL `http://www.prismtechnologies.com`

Rabe 1999 Rabe, Markus: Beginnt ein neues Zeitalter der Simulation? In: (Deussen u. a. 1999), S. 3–18

Rechenberg 1972 Rechenberg, Peter: *Die Simulation kontinuierlicher Prozesse mit Digitalrechnern.* Vieweg, 1972

Reilly und Williams 2001 Reilly, Sean ; Williams, Howard: IDL4HLA: Implementing CORBA IDL Middleware for HLA. In: *Spring Simulation Interoperability Workshop.* Orlando, Florida : Institute for Simulation and Training, March 2001, S. 01S–SIW–065

Robinson und Pidd 1998 Robinson, S. ; Pidd, M.: Provider and customer expectations of successful simulation projects. In: *Journal of the Operational Research Society* 49 (1998), March, Nr. 3, S. 200–209

Sarjoughian 2002 Sarjoughian, Hessam S.: *The DEVSJAVA Simulation Environment.* 2002. – URL `http://www.acims.arizona.edu/SOFTWARE/software.shtml`. – Version 2.7, Juli 2002

Schöllhammer 2001 Schöllhammer, Thomas: *Eine offene Experimentierumgebung für verteilte Simulationsobjekte*, Universität Hamburg, Fachbereich Informatik, Diplomarbeit, 2001

Schmidt 2000 Schmidt, B.: *Einführung in die Simulation mit Simplex3.* SCS Europe, 2000

Schulze u. a. 2002 Schulze, Thomas ; Wytzisk, Andreas ; Simonis, Ingo ; Raape, Ulrich: Distributed Spatio-Temporal Modeling and Simulation. In: (Yücesan u. a. 2002), S. 695–703

Seila und Miller 1999 Seila, Andrew F. ; Miller, John A.: Scenario management in Web-based simulation. In: (Farrington u. a. 1999), S. 1430–1437

Sevinc 1991 Sevinc, Suleyman: Extending Common Lisp Object System for Discrete Event Modeling and Simulation. In: Nelson, Barry L. (Hrsg.) ; Kelton, W. D. (Hrsg.) ; Clark, Gordon M. (Hrsg.): *Winter Simulation Conference*, IEEE Computer Society, 1991, S. 204–206

Shen 2000 Shen, Chien-Chung: Discrete-Event Simulation on the Internet and the Web. In: *Future Generation Computer Systems* 17 (2000), Nr. 2, S. 187–196

Shneiderman 1998 Shneiderman, Ben: *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* 3rd. ed. Addison-Wesley, 1998

Siegel 2000 Siegel, Jon: *CORBA 3 fundamentals and programming.* Wiley, 2000

Straßburger und Schulze 2001 Strassburger, S. ; Schulze, T.: Verteilte- und Web-basierte Simulation: Gemeinsamkeiten und Unterschiede. In: Panreck, K. (Hrsg.) ; Dörrscheidt, F. (Hrsg.): *15. Symposium Simulationstechnik.* Paderborn : SCS Europe, September 2001, S. 449–454

Straßburger 2001 Strassburger, Steffen: *Distributed Simulation Based on the High Level Architecture in Civilian Application Domains*, Universität Magdeburg, Fachbereich Informatik, Dissertation, 2001

Szyperski 2002 Szyperski, Clemens: *Component Software: Beyond Object-Oriented Programming*. Second Edition. Addison-Wesley, 2002

Taylor 2000 Taylor, Simon J. E.: Groupware and the Simulation Consultant. In: (Joines u. a. 2000), S. 83–89

Thai und Lam 2002 Thai, Thuan ; Lam, Hoang Q.: *.NET Framework Essentials*. Second Edition. O'Reilly, 2002

Tittel u. a. 1996 Tittel, Ed ; Gaither, Mark ; Hassinger, Sebastian ; Erwin, Mike: *WWW-Programmierung mit CGI: Entwicklung interaktiver Anwendungen*. International Thomson Publishing, 1996

Tolk 2000 Tolk, Andreas: HLA-OMT versus Traditional Data and Object Modeling — Chance or Shoehorn? In: *Simulation Interoperability Workshop*. Orlando, Florida : Institute for Simulation and Training, March 2000, S. 00S–SIW–024

Tolk 2002 Tolk, Andreas: Avoiding another Green Elephant — A Proposal for the Next Generation HLA based on the Model Driven Architecture. In: *Simulation Interoperability Workshop*. Orlando, Florida : Institute for Simulation and Training, September 2002, S. 02F–SIW–004

Tu 2000 Tu, Mark Tuan V.: *Steuerung offener verteilter Anwendungssysteme: Konstruktion interaktionsorientierter Regelverarbeitungs- und Verhandlungsmechanismen — dargestellt am Beispiel elektronischer Dienstemärkte*, Universität Hamburg, Fachbereich Informatik, Dissertation, 2000

Ulrikson u. a. 2001 Ulrikson, Jenny ; Moradi, Farshad ; Svensson, Olof: A Web-based Environment for building Distributed Simulations. In: *Simulation Interoperability Workshop*. London : University of Westminster, June 2001, S. 02E–SIW–036

Vogel und Duddy 1998 Vogel, Andreas ; Duddy, Keith: *Java Programming with CORBA*. Second Edition. John Wiley & Sons, 1998

Wenzel 2000 Wenzel, Sigrid: Referenzmodelle für die Simulation in Produktion und Logistik. In: (Möller 2000), S. 1–10

Wittmann 1992 Wittmann, Jochen: Model and Experiment — A New Approach to Definitions. In: Luker, P. (Hrsg.): *Summer Computer Simulation Conference*. Reno, Nevada : SCS, July 1992, S. 115–119

Wittmann 1993 Wittmann, Jochen: *Eine Benutzerschnitstelle für die Durchführung von Simulationsexperimenten: Entwurf und Implementierung der Experimentierungebung für das Simulationssystem SIMPLEX II*, Universität Erlangen-Nürnberg, Technische Fakultät, Dissertation, 1993

Yücesan u. a. 2002 Yücesan, E. (Hrsg.) ; Chen, C.-H. (Hrsg.) ; Snowdon, J. L. (Hrsg.) ; Charnes, J. M. (Hrsg.): *Winter Simulation Conference*. SCS, 2002

Zeigler 1976 Zeigler, Bernard P.: *Theory of Modeling and Simulation*. Wiley Interscience, 1976

Zeigler 1984 Zeigler, Bernard P.: *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, 1984

Zeigler u. a. 1999a Zeigler, Bernard P. ; Ball, George ; Cho, Hyup ; Lee, J.S. ; Sarjoughian, Hessam: Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions. In: *Spring Simulation Interoperability Workshop.* Orlando, Florida : Institute for Simulation and Training, March 1999, S. 158–160

Zeigler und Kim 1996 Zeigler, Bernard P. ; Kim, Doohwan: Design of High Level Modelling / High Performance Simulation Environments. In: (Louchs und Preiss 1996), S. 154–161

Zeigler u. a. 1999b Zeigler, Bernard P. ; Kim, Doohwan ; Buckley, Stephen J.: Distributed Supply Chain Simulation in a DEVS/CORBA Execution Environment. In: (Farrington u. a. 1999), S. 1333–1340

Zeigler u. a. 2000 Zeigler, Bernard P. ; Praehofer, Herbert ; Kim, Tag G.: *Theory of Modeling and Simulation.* Second Edition. Academic Press, 2000

Zeigler und Sarjoughian 2000 Zeigler, Bernard P. ; Sarjoughian, Hessam S.: Creating Distributed Simulation Using DEVS M&S Environments. In: (Joines u. a. 2000), S. 158–160