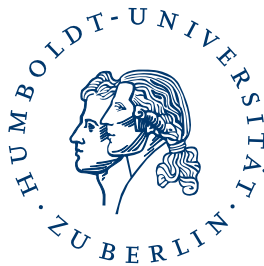


Revision der Simulationsbibliothek ODEmx

Integration eines generischen Logging-Konzeptes unter
Einbeziehung eines relationalen Datenbanksystems und
Erweiterung des Protokolls simulationsmoduls

Diplomarbeit
Ronald Kluth



HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT II
INSTITUT FÜR INFORMATIK

Gutachter
Prof. Dr. Joachim Fischer
Prof. Dr. Jens-Peter Redlich

Betreuer
Dr. Klaus Ahrens

Berlin, den 5. Mai 2010

Inhaltsverzeichnis

1	Einleitung	1
1.1	Problemstellung	2
1.2	Zielsetzung	3
1.3	Vorgehensweise	4
2	Grundlagen	7
2.1	Die Simulationsbibliothek ODEMx	7
2.1.1	Struktur und Funktionalität	7
2.1.2	Erweiterungen und Restrukturierung	9
2.1.3	Konzepte zur Datenerfassung	12
2.2	Datenbanken	13
2.2.1	Begriffsklärung	14
2.2.2	Datenmodellierung	15
2.2.3	Normalisierung	17
2.2.4	Generelle Vorteile der Datenbanknutzung	19
3	Logging	21
3.1	Einführung	21
3.1.1	Begriffsklärung	21
3.1.2	Verwendung in der Computersimulation	21
3.1.3	Konzepte aktueller Logging-Frameworks	22
3.2	Ein generisches Logging-Konzept	25
3.2.1	Analyse des Trace-Mechanismus von ODEMx	25
3.2.2	Zielsetzungen	27
3.2.3	Rollen und Zusammenspiel aktiver Komponenten	30
3.2.4	Klassen und Assoziationen	31
3.3	Softwaretechnische Umsetzung	33
3.3.1	Essentielle Werkzeuge	33
3.3.2	Nutzungsebenen	36

3.3.3	Kanäle, Konsumenten und Filter	38
3.3.4	Identifizierbare Produzenten und Verwaltung von Kanälen .	41
3.3.5	Ein Log-Datentyp zum Transport beliebiger Daten	45
3.3.6	Filtern von Log-Daten	49
3.3.7	Textbasierte Ausgabekomponenten	51
3.3.8	Anbindung an relationale Datenbanken	54
3.4	Integration der Logging-Bibliothek in ODEMX	64
3.4.1	Der Log-Datentyp SimRecord	65
3.4.2	Erzeugung und Verteilung von Log-Daten	66
3.4.3	Filtern von Log-Daten	71
3.4.4	Verarbeitung und textbasierte Darstellung von Log-Daten .	74
3.4.5	Anbindung an relationale Datenbanken	78
3.4.6	Statistikerfassung und Berichterzeugung	88
3.4.7	Laufzeitvergleich des bisherigen Trace-Mechanismus und des Loggings via Kanal trace	93
3.5	Zusammenfassung	95
4	Protokollsimulation	99
4.1	Einführung	99
4.1.1	Begriffsklärung	99
4.1.2	Modellierung von Kommunikationsprotokollen	99
4.1.3	Simulation von Kommunikationsprotokollen	102
4.2	Das ODEMX-Protokollsimulationsmodul	104
4.2.1	Konzept	104
4.2.2	Softwaretechnische Umsetzung in Version 2.2 von ODEMX .	105
4.3	Ansatz zur Erweiterung des Protokollsimulationsmoduls	108
4.3.1	Problemanalyse der bisherigen Implementation	109
4.3.2	Konzept für die Erweiterung der Implementation	114
4.4	Softwaretechnische Umsetzung des erweiterten Protokollsimula- tionsmoduls	117
4.4.1	Protokolldateneinheiten	117
4.4.2	Dienstzugangspunkte und Diensterbringer	118
4.4.3	Kommunikationsdienste und Fehlermodellierung	119
4.4.4	Beispiel XCS 1 - Ein Einfacher Kommunikationsdienst	121
4.4.5	Protokollstacks mit Schichten und Instanzen	125
4.4.6	Beispiel XCS 2 - Zweischichtiger Kommunikationsdienst . .	126

4.4.7	Netzwerkschnittstellen und Übertragungsmedium	130
4.4.8	Beispiel XCS 3 - Kommunikation via Übertragungsmedium .	133
4.4.9	Speicherverwaltung	136
4.5	Zusammenfassung	137
5	Resultate und Ausblick	139
5.1	Der kommende C++-Standard (C++0x)	140
A	Konfigurationsmöglichkeiten der Bibliotheken	143
B	XML-Schema-Definitionen	145
C	Simulationsprogramm zum Laufzeittest	147
D	Ausgaben der Beispielprogramme	149
	Literaturverzeichnis	155

1 Einleitung

Computersimulation ist heutzutage ein vielfach eingesetztes Hilfsmittel, um das Verhalten realer und fiktiver Systeme experimentell zu erforschen. Dafür muss ein Untersuchungsziel festgelegt und ein angemessenes Modell des Systems entwickelt werden, an dem unter Variation von Parametern eine Vielzahl von Experimenten durchgeführt werden kann. Am Institut für Informatik der Humboldt-Universität zu Berlin geschieht dies zum Beispiel im Rahmen des Projekts SAFER [SAFER], bei dem aus einem in der Spezifikationssprache SDL-RT formulierten Alarmierungsprotokoll zur Erdbebenfrühwarnung C++-Code für die Simulation generiert wird, um das Verhalten großer Sensornetzwerke zu erforschen. Ein weiteres Beispiel ist das Projekt SimRing, in dem simulativ Verbesserungsmöglichkeiten im Produktionsablauf eines Stahlwerks untersucht werden [Eves06], wobei Erkenntnisse aus computergestützten Experimenten bereits im Betrieb umgesetzt worden sind.

Die Erforschung eines Untersuchungsziels an einem Modell erfordert meist eine große Anzahl von Experimenten mit verschiedenen Parameterwerten, die zusammen mit den Simulationsdaten für eine spätere Analyse gespeichert werden müssen, wobei abhängig vom Detailgrad der Datenerfassung, der Komplexität des Simulators und der Simulationsdauer schon bei einzelnen Simulationsläufen gigantische Datenmengen entstehen können. Um den Überblick über Simulationsexperimente, Parameterbelegungen und Ergebnisse zu behalten, wird derzeit am Lehrstuhl für Systemanalyse ein Experimentmanagement-System (EMS) mit Datenbankbindung verwendet [Fischer08]. Es ist dabei von Vorteil, dass Datenbankmanagementsysteme generell Anfragesprachen unterstützen, mit denen spezifische Informationen extrahiert werden können. Der Zugriff auf die gesammelten Daten ist dadurch mit verschiedenen Analysewerkzeugen möglich, solange diese mit dem verwendeten Datenbankmanagementsystem interagieren können.

Zur Unterstützung der Modellierung und Simulation wird in vielen Fällen auf spezielle Simulationswerkzeuge oder auch Simulationsframeworks zurückgegriffen.

Letztere liegen oft als Softwarebibliotheken in einer bestimmten Programmiersprache vor und stellen häufig benötigte Modell- und Simulationsbausteine bereit. Die vorliegende Arbeit befasst sich mit der Überarbeitung von Komponenten einer solchen Bibliothek.

1.1 Problemstellung

Am Institut für Informatik beschäftigt sich der Lehrstuhl Systemanalyse bereits seit vielen Jahren mit der objektorientierten Ereignis- und Prozesssimulation in C++. Als Resultat dieser Arbeit entstand im Jahr 1993 die Klassenbibliothek ODEM, deren Anwendungsgebiete sowohl in der Lehre als auch der Forschung lagen [Fischer96]. Im Rahmen einer Diplomarbeit erfolgte im Jahr 2003 eine konzeptuelle und codetechnische Modernisierung der Bibliothek unter dem Namen ODEMX, die neben vielen weiteren Neuerungen auch drei verschiedene Konzepte zur Datenerfassung einführte [Gerst03].

Wie die Vorgängerbibliothek wird ODEMX sowohl in der Lehre als auch in aktuellen Forschungsprojekten als Simulationswerkzeug verwendet. Allerdings zeigten sich Schwächen im Bereich der Datenerfassung, was zum einen die teilweise komplizierte Verwendung dieser Komponenten betrifft und zum anderen ihre Implementation. Das führte letztendlich dazu, dass viele Entwickler ihre eigene Datenerfassung implementierten. Zudem offenbarte die Einführung eines datenbankbasierten Experimentmanagement-Systems eine Lücke zwischen Simulationsexperimenten und ihrer Verwaltung und Auswertung, da ODEMX keine direkte Datenbankbindung ermöglicht. Diese Gründe erfordern eine Untersuchung und Bewertung der vorhandenen Datenerfassungsmechanismen sowie die Integration neuer Komponenten.

In den Jahren 2006-2009 wurden an der Bibliothek vom Autor der Arbeit eine Reihe von Erweiterungen vorgenommen, wozu auch die Einführung eines speziellen Moduls zur Simulation von Kommunikationsprotokollen gehört. Während der Anwendung dieser Komponenten wurden allerdings verschiedene Unzulänglichkeiten im zugrunde liegenden Konzept des Moduls identifiziert, welche die Nachbildung realer Protokolle erschweren. Deshalb befasst sich der zweite Teil der Arbeit mit der Überarbeitung und Erweiterung des Protokollsimulationsmoduls.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Analyse und Korrektur bekannter Schwachstellen von ODEMX. Im Bereich der Datenerfassung beinhaltet dieses Vorhaben eine Überarbeitung bestehender Mechanismen sowie die Implementation einer Anbindung an relationale Datenbanken. Da die Protokollierung des Programmablaufs (Logging) ein wichtiger Bestandteil von Software im Allgemeinen ist, soll auch eine separate Nutzung der Datenerfassungskomponenten ermöglicht werden. Als Konsequenz wird das Thema Logging gesondert untersucht und im Rahmen der Arbeit eine eigenständige kompakte Bibliothek entwickelt, die anschließend als Grundlage der Datenerfassung in ODEMX integriert wird.

Der Bereich der Protokollsimulation erfordert eine konzeptuelle Erweiterung des Protokollmoduls und die Implementation des neuen Konzepts. Die Entwicklung zur Protokollsimulation mit ODEMX wurde angestoßen, um eine Möglichkeit für vergleichende Untersuchungen mit anderen Netzwerksimulatoren zu schaffen. Die existierende Implementation orientiert sich zwar am intuitiven OSI-Schichtenmodell, enthält aber verschiedene Annahmen und Einschränkungen, die im Rahmen der Arbeit zu analysieren und zu beheben sind. In diesem Zusammenhang ist auch die Zugänglichkeit des Moduls von Bedeutung, da existierende Simulationswerkzeuge auf diesem Gebiet einen nicht unerheblichen Einarbeitungsaufwand verlangen, selbst wenn nur Basisfunktionalität genutzt werden soll. Um diesem Problem entgegen zu wirken, soll das neue Konzept mehrere Abstraktionsgrade unterstützen.

Vielseitigkeit und Wiederverwendbarkeit von Komponenten sind wichtige Eigenschaften bei der Entwicklung von Softwarebibliotheken. Generische Programmierung ist ein Verfahren, das diese beiden Ansprüche erfüllen kann, indem Datenstrukturen oder Algorithmen möglichst allgemein formuliert werden, um durch Parametrisierung ihre Verwendung mit unterschiedlichen Datentypen zu erlauben. Die für ODEMX verwendete Programmiersprache C++ unterstützt generische Programmierung durch sogenannte Templates, die auch im Rahmen der Arbeit ihre Anwendung finden. Weitere ausgiebig genutzte Werkzeuge sind in diesem Zusammenhang die C++-Standardbibliothek, welche generische Klassen und Algorithmen bereitstellt, sowie die mit dem Technical Report 1 (TR1) hinzugekommenen Erweiterungen der Standardbibliothek aus dem Jahr 2005 [Becker06].

Als Ergebnis der Arbeit entsteht Version 3.0 der Simulationsbibliothek ODEMX, die in den Bereichen Datenerfassung und Protokollsimulation eine grundlegende Erneuerung erfährt. In diesem Rahmen wird die Datenerfassung um eine generische Bibliothek erweitert, auf deren Grundlage unter anderem eine Datenbank-anbindung zu ODEMX hinzugefügt wird. Das Protokollsimulationsmodul wird in Version 3.0 mehrere Abstraktionsgrade ermöglichen und mehr Modellierungsfreiheiten bieten, so dass auch komplexere Protokoll- und Netzwerkmodelle damit implementiert werden können.

1.3 Vorgehensweise

In Kapitel 2 wird zunächst das notwendige Grundlagenwissen behandelt. Da die Revision der Simulationsbibliothek ODEMX das übergreifende Thema der vorliegenden Arbeit ist, erfolgt als erstes eine kurze Vorstellung der Bibliothek mit Fokus auf ihrer Grundfunktionalität, dem Entwicklungsfortschritt der letzten Jahre und den bisher vorhandenen Datenerfassungsmechanismen. Der darauf folgende Abschnitt gibt dann eine kurze Einführung zum Thema Datenbanken, weil sowohl die Logging-Bibliothek als auch ODEMX diesen Aspekt in der Umsetzung der Datenerfassung unterstützen sollen. Die daran anschließenden Kapitel der Arbeit sind entsprechend der Problemstellung den beiden Schwerpunkt-Themen gewidmet.

Kapitel 3 befasst sich einerseits allgemein mit dem Thema Logging und andererseits mit seiner Integration in ODEMX. Auf dem Gebiet des Loggings gibt es für C++ und ähnliche imperative Programmiersprachen wie Java verschiedene als Softwarebibliotheken verfügbare Implementationen von Logging-Mechanismen, anhand derer häufig genutzte Konzepte zur Ablaufprotokollierung und Datenerfassung vorgestellt und mit Hinblick auf die Zielstellung der Arbeit bewertet werden sollen. Weiterhin erfolgt eine Analyse der bisherigen Datenerfassungskonzepte von ODEMX und die Bewertung ihrer Eignung für eine Datenbank-anbindung. Der Hauptteil dieses Themengebietes befasst sich mit der Vorstellung eines generischen Logging-Konzeptes und seiner Implementation sowie der Integration in ODEMX. Abgeschlossen wird der Bereich mit Laufzeitvergleichstests zwischen Version 2.2 und Version 3.0 von ODEMX.

Kapitel 4 behandelt die Erweiterung des ODEMX-Protokollsimulationsmoduls. Das Themengebiet wird eingeführt mit einem Grundlagenkapitel zur Protokollmodel-

lierung, gefolgt von einer Einführung in das Modellierungskonzept des existierenden Protokollsimulationsmoduls. Daran anschließend werden die Defizite des Moduls erörtert und Lösungsmöglichkeiten aufgezeigt. Im Sinne der oben angesprochenen Zugänglichkeit des Moduls findet auch eine konzeptuelle Erweiterung mit verschiedenen Abstraktionsstufen statt. Danach wird die softwaretechnische Umsetzung dieser Konzepte dargelegt und anhand eines begleitenden Beispiels erklärt.

Den Abschluss der Arbeit bildet Kapitel 5 mit einer Zusammenfassung der Ergebnisse sowie einem Ausblick auf mögliche weitere Verbesserungen an der Simulationsbibliothek ODEmx.

Notation und Konventionen

Im Laufe der Arbeit werden eine Reihe von Definitionen benötigt. Der jeweils definierte Begriff wird im **Fettdruck** dargestellt. Bezeichner von Klassen und Methoden, die sich auf bestimmte Implementationsdetails beziehen, werden durch Schreibmaschinenschrift markiert. Eigennamen und Bezeichner sonstiger Artefakte sind durch serifenlose Schrift gekennzeichnet. An verschiedenen Stellen werden deutsche Begriffe verwendet, deren englische Bezeichnung daneben in Klammern aufgeführt ist. Diese ist dann durch *Kursivschrift* abgesetzt.

Implementationsdetails, die zum Verständnis bestimmter Abschnitte benötigt werden, sind in rechteckigen Boxen mit weißem Hintergrund beschrieben. Zur Veranschaulichung von Konzepten und deren Implementation werden meist Code-Fragmente als Beispiele herangezogen. Sind diese in abgerundeten Boxen mit weißem Hintergrund präsentiert, so handelt es sich um Code, der bereits durch eine Bibliothek bereitgestellt wird. Quellcode in grau unterlegten Rechtecken stellt dagegen Beispiele dar, in denen die Verwendung verschiedener Komponenten verdeutlicht wird.

2 Grundlagen

2.1 Die Simulationsbibliothek ODEMx

Im Abschnitt 1.1 wurde bereits erwähnt, dass ODEMx auf Konzepten der Vorgängerbibliothek ODEM aufbaut, welche ausführlich in [Fischer96] beschrieben wird. Die theoretische Basis von ODEM und ODEMx ist der von Bernard Zeigler definierte DEVS-Formalismus (*Discrete Event System Specification*) [Zeigler00]. Praxisorientierte Einführungen in das Thema der darauf aufbauenden diskreten Ereignissimulation findet der geneigte Leser in [Fischer96] und in [Kluth07].

Eine Darstellung der Entwicklungsgeschichte beider Bibliotheken findet sich in [Kluth07], weshalb sich dieses Kapitel auf eine Vorstellung der Funktionalität und Weiterentwicklung der Simulationsbibliothek ODEMx beschränkt. Der nächste Abschnitt gibt dazu eine Übersicht der ersten ODEMx-Version. Die seither vorgenommenen Änderungen und Erweiterungen werden im Anschluss daran erläutert.

2.1.1 Struktur und Funktionalität

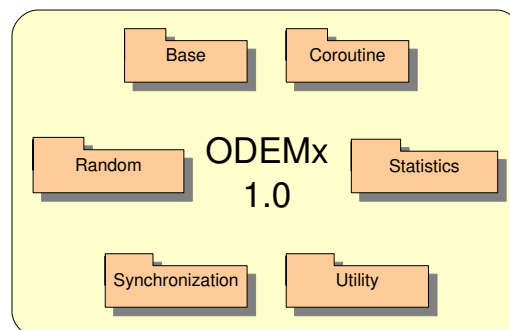


Abbildung 2.1: Modularer Aufbau der Bibliothek ODEMx

Die erste Version von ODEMx wurde durch die in Abbildung 2.1 dargestellten Module strukturiert. Im Modul Base sind die grundlegenden Bestandteile für die prozessorientierte diskrete Ereignissimulation gebündelt: Ereigniskalender, Simulationssteuerung, Klassen für zeitdiskrete und zeitkontinuierliche Prozesse sowie HTML-Ausgabeklassen und ein Ausgabefilter.

Coroutine enthält die Implementation eines Koroutinenkonzepts für die quasi-parallele Ausführung von Prozessen. Alle Koroutinen werden einem Koroutinenkontext zugeordnet, der die Kapselung mehrerer Simulationskontexte in einem Programm ermöglicht. ODEMx-Koroutinen unterstützen sowohl Win32- als auch POSIX-basierte Zielplattformen, womit die Klassen des Moduls eine Abstraktionsschicht formen, die systemspezifische Implementationsdetails vor anderen ODEMx-Klassen verbirgt.

Im Modul Random sind deterministische Zufallszahlengeneratoren implementiert, mit deren Hilfe sich für die statistische Simulation verschiedene ganzzahlige und reelle Wahrscheinlichkeitsverteilungen wie zum Beispiel eine Poisson-Verteilung, Gleichverteilung oder Normalverteilung modellieren lassen. Dass die erzeugten Zahlenfolgen bei gleichen Ausgangsbedingungen immer gleich sind, ist wichtig für die Reproduzierbarkeit von Simulationsergebnissen.

Das Modul Statistics umfasst Klassen zur Berechnung statistischer Kennwerte wie Minimum, Maximum, Mittelwert und Standardabweichung. Hier finden sich verschiedene Werkzeuge vom einfachen Zähler über akkumulierende Komponenten bis hin zur linearen Korrelationsanalyse. Auch Histogramme werden unterstützt.

Synchronization stellt verschiedene Mechanismen zur Synchronisation bereit, unter anderem Warteschlangen für die Master-Slave-Synchronisation oder bedingtes Warten, die auch statistische Daten sammeln. Weiterhin sind Klassen zur Modellierung von begrenzten und unbegrenzten Ressourcen enthalten.

Den Abschluss bildet das Modul Utility, welches verschiedene Hilfsklassen zur Datensammlung, Ausgabe und Fehlerbehandlung enthält. Detaillierte Beschreibungen der zugrundeliegenden Konzepte für die in diesem Abschnitt aufgeführten Komponenten finden sich in [Gerst03].

2.1.2 Erweiterungen und Restrukturierung

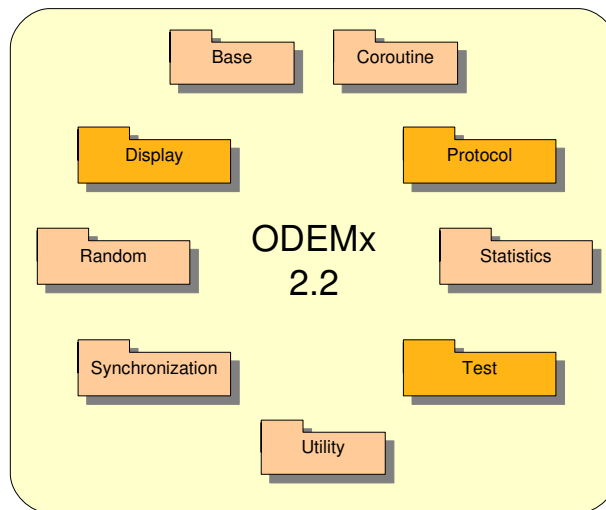


Abbildung 2.2: Aufbau der Bibliothek ODEMx, Version 2.2

Abbildung 2.2 zeigt die Modulstruktur der aktuell in Forschung und Lehre verwendeten Version 2.2 von ODEMx. Die Markierung verdeutlicht, dass zu der Bibliothek drei weitere Module hinzugekommen sind. Aber auch die Funktionalität existierender Module wurde im Laufe der Jahre in verschiedenen Punkten erweitert.

Zur Bündelung der Ausgabekomponenten von ODEMx entstand das Modul Display, dem zunächst die HTML-Ausgabeklassen und Ausgabefilter zugeordnet wurden. Hinzu kamen Klassen für die Formatierung der Simulationszeit im offiziellen deutschen Zeitformat und im ISO 8601-Format. Die starre HTML-Ausgabe wurde ergänzt durch Klassen zur XML-Ausgabe, die in Verbindung mit Javascript, XSL-Transformation und Cascading Style Sheets eine dynamische Darstellung von Simulationsabläufen in modernen Web-Browsern ermöglichen. Dadurch können die aufgezeichneten Zustandsänderungen anhand verschiedener Eigenschaften interaktiv gefiltert werden. Auch das Problem monolithischer Dateien wurde bei der XML-Ausgabe behandelt, indem automatisch eine Verteilung auf mehrere Dateien erfolgt.

Mit dem Modul Protocol kamen erste Bausteine für die Simulation von Kommunikationsprotokollen hinzu. Eine Analyse und Überarbeitung dieser Komponenten erfolgt in Kapitel 4, weshalb hier nicht näher darauf eingegangen wird.

Da Änderungen in Code und Struktur einer Software auch Fehler an unerwarteten Stellen hervorrufen können, ist die Inklusion von Testfällen wichtig für die regelmäßige Durchführung automatischer Regressionstests, mit denen die erwartete Funktionsweise aller Komponenten sichergestellt wird. Als Konsequenz wurden mit dem Modul Test Komponententests für alle in ODEMx enthaltenen Klassen ergänzt. Die Grundlage dafür bildet die in ODEMx integrierte quelloffene Bibliothek `UnitTest++` [UTPP].

Die Erweiterung bestehender Module erfolgte im Rahmen einer Studienarbeit [Kluth07]. So wurden einige noch fehlende Konzepte und Komponenten aus der Vorgängerbibliothek ODEM nun in ODEMx reimplementiert. Das im Modul Base bereitgestellte prozessorientierte Simulationskonzept wurde dabei um ein ereignisorientiertes Konzept ergänzt, so dass sich neben prozessbasierten auch komplett ereignisbasierte oder gemischte Simulationen unter Verwendung von Ereignis- und Prozessklassen implementieren lassen. Weiterhin wurde ein Warte- und Alarmierungskonzept für Prozesse implementiert, dessen grundlegende Komponenten Memory, Port und Timer dem Modul Synchronization zugeordnet sind. Die bereitgestellten Ressourcenbausteine wurden um die Klassentemplates `BinT` und `ResT` erweitert, welche durch Typparametrisierung die Verwendung nutzerdefinierter Tokentypen ermöglichen.

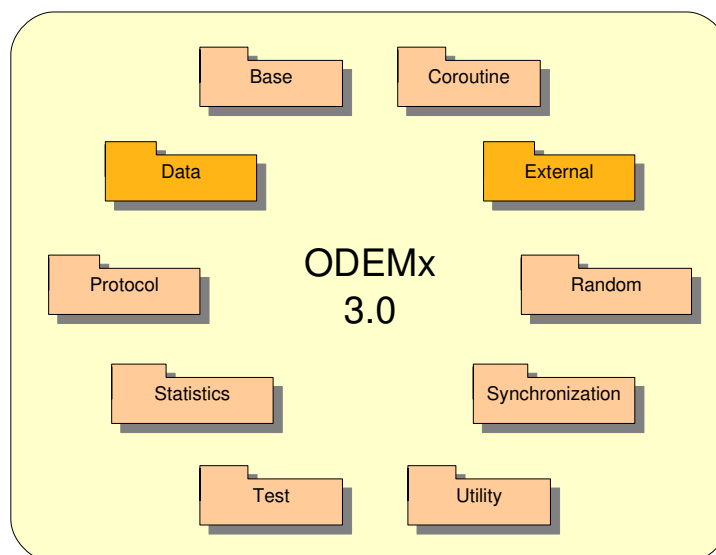


Abbildung 2.3: Aufbau der Bibliothek ODEMx, Version 3.0

Abbildung 2.3 zeigt die Modulstruktur der aus dieser Arbeit hervorgehenden Version 3.0 von ODEMx. Das Modul Data enthält alle in Abschnitt 3.4 beschriebenen Datenerfassungskomponenten und vereint die Klassen des bisherigen Display-Moduls mit neuen Ausgabekomponenten und Puffern zum Zwischenspeichern von Simulationsdaten. Mit External kommt ein weiteres Modul zur Integration externer Bibliotheken hinzu, die ODEMx für seine Funktionalität benötigt. Neben den hier bearbeiteten Themen der Datenerfassung und der Protokollsimulation wurden noch andere nennenswerte Änderungen vorgenommen.

Im Rahmen der Studienarbeit [Kluth07] hat sich herausgestellt, dass die Implementation des Schedulingmechanismus sehr komplex und nur schwer nachvollziehbar war. Aus diesem Grund wurde dieser Teil aus der Klasse Simulation extrahiert und in der separaten Klasse Scheduler umgesetzt, die ebenfalls dem Modul Base zugeordnet ist. Eine Vereinfachung der Implementation wurde außerdem dadurch erreicht, dass die Programmausführung nach jedem abgearbeiteten Eintrag des Ereigniskalenders zum Scheduler zurückkehrt, unabhängig davon, ob zuletzt ein einfaches Ereignis oder ein Prozess mit eigenem Laufzeitstack aktiviert wurde. Ein Nebeneffekt davon ist, dass man sich an dieser Stelle sicher sein kann, nicht auf dem Laufzeitstack eines Prozesses zu arbeiten, wodurch gefahrlos die Löschung terminierter Prozessobjekte während des Simulationslaufes implementiert werden konnte. Die Verwendung dieser Funktionalität ist optional und muss durch Aufruf einer Methode des Simulationskontextes aktiviert werden. Die Speicherfreigabe terminierter Prozessobjekte ist notwendig, wenn die Erzeugung vieler Prozessobjekte und der dazugehörigen Laufzeitstacks zu Speicherengpässen führt.

Die Koroutinenimplementation wurde durch Ahrens verbessert, indem plattformübergreifend die von Win32 bekannte Fiber-Schnittstelle genutzt wird. Auf POSIX-basierten Plattformen wird mit der Klasse ucFiber eine Abstraktion des sogenannten Nutzerkontextes ucontext und zugehöriger Funktionen verwendet. Der Vorteil dieser Variante ist, dass jede Koroutine einen eigenen Laufzeitstack erhält, wodurch der aktive Laufzeitstack des Simulationsprogramms nun nicht mehr durch ODEMx kopiert und ausgetauscht werden muss. Gerade letzteres hat manchmal subtile Fehler verursacht, weil Zeiger auf Stackadressen verwiesen, die nicht den erwarteten Inhalt hatten.

2.1.3 Konzepte zur Datenerfassung

An dieser Stelle erfolgt zunächst eine kurze Zusammenfassung zu den verschiedenen Datenerfassungsmechanismen von ODEMX, begleitet von einer Bewertung des jeweiligen Konzepts bezüglich der Aufzeichnung aller Zustandsänderungen zur Speicherung in einer relationalen Datenbank.

Report

Das **Report**-Konzept ermöglicht die Erstellung zusammenfassender Statistikberichte für bestimmte Simulationskomponenten, was vor allem bei Warteschlangen, Ressourcenbausteinen und Zufallszahlengeneratoren Anwendung findet. Einem Report-Objekt werden dabei beliebig viele Reportproduzenten zugeordnet, die auf Anforderung ihre gesammelten Daten in Form einer oder mehrerer Tabellen an das Report-Objekt weitergeben.

Das Report-Konzept arbeitet mit aggregierten Daten, die während der Laufzeit eines Simulationsprogramms von den einzelnen Objekten berechnet werden. Durch den auf Anfrage erzeugten Bericht gibt es jedoch keinen Zugriff auf spezifische Daten, die einzelne Zustandsänderungen des Programms zu einem bestimmten Zeitpunkt beschreiben. Außerdem ist das Konzept auf bestimmte Komponententypen begrenzt, womit es für eine Datenbankbindung, die alle Zustandsänderungen erfassen können soll, nicht in Frage kommt.

Observation

Das **Observation**-Konzept bezieht sich auf einzelne Objekte und die Beobachtung ihres Verhaltens. Es basiert auf der Kopplung von Objekten durch vordefinierte Beobachterschnittstellen, die einheitlichen Konventionen bezüglich Attributsänderungen und Simulationsereignissen folgen. Zwar lassen sich damit auch Interaktionen zwischen Modellelementen modellieren, der hauptsächliche Anwendungsbereich ist jedoch die Erfassung und Auswertung von Simulationsdaten. Objekte können dabei zwei verschiedene Rollen einnehmen: die des *Observer*, der ein Objekt überwacht und die des *Observable*, eines Objekts, das überwacht werden kann. Objekte der zweiten Gruppe, deren Zustandsänderungen beobachtbar sein sollen, definieren dafür eine Observer-Schnittstelle, welche zur Datenerfassung verwendet werden kann. Sie müssen zudem vom Klassentemplate *Observable* abgeleitet

sein, wobei als Parameter ihre Schnittstellendefinition angegeben wird. Beispielsweise definiert die Klasse `Process` die Beobachter-Schnittstelle `ProcessObserver` und muss deshalb von `Observable<ProcessObserver>` abgeleitet sein.

Um mit diesem Konzept alle Simulationsdaten erfassen zu können, müsste jedem einzelnen Objekt eines Simulationsprogramms ein Beobachter-Objekt zugeordnet werden. Allein wegen der Vielzahl unterschiedlicher Observer-Schnittstellen ist dies jedoch unpraktikabel, da eine Datenbankankbindung eine Implementierung aller Schnittstellen bereitstellen müsste, damit alle Zustandsänderungen beobachtet werden können.

Trace

Auch im **Trace**-Konzept spielen Objekte zwei verschiedene Rollen: die eines *Trace Producer*, der Daten sendet, oder die eines *Trace Consumer*, der Daten empfängt und weiterverarbeitet. Die Unterstützung des Konzepts erfordert allerdings von jeder datenerzeugenden Komponente, dass sie alle relevanten Informationen durch Funktionsrufe an ein Trace-Objekt sendet, welches als Vermittler die Daten an Konsumenten weiterverteilt. Simulationsereignisse werden in Form sogenannter Markierungen versendet, welche jeweils Instanzen der Klasse `MarkType` sind. Bibliotheksseitig wird die Erzeugung einfacher und auch komplexer Trace-Markierungen unterstützt.

Durch den Trace-Mechanismus ist es möglich, alle Zustandsänderungen eines Systemmodells zu erfassen, womit ODEmx-spezifisch das umgesetzt wird, was eine allgemeine Datenerfassung auch leisten soll. Für eine Datenbankankbindung zur umfassenden Sammlung von Simulationsdaten wäre dieses Konzept prinzipiell geeignet. Die mit dem Trace-Mechanismus von ODEmx gesammelten Erfahrungen werden deshalb auch bei der Entwicklung des im Kapitel 3 beschriebenen generischen Logging-Konzepts mit einfließen.

2.2 Datenbanken

Eines wichtiges Ziel der Arbeit ist die Schaffung einer Datenbankankbindung für ODEmx, welche das systematische und sichere Archivieren von Ablaufprotokollen und Ergebnissen durchgeführter Simulationsexperimente ermöglicht. Auf diese

Weise kann eine bessere Integration mit dem am Lehrstuhl entwickelten und in der aktuellen Forschung eingesetzten Experimentmanagement-System erreicht werden. Daher geben die nachfolgenden Abschnitte eine kurze Einführung zum Thema Datenbanken, die auf dem Grundlagenbuch von Elmasri und Navathe basiert [Elma00].

2.2.1 Begriffsklärung

Der Begriff der Datenbank bedarf allerdings zunächst einer genauen Definition. Eine **Datenbank** ist eine Sammlung von Daten, die einen Ausschnitt der realen Welt beschreiben, wobei **Daten** in diesem Kontext bekannte Tatsachen sind, die aufgezeichnet werden können. Eine Datenbank hat folgende Eigenschaften:

- Sie stellt Aspekte der realen Welt als sogenannte Miniwelt dar. Änderungen in der Miniwelt spiegeln sich in der Datenbank wieder.
- Sie ist eine logisch zusammenhängende Sammlung von Daten. Zufällige Datensammlungen werden also nicht als Datenbank bezeichnet.
- Sie wird für einen bestimmten Zweck entworfen, entwickelt und mit Daten gefüllt. Eine bestimmte Nutzergruppe verwendet sie in zweckbezogenen Anwendungen.

Computergestützte Datenbanken werden meist mit Hilfe eines Datenbankmanagementsystems erstellt und gepflegt. Ein **Datenbankmanagementsystem (DBMS)** ist ein Softwaresystem, das mehrere Prozesse im Umgang mit Datenbanken vereinfacht:

- Definition einer Datenbank: Spezifikation von Datentypen, Strukturen und Einschränkungen für die zu speichernden Daten
- Konstruktion einer Datenbank: das Speichern der Daten auf einem Speichermedium, das vom DBMS kontrolliert wird
- Interaktion mit einer Datenbank: Anfragen zum Abruf bestimmter Daten, Eintragen von Änderungen der Miniwelt sowie das Erzeugen von Berichten aus Daten

Die Vereinigung von Datenbanken und DBMS-Software bildet dann ein sogenanntes **Datenbanksystem**. Unter den Datenbanksystemen kann weiterhin zwischen **eingebetteten** und **externen** Systemen unterschieden werden. Erstere sind meist

direkt in eine Software integriert und verlangen wenig oder gar keine Nutzerinteraktion. Es gibt darunter sowohl Systeme, die ausschließlich im Hauptspeicher vorgehalten werden, als auch dateibasierte oder kombinierte Ansätze. Die Kategorie der externen Datenbanksysteme betrifft Systeme, die separat von der darauf zugreifenden Software installiert und damit unabhängig von speziellen Anwendungen sind. Sie können von jeder Software angesprochen werden, die sich mit dem DBMS verbinden und über eine vorgegebene Schnittstelle kommunizieren kann.

2.2.2 Datenmodellierung

Das 1976 von Chen entwickelte **Entity-Relationship-Modell (ER-Modell)** wird im Rahmen der semantischen Datenmodellierung zur Beschreibung eines systemrelevanten Ausschnitts der Realität verwendet. Nach Chens Definition ist eine Entität „ein Ding, das eindeutig identifiziert werden kann“ [Chen76]. Diese Arbeit orientiert sich an Schubert [Schub04], der diese Definition aufgreift und zudem Eigenschaften von Entitäten mit einbezieht, was zu folgender Definition führt:

Eine **Entität** ist jedes Ding, das in einem systemrelevanten Ausschnitt der Realität mehrere Eigenschaften hat und aufgrund dieser Eigenschaften eindeutig identifiziert werden kann. Die Eigenschaften dieses Dinges heißen **Attribute**.

Das ER-Modell ist ein konzeptuelles Datenmodell, das zur Erstellung konzeptueller Datenschemata verwendet wird. Grundlage für die Erstellung der Schemata ist eine Typisierung oder Klassenbildung von Entitäten und deren Beziehungen untereinander. Der Prozess der Typisierung von Entitäten ist nicht durch einen formalen Algorithmus vorgegeben. Nach Schubert gilt jedoch folgende Regel:

Alle Entitäten, die mit der gleichen Kombination von Attributen beschrieben werden und für die es außerdem ein Attribut gibt, das genau für diese Entitäten den gleichen Attributwert hat, werden zu einer **Entitätenklasse** zusammengefasst.

Das aus der Typisierung und Beziehungsbestimmung resultierende Schema ist eine kompakte Beschreibung vorgegebener Datenanforderungen, die ausführliche Beschreibungen der Entitätenklassen, Beziehungen und Einschränkungen beinhaltet.

Hauptbestandteil dieser Modellierungsart sind spezielle Diagramme, sogenannte **ER-Diagramme**, anhand derer die analysierten Strukturen veranschaulicht werden.

Den in dieser Arbeit behandelten relationalen DBMS (RDBMS) liegt das 1970 von Codd entwickelte **relationale Datenmodell** zugrunde. In diesem Modell werden Relationen durch ein **Relationsschema** beschrieben. Dieses setzt sich aus einem Relationsnamen und einer Menge von Attributen zusammen. Jedes Attribut bezeichnet dabei eine Rolle, die sein Wertebereich im Relationsschema spielt. Eine **Relation** ist definiert als Menge von Tupeln. Jedes zu einer Relation gehörige Tupel stellt eine bestimmte Tatsache dar. Indem einige Relationen Tatsachen über Entitäten wiedergeben, andere Relationen jedoch Beziehungen darstellen, wird im relationalen Datenmodell eine einheitliche Darstellung von Entitäten und Beziehungen erreicht.

Da sich alle Elemente einer Menge unterscheiden, muss dies auch auf die einzelnen Tupel einer Relation zutreffen. Es gibt also immer eine Teilmenge von Attributen, deren Wertekombination sich bei zwei beliebigen verschiedenen Tupeln der gleichen Relation unterscheidet. Eine solche Attributmenge heißt **Superschlüssel**. Der Default-Superschlüssel ist die Menge aller Attribute. Ist der Superschlüssel minimal¹, so wird diese Menge als **Schlüssel** bezeichnet. Im Allgemeinen kann ein Relationsschema mehr als einen Schlüssel umfassen – daher wird jeder dieser Schlüssel als **Schlüsselkandidat** bezeichnet. Einer dieser Kandidaten wird dann als sogenannter **Primärschlüssel** festgelegt, mit dessen Werten die Tupel einer Relation identifiziert werden. In relationalen Datenbanken werden Relationen als Tabellen repräsentiert, wobei die Tupel der Relation einzelnen Tabellenzeilen zugeordnet werden. Die Spaltenüberschriften entsprechen dabei den Attributen des Relationsschemas.

Im relationalen Datenmodell lassen sich verschiedene Integritätsbedingungen angeben. Beispielsweise gilt für die eben definierten Primärschlüssel die Bedingung der **Entitätsintegrität**, die besagt, dass kein Primärschlüsselwert den Nullwert² annehmen darf, weil sonst möglicherweise Tupel einer Relation nicht mehr unterschieden werden können. Auch zwischen Tupeln verschiedener Relationen gibt es derartige Bedingungen. Wichtig ist hier vor allem die **referenzielle Integrität**,

¹ Es kann kein Attribut entfernt werden, ohne die Eindeutigkeit zu verletzen.

² Enthält ein Attribut eines Tupels den Nullwert, so ist der Wert entweder unbekannt oder existiert für dieses Tupel nicht.

welche besagt, dass ein Tupel der Relation, auf die sich ein Tupel einer anderen Relation bezieht, existieren muss. Bedingungen der referenziellen Integrität entstehen normalerweise aus den Beziehungen zwischen den Entitätenklassen, die durch Relationsschemata dargestellt werden.

In diesem Rahmen wird das Konzept des Fremdschlüssels verwendet. Man betrachte dafür zwei Relationsschemata R_1 und R_2 . Sei PK der Primärschlüssel von R_1 . Eine Attributmenge FK im Relationsschema R_2 ist ein **Fremdschlüssel** von R_2 , der R_1 referenziert, falls die Attribute in FK die gleichen Wertebereiche wie die Attribute in PK haben und die Wertekombination von FK eines Tupels t_2 der Relation R_2 auch als Wert von PK in einem Tupel t_1 der Relation R_1 auftritt. In diesem Fall **referenziert** t_2 das Tupel t_1 . R_1 ist dann die **referenzierte** und R_2 die **referenzierende Relation**. Im Gegensatz zu Primärschlüsselattributen dürfen Fremdschlüsselattribute jedoch auch den Nullwert annehmen.

Eine relationale Datenbank enthält typischerweise viele Relationen mit Tupeln, die verschiedenste Zusammenhänge untereinander aufweisen. Ein **relationales Datenbankschema** besteht daher aus einer Menge von Relationsschemata und einer Menge von Integritätsbedingungen, die bei jeder Modifikation der Datenbank erfüllt sein müssen.

2.2.3 Normalisierung

Die **Normalisierung** eines relationalen Datenbankschemas dient der Vermeidung von Datenredundanz, welche Inkonsistenzen zwischen Datensätzen verursachen kann. Diese sogenannten **Anomalien** können beim Einfügen, Modifizieren, oder Löschen von Datensätzen auftreten. Die aus einer Normalisierung resultierenden Datenbankschemata sind weniger fehleranfällig in ihrer Verwendung.

Der Faktor Redundanz spielt im Kontext der Computersimulation eine wichtige Rolle, da Simulationsexperimente im Allgemeinen große Datenmengen erzeugen. Allerdings ist der durch Redundanz entstehende Speicherverbrauch auch gegen den Faktor Leistung abzuwägen, denn häufig gilt es, zusätzliche Datenbankabfragen wegen der zeitaufwendigen Ein-/Ausgabeoperationen beim Festplatten- oder Netzwerkzugriff zu vermeiden. Die Überprüfung eines relationalen Datenbankschemas bezüglich seiner Redundanzfreiheit erfolgt anhand der sogenannten Normalformen, von denen an dieser Stelle die ersten drei betrachtet werden.

Die **erste Normalform** liegt vor, wenn die Wertebereiche aller Attribute einer Relation nur atomare Werte enthalten. Es dürfen also keine Werte vorkommen, die aus mehreren Einzelwerten bestehen – ein Beispiel wäre ein Attribut Adresse, dessen Werte jeweils in Straße, Hausnummer und Ort zerlegt werden können.

Die zweite Normalform befasst sich mit Nichtschlüsselattributen, die nur von einem Teil der Attribute eines Schlüsselkandidaten abhängen. Dafür benötigt man jedoch zuerst den Begriff der funktionalen Abhängigkeit:

Seien A, B beliebige Teilmengen der Attribute einer Relation R . Dann heißt B **funktional abhängig** von A , wenn zu jeder Wertekombination der Attribute aus A genau eine Wertekombination der Attribute aus B gehört. Notiert wird dies als $A \rightarrow B$.

Die **zweite Normalform** liegt vor, wenn die erste Normalform erfüllt und kein Nichtschlüsselattribut von einer echten Teilmenge eines Schlüsselkandidaten funktional abhängig ist. Diese Normalform ist automatisch erfüllt, wenn die Schlüsselkandidaten jeweils aus maximal einem Attribut bestehen.

Die dritte Normalform behandelt Abhängigkeiten zwischen Nichtschlüsselattributen. Sie benötigt den Begriff der transitiven Abhängigkeit:

Seien A, B, C beliebige Teilmengen der Attribute einer Relation R . Dann heißt C **transitiv abhängig** von A , wenn $A \rightarrow B$ und $B \rightarrow C$. Notiert wird dies als $A \rightarrow B \rightarrow C$.

Die **dritte Normalform** liegt vor, wenn die zweite Normalform erfüllt ist und kein Nichtschlüsselattribut von einem Schlüsselkandidaten transitiv abhängt. Nichtschlüsselattribute dürfen also nicht von einer Menge anderer Nichtschlüsselattribute abhängig sein.

Werden diese Normalformen von einem relationalen Datenbankschema nicht erfüllt, so können wie eingangs erwähnt verschiedene Arten von Anomalien in der Datenbank auftreten. Eine **Einfügeanomalie** ist gegeben, wenn das Einfügen eines neuen Datensatzes die korrekte Angabe bestehender Informationen verlangt, um die Konsistenz der Datensätze zu wahren, oder das Einfügen eines Datensatzes nicht möglich ist, weil die Werte bestimmter Attribute nicht bekannt sind. Eine **Löschanomalie** tritt auf, wenn Datensätze anhand der Werte bestimmter Attribute aus einer Tabelle entfernt werden und dabei Informationen verloren gehen, die in anderen Attributen einer Relation vorgehalten werden. **Modifikationsanomalien**

treten immer dann auf, wenn bei der Änderung eines Attributwertes nicht alle Datensätze aktualisiert werden, die denselben Wert enthalten müssen.

2.2.4 Generelle Vorteile der Datenbanknutzung

Datenbanksysteme dienen der kontrollierten Verwaltung von Daten, die von verschiedenen Anwendungen und Nutzern gemeinsam verwendet und bearbeitet werden können. Die zuverlässige Umsetzung dieser Funktionalität wird üblicherweise durch eine Transaktionsverwaltung realisiert. Eine **Transaktion** ist dabei eine Sequenz von Interaktionen mit der Datenbank, deren Ausführung untrennbar (atomar) ist, so dass entweder alle Aktionen ausgeführt werden sollen oder keine. Sichergestellt ist dies, wenn Transaktionen eines RDBMS die sogenannten **ACID-Eigenschaften** (*atomicity, consistency, isolation, durability*) erfüllen [Härder83]:

- **Atomarität:** Transaktionen werden entweder ganz oder gar nicht ausgeführt. Im Erfolgsfall wird die Transaktion mit dem sogenannten **Commit** abgeschlossen und die Datenbank ist in einem konsistenten Zustand. Im Fehlerfall werden sämtliche zwischenzeitlichen Änderungen durch das sogenannte **Rollback** rückgängig gemacht, so dass der zu Transaktionsbeginn geltende Zustand wieder hergestellt wird.
- **Konsistenz:** Transaktionen dürfen nur gültige Datensätze zum Resultat haben, welche die Integritätsbedingungen eines Datenbestands nicht verletzen. Sollte dies doch der Fall sein, so muss die Transaktion abgebrochen werden, wodurch wegen der Atomarität der konsistente Ausgangszustand wieder hergestellt wird.
- **Isolation:** Parallel ablaufende Transaktionen dürfen sich nicht gegenseitig beeinflussen. Werden also am Anfang einer Transaktion Daten gelesen, die verändert und wieder geschrieben werden sollen, so darf in der Zwischenzeit keine andere Transaktion diese Daten verändern.
- **Dauerhaftigkeit:** Sobald ein erfolgreicher Transaktionsabschluss in der Datenbank persistent gemacht wurde, muss das DBMS garantieren, dass dieses Resultat auch nach Systemfehlern bestehen bleibt, so dass der letzte konsistente Zustand in jedem Fall wiederhergestellt werden kann.

Da moderne RDBMS die ACID-Eigenschaften erfüllen, bringt das Speichern von Simulationsdaten in einer Datenbank also neben deutlich verbesserter Strukturie-

rung der Datenmengen und einer einheitlichen Zugriffs- und Verarbeitungslogik noch weitere wichtige Vorteile mit sich.

Die Atomarität der Transaktionen sichert, dass immer komplette Datensätze geschrieben werden, selbst wenn mehrere Programme gleichzeitig ausgeführt werden, welche dieselbe Datenbank modifizieren. Bei der parallelen Nutzung einer normalen Textdatei wäre dies nicht ohne weiteres der Fall. Betrachtet man beispielsweise strukturierte Daten wie XML, so wäre es durchaus möglich, dass Elemente nicht korrekt abgeschlossen werden, weil gleichzeitig ein anderes Programm seine Daten dazwischen schreibt.

Die Konsistenz gespeicherter Datensätze ist eine Eigenschaft, die mit reiner Dateiarbeit nur schwer zu erreichen ist. Sie sichert die Widerspruchsfreiheit von Datenbeständen. RDBMS bieten dafür die Möglichkeit Integritätsbedingungen zu definieren. Damit lassen sich beispielsweise Referenzbeziehungen zwischen Relationen festlegen oder gültige Wertebereiche für einzelne Attribute von Relationen definieren. Diese Integritätsbedingungen werden bei der Modifikation der Datenbank automatisch auf Gültigkeit überprüft.

Die Isolation sorgt weiterhin dafür, dass parallel laufende Programme ihre gesammelten Daten nicht gegenseitig überschreiben. So wäre es bei Verwendung derselben Textdatei durchaus denkbar, dass die gerade erzeugten Daten eines Programms durch ein parallel laufendes Programm überschrieben werden, weil dessen Dateizeiger nicht auf die letzte Position aktualisiert wurde. Bei der Dateiarbeit müsste also die notwendige Synchronisation zusätzlich implementiert werden.

Ein wichtiger Faktor ist natürlich auch die Dauerhaftigkeit, die das zuverlässige Archivieren großer Datenmengen garantiert. Beispielsweise muss es während der Auswertungsphase von Simulationsexperimenten immer möglich sein, schnell und gezielt auf bisher gewonnene Simulationsergebnisse zuzugreifen. Es wäre fatal, wenn einmal gewonnene Resultate durch Systemfehler verloren gingen, da die Durchführung von Simulationsexperimenten typischerweise sehr zeitaufwendig ist und unnötige Wiederholungen von Experimenten zu vermeiden sind.

3 Logging

3.1 Einführung

3.1.1 Begriffsklärung

Mit steigender Komplexität von Software erhöht sich auch die Wahrscheinlichkeit, dass diese Fehler enthält, weshalb prinzipiell bei jeder Ausführung eines Programms mit dem Auftreten von Fehlverhalten gerechnet werden muss. Eine praktikable Strategie, den Hergang eines solchen Ereignisses nachvollziehbar zu machen, ist dabei das Protokollieren relevanter Daten des aktuellen Laufzeitzustands, wodurch zu jedem Ausführungszeitpunkt eines Programms präzise Kontextinformationen bereitgestellt werden können. Dieses Vorgehen wird in der Softwareentwicklung als **Logging** bezeichnet. Die einfachste Form des Loggings ist die Ausgabe von Laufzeitinformationen auf einer Textkonsole, was allerdings bei hohem Datenaufkommen schnell unüberschaubar und damit nutzlos wird. Die Konsequenz ist meist das Speichern von strukturierten Daten in Dateien oder einer Datenbank [Gupta05].

3.1.2 Verwendung in der Computersimulation

Da sich die Computersimulation primär mit der Berechnung und Analyse von erreichten Systemzuständen und den daraus zu gewinnenden Daten befasst, kommt dem Logging in diesem Betätigungsfeld neben der Fehleranalyse eine noch bedeutsamere Rolle zu: die allgemeine Datenerfassung während des Simulationslaufs. Damit einher geht die Forderung nach Persistenz der Daten zur späteren Auswertung, welche durch Speicherung in Dateien oder einer Datenbank erfüllt werden kann.

Allerdings entstehen erfahrungsgemäß bereits bei einzelnen Simulationsexperimenten sehr große Datenmengen, die im Falle der Dateiarbeit oft das Filtern, also gezieltes Weglassen von Informationen, unausweichlich machen. So lassen sich beispielsweise HTML-Dateien in der Größenordnung weniger Megabytes bereits nicht mehr in gängigen Browsern verarbeiten. Andererseits kann auch das Filtern problematisch sein, da sich während der Auswertung eines Experiments zusätzliche Fragestellungen ergeben können, für deren Beantwortung dann unter Umständen wichtige Daten fehlen. Wünschenswert wäre die Sicherung der kompletten Simulationsdaten aller durchgeführten Experimente, weshalb die Nutzung einer Datenbank vorzuziehen ist. Dies bringt den weiteren Vorteil mit sich, dass in der Auswertungsphase die Infrastruktur der Datenbank zur gezielten Abfrage von Daten verwendet werden kann.

3.1.3 Konzepte aktueller Logging-Frameworks

Obwohl in der Softwareentwicklung häufig verschiedene Formen des Loggings eingesetzt werden, gibt es wenig Fachliteratur, die sich speziell diesem Thema widmet. Lediglich Publikationen zu log4j wie [Gülcü03] und [Gupta05] bilden hier die Ausnahme. Dabei handelt es sich um ein populäres quelloffenes Logging-Framework für Java, das von der Apache Software Foundation gepflegt wird und neben direkten Portierungen auch viele Logging-Implementationen in anderen Programmiersprachen inspiriert hat. Aber auch Logging-Bibliotheken anderen Ursprungs basieren meist auf den gleichen Konzepten, welche im Folgenden anhand zweier unterschiedlicher Implementierungen herausgearbeitet werden.

Da die Programmiersprache C++ die Grundlage von ODEmx bildet, liegt es nahe, vorhandene Logging-Frameworks in derselben Sprache im Hinblick auf eine mögliche Verwendung zu untersuchen. Wie bereits erwähnt, gibt es Portierungen von log4j für verschiedene Sprachen, und so wird an dieser Stelle die ebenfalls von der Apache Software Foundation bereitgestellte C++-Variante log4cxx untersucht [Apache]. Für den zweiten Kandidaten wurde mit Boost eine andere Sammlung hochwertiger, quelloffener C++-Bibliotheken betrachtet. In der Vergangenheit sind bereits mehrere dieser Bibliotheken als Standarderweiterung akzeptiert worden. Zwar gibt es im Hauptzweig keine offizielle Logging-Bibliothek, aber es existiert bereits eine ausgereifte Version von Boost.Log, die hier zur Konzeptanalyse herangezogen werden kann [Boost].

Produzenten, Konsumenten und Filter

Der Prozess des Loggings wird im Wesentlichen durch drei Arten von Komponenten umgesetzt: produzierende, konsumierende und filternde Komponenten. Die Quelle von Log-Daten sind meist Objekte von Klassen der ersten Kategorie, die in vielen Implementationen als `Logger` bezeichnet werden. In beiden Bibliotheken finden sich Klassen mit diesem Namen. Allerdings ist das Konzept von `Boost.Log` an dieser Stelle allgemeiner, da Log-Daten auch ohne `Logger`-Objekte erzeugt werden können. Die zugehörigen Senken für Log-Einträge werden durch Klassen der zweiten Kategorie repräsentiert, wobei diese in `log4cxx` `Appender` und in `Boost.Log` `sink` genannt werden. Die dritte Kategorie wird in beiden Bibliotheken unterschiedlich gehandhabt. In `log4cxx` gibt es die Basisklasse `Filter`, deren Spezialisierungen verschiedene Filter auf der Basis von Zeichenketten, Ausdrücken oder bestimmten Eigenschaften von Log-Einträgen implementieren, wohingegen `Boost.Log` aus booleschen Ausdrücken über den Attributen von Log-Einträgen `Filter` in Form von C++-Funktionen generiert.

Zentrale Kontrolle

In `log4cxx` haben alle `Logger` einen Namen und werden anhand dessen hierarchisch in einer zentralen Datenstruktur, der sogenannten `LoggerRegistry`, eingeordnet. Dadurch lassen sich bestimmte Eigenschaften von hierarchisch höher stehenden `Loggern` erben. Ebenso kann auf diese Weise gesteuert werden, zu welchen Senken Log-Einträge geleitet werden sollen. `Boost.Log` verzichtet zwar auf `Loggernamen` und eine globale Hierarchie, weil sich ähnliche Funktionalität auch mit Attributen und Filtern umsetzen lässt, aber auch hier gibt es eine zentrale Komponente, den sogenannten `core`, welcher anhand der Attribute eines Log-Eintrags überprüft, ob dieser von mindestens einer Senke akzeptiert wird, und ihn gegebenenfalls weiterleitet.

Einstufung oder Wichtung

Eine Eigenschaft, die viele Logging-Frameworks implementieren, ist eine Einstufung der Log-Einträge nach ihrer Wichtigkeit, wofür in der Regel eine aufsteigende Folge sogenannter Levels vorgegeben wird. In `log4cxx` sind dies `ALL`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL` und `OFF`. Damit lässt sich ein Schwellwert angeben, den ein

Log-Eintrag überschreiten muss, um weitergeleitet zu werden. Boost.Log macht keine solche Vorgabe, liefert aber eine Klasse `severity_logger`, mit deren Hilfe beliebige vom Anwender vorgegebene Levels als Attribute verwendet werden können. Analog dazu gibt es noch eine Klasse `channel_logger`, die Attribute für die Kategorisierung von Log-Daten verwendet.

Zeichenketten als Datentypen

Da Log-Einträge sehr häufig als Zeichenketten (*strings*) abgespeichert werden, könnten diese entweder direkt durch einen `String`-Typ repräsentiert sein, oder durch eine Klasse, die neben einer Zeichenkette noch weitere Attribute aufnehmen kann, die dann zum Beispiel beim Filtern und Formatieren verwendet werden. Beide Bibliotheken verwenden intern die zweite Variante, auch wenn die Nutzerschnittstelle durch Funktionen gegeben ist, die Zeichenketten als Parameter fordern. In `log4cxx` transportieren Objekte der Klasse `LogEvent` neben der Nachricht fest vorgegebene Informationen zur Identifikation des Erzeugers und der Codeposition, sowie einen Zeitstempel, während in Boost.Log Objekte der Klasse `record` Textnachrichten aufnehmen und zusätzlich beliebige nutzerdefinierte Attribute transportieren können.

Formatierung der Ausgabe

Da von Zeichenketten als Zielformat ausgegangen wird, stellen beide untersuchten Bibliotheken eine Möglichkeit der Formatierung bereit: `log4cxx` bietet dafür die Klasse `Layout` und mehrere vordefinierte Spezialisierungen, während Boost.Log zwei Möglichkeiten anbietet, Format-Funktoren mit einer bestimmten Signatur zu generieren.

Multithreading

Bei der Verwendung mehrerer Threads in einem Programm können während der Laufzeit schwer nachvollziehbare Fehler wie Verklemmungen (*deadlocks*) oder Wettlaufsituationen (*race conditions*) auftreten, deren Analyse oftmals nur durch konsequentes Logging gelingt. Daher verfügen sowohl `log4cxx` als auch Boost.Log über Komponenten zur Identifikation von Threads und ihrem aktuellen Laufzeitzustand.

Konfigurationsdateien

Ein bedeutsamer Unterschied der beiden Bibliotheken liegt im Bereich der Konfiguration. Zwar sind beide Bibliotheken innerhalb eines Programms ähnlich zu konfigurieren, log4cxx ermöglicht dies jedoch auch noch auf andere Weise: durch Konfigurationsdateien. So ist es sogar während der Laufzeit eines Programms möglich, die Logging-Einstellungen durch Änderung einer einfachen Text- oder XML-Datei zu beeinflussen. Ein offensichtlicher Vorteil ist dies bei größeren oder lange laufenden Anwendungen, wo ein erneutes Übersetzen oder die Beendigung eines Programms nicht ohne Weiteres geschehen können.

3.2 Ein generisches Logging-Konzept

In diesem Abschnitt erfolgt zunächst eine Analyse der Implementation des bisher in ODEMX verwendeten Trace-Mechanismus, die mögliche Verbesserungsansätze aufzeigen soll. Daran anschließend werden unter Einbeziehung vorangegangener Betrachtungen die angestrebten Ziele und grundlegende Aspekte eines generischen Logging-Konzeptes beschrieben, gefolgt vom rollenbasierten Entwurf von Komponentenklassen und einer Erläuterung ihres Zusammenspiels. Alle in diesem Abschnitt dargelegten Aspekte des generischen Logging-Konzeptes sind unabhängig von der späteren Implementationssprache.

3.2.1 Analyse des Trace-Mechanismus von ODEMX

Die Struktur des in Abschnitt 2.1.3 beschriebenen Trace-Mechanismus von ODEMX ist in Abbildung 3.1 dargestellt. Darin sind bereits mehrere der oben herausgearbeiteten Konzepte moderner Logging-Bibliotheken umgesetzt. Die Quelle von Trace-Einträgen ist durch eine Schnittstelle für Produzenten vorgegeben. Das Gegenstück dazu ist eine Schnittstelle für Konsumenten, welche die Senken repräsentiert. Die Klassen `Process` und `XmlTrace` sind Beispiele für Implementierungen dieser Schnittstellen. Eine zentrale Kontrolleinrichtung ist durch die Klasse `Trace` gegeben, welche für die Weiterleitung von Trace-Markierungen beliebig vieler Produzenten an beliebig viele Konsumenten verantwortlich ist. Das Filtern wiederum wird von den Konsumenten selbst implementiert, indem diese von der Klasse `TraceFilter` abgeleitet werden.

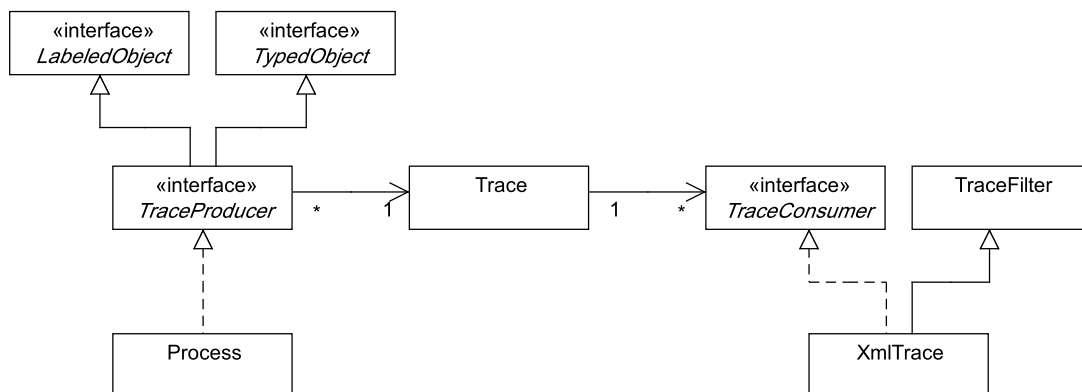


Abbildung 3.1: Klassenstruktur des Trace-Mechanismus

Die Verwendung vordefinierter Schnittstellen bildet in der objektorientierten Programmierung eine wichtige Grundlage für austauschbare Komponenten. So kann die Klasse *Trace* Daten an beliebige Konsumentenklassen weiterleiten, wenn diese die *TraceConsumer*-Schnittstelle implementieren. Die Schnittstelle *TraceProducer* garantiert dagegen, dass alle Produzenten einen eindeutigen Namen haben (*LabeledObject*) und dass ihr Typ abfragbar ist (*TypedObject*). Dieser Flexibilität bezüglich der Austauschbarkeit steht jedoch die Implementation des *Trace*-Konzepts entgegen, die dem Nutzer eine ausgiebige Verwendung unnötig erschwert. Die Schnittstelle für Konsumenten verlangt vom Anwender die Implementierung von 38 rein virtuellen Methoden, da zur Erzeugung von *Trace*-Markierungen und zusätzlichen Detailsinträgen jeweils separate Methoden für jeden einfachen C++-Datentyp vorgegeben sind. Diese Vorgabe ist viel zu umfangreich, um eine zügige Implementierung neuer Ausgabe- oder Darstellungskomponenten zu ermöglichen.

Problematisch ist auch, dass für jede Produzentenklasse in ODEMX statische *Trace*-Einträge definiert werden. Dadurch können bei der Verwendung von globalen Objekten im Simulationsprogramm Initialisierungsfehler auftreten, da der C++-Standard keine Reihenfolge für die Konstruktion dieser Objekte vorgibt. Auch gab es einige Missverständnisse wegen der unpassenden Namensgebung mancher Komponenten, die nicht auf ihre Funktion schließen lässt. Beispiele sind die Klasse *MarkType*, welche *Trace*-Einträge beschreibt, und die Klasse *Tag*, welche zusätzliche Details von Einträgen repräsentiert.

In Anlehnung an XML gibt es die Möglichkeit, in einzelnen Trace-Einträgen eine beliebige Hierarchie von Detailinformationen zu bilden. Das ist jedoch hinderlich für die Entwicklung von Trace-Konsumenten oder Auswertungswerkzeugen, die ein genau definiertes Format benötigen, um für beliebige Simulationen einsetzbar zu sein. Wenn Trace-Einträge eines Simulationsprogramms beliebige Detailhierarchien enthalten können, deren Beschreibung durch ein allgemeines XML- oder Datenbankschema nicht möglich ist, würden extra dafür spezielle Auswertungswerkzeuge benötigt, welche die erzeugten Daten verarbeiten können. Beispielsweise müsste für jedes XML-Format ein eigener Parser implementiert werden, um gezielt Daten zu extrahieren. Aus diesem Grund fanden Trace-Einträge mit beliebigen Detailhierarchien in Anwendungen von ODEMx bisher auch keine Verwendung.

Als Ergebnis dieser Betrachtungen zum Trace-Mechanismus bleibt festzuhalten, dass sich die modulare Grundstruktur mit den gegebenen Rollen bewährt hat, auch wenn Schwächen in der Implementation nicht das volle Potenzial zur Wirkung kommen ließen. So ist gerade eine einfache Konsumentenschnittstelle wichtig, um die Erstellung von Ausgabekomponenten zu erleichtern, und die Namensgebung von Klassen sollte sich an ihrer Funktionalität orientieren. Außerdem wird eine Einschränkung im Bereich der Detailhierarchien für die Datenbankbindung unvermeidbar sein, weil für die Aufnahme beliebiger Simulationsdaten in derselben Datenbank ein festes relationales Datenbankschema benötigt wird.

3.2.2 Zielsetzungen

Auf der Grundlage der beiden vorhergehenden Abschnitte, welche die Konzepte moderner Logging-Bibliotheken und eine Analyse des ODEMx-internen Trace-Mechanismus zum Inhalt haben, werden an dieser Stelle Ziele formuliert, die das angestrebte generische Logging-Konzept umsetzen soll.

Wie bereits im letzten Abschnitt angedeutet wurde, soll die Komponentenstruktur des Logging-Konzeptes am bisherigen Trace-Mechanismus angelehnt sein, da dieses Konzept die naheliegendste Umsetzung einer $m \times n$ -Beziehung zwischen Quellen und Senken darstellt und in ähnlicher Form auch in anderen Bibliotheken zur Anwendung kommt. Es sollen also Komponenten für Produzenten und Konsumenten angeboten werden, die durch einen Verteilungsmechanismus zu koppeln

sind. Die Umsetzung der Konsumenten-Schnittstelle muss dabei deutlich einfacher ausfallen, um die Schaffung neuer Komponenten zu erleichtern.

Bei der Betrachtung der Datentypen hat sich herausgestellt, dass Logging-Bibliotheken auf Zeichenketten fokussiert sind, auch wenn diese intern nochmals gekapselt werden. Das zugrundeliegende Konzept aus Produzenten und Konsumenten beliebiger Daten könnte jedoch durchaus von der Nutzung anderer Datentypen profitieren. Beispielsweise wäre es möglich, dass spezialisierte Produzenten bestimmte Zahlenformate an spezialisierte Konsumenten senden, die damit weitere Berechnungen anstellen – eine String-Kodierung nur zum Transport der Daten mit anschließender Dekodierung wäre in einem solchen Fall nicht sinnvoll. Mit der Bibliothek Boost.Log ließe sich dieser Spezialfall sogar realisieren, indem man die Zeichenkette der Klasse `record` ignoriert und ein Attribut zum eigentlichen Datentransport benutzt. Um wirklich generisch zu sein, sollte das Logging-Konzept jedoch einen allgemeineren Mechanismus anbieten, was durch eine Parametrisierung der Komponenten bezüglich des Typs von Log-Daten erreicht werden kann.

Der Software-Entwicklungsprozess erfordert häufig eine Möglichkeit zur Kategorisierung von Log-Daten, so dass beispielsweise Debug-Informationen, Warnungen oder Fehlerausschriften getrennt beobachtet werden können. Unterstützung dafür findet sich auch in den oben betrachteten Logging-Bibliotheken `log4cxx` und `Boost.Log`. Der bisherige Trace-Mechanismus verfügt nicht über solche Funktionalität, da alle gesammelten Daten als gleichartige Trace-Einträge behandelt werden. Das allgemeine Logging-Konzept soll jedoch eine Kategorisierung von Daten erlauben. Im Gegensatz zu `log4cxx` wird allerdings nicht davon ausgegangen, dass eine Kategorisierung mit einer Wichtung von Log-Daten gleichgesetzt werden kann, weil keine Annahmen über die Bedeutung einzelner Kategorien gemacht werden können. Stattdessen soll das Konzept in Analogie zur `Boost.Log`-Klasse `channel_logger` beliebige Log-Kanäle unterstützen. Die Log-Levels von `log4cxx` zeigen jedoch, dass es durchaus sinnvoll ist, bestimmte Kategorien standardmäßig anzubieten. Tabelle 3.1 listet diese Kategorien auf und beschreibt ihre Intention.

Wegen der im Allgemeinen zahlreichen Zustandsänderungen eines Programms ist beim Logging immer ein hohes Datenaufkommen zu erwarten, weshalb auch eine Möglichkeit angeboten werden soll, um Daten nach bestimmten Kriterien zu filtern. Da der verwendete Log-Datentyp allerdings auf Grund der angestrebten Generizität nicht festgelegt ist, kann eine solche Vorgabe nur durch eine Schnittstelle geschehen. Weiterhin wurde in Abschnitt 3.1.2 bereits dargelegt, dass Filtern

Kategorie	Beschreibung
Trace	ein detaillierter Gesamtüberblick über Zustandsänderungen
Debug	für den Entwicklungsprozess relevante Daten
Information	beliebige Programmablaufinformationen
Warnung	unerwartete Eingaben oder Programmverläufe, die Fehler nach sich ziehen können
Fehler	Probleme, die das Programm jedoch nicht zum Abbruch zwingen
Kritischer Fehler	Probleme, die einen Abbruch oder Absturz zur Folge haben können

Tabelle 3.1: Typische Kategorisierung von Log-Daten

nicht immer sinnvoll ist und insbesondere der Bereich der Computersimulation von einer Datenbanknutzung profitieren könnte. Da diese Funktionalität sicher auch für andere Anwendungen vorteilhaft wäre, soll schon die generische Logging-Bibliothek eine Unterstützung für die Implementation von Ausgabekomponenten mit Datenbankbindung anbieten. Die erste Version der Bibliothek bleibt jedoch auf RDBMS beschränkt, da diese am weitesten verbreitet sind und in der Regel einen Dialekt der standardisierten Anfragesprache SQL als Zugriffsschnittstelle anbieten. Dieser Aspekt ist bedeutsam, weil die Logging-Bibliothek eine Interaktion mit möglichst vielen verschiedenen RDBMS erlauben soll. Im Sinne der Vielseitigkeit der Logging-Bibliothek sollten sowohl eingebettete als auch externe Datenbanksysteme nutzbar sein.

Einer der Hauptgründe, weshalb trotz verschiedener frei verfügbarer Logging-Bibliotheken dennoch häufig die Standardausgabe verwendet wird, ist der Komplexitätsgrad der Bibliotheken. Die Einbindung und Verwendung der aus dieser Arbeit resultierenden Logging-Bibliothek sollte also nur minimalen Aufwand erfordern. Da jeder C++-Programmierer mit den von der Standardbibliothek bereitgestellten Datenströmen für die Konsolen- und Dateiarbeit und deren Verwendung vertraut ist, wird eine ähnliche Nutzung der Logging-Bibliothek angestrebt.

Multithreading wird bei der Implementation der ersten Version der Bibliothek zunächst keine Berücksichtigung finden, da einerseits die C++-Standardbibliothek dafür bisher keine Unterstützung bereitstellt und andererseits der primäre Anwendungsfall – die Nutzung in ODEmx – daraus keine Vorteile ziehen könnte. In ODEmx-Modellen wird Parallelität auf sequentialisierte Koroutinenausführung abgebildet, die stets in einem Thread abläuft.

Konfigurationsdateien sind für den Anwendungsfall ODEmX ebenso von geringem Nutzen, da Simulationsprogramme in der Regel nur einmal für die Ausgabe konfiguriert werden, um dann mit denselben Einstellungen viele Experimente durchzuführen.

3.2.3 Rollen und Zusammenspiel aktiver Komponenten

Die am Logging-Vorgang beteiligten aktiven Komponenten können jeweils eine von drei verschiedenen Rollen ausüben: Quelle, Kanal oder Senke. Die **Quelle** wird hier genauso betrachtet wie im Fall von Boost.Log: diese Rolle kann im Prinzip jedes Objekt einnehmen, das die Möglichkeit hat, Log-Daten zu erzeugen und diese an einen Kanal zu senden. In Analogie zum Trace-Konzept wird als Erweiterung aber auch die Verwendung eindeutig bezeichneter Quellobjekte, sogenannter **Produzenten**, unterstützt. Ein **Kanal** repräsentiert eine Kategorie von Log-Daten. Kanal-Objekte haben die Aufgabe, Log-Einträge von der Quelle zur Senke zu transportieren. Damit bilden sie eine geeignete Stelle für das globale Filtern von Log-Daten, weil hier die Weiterleitung sofort unterbunden werden kann. Eine **Senke** konsumiert und verarbeitet Log-Daten, die sie über eine vorgegebene Schnittstelle von einem oder mehreren Kanälen erhält. Derartige Objekte werden im weiteren Verlauf der Arbeit als **Konsumenten** bezeichnet.

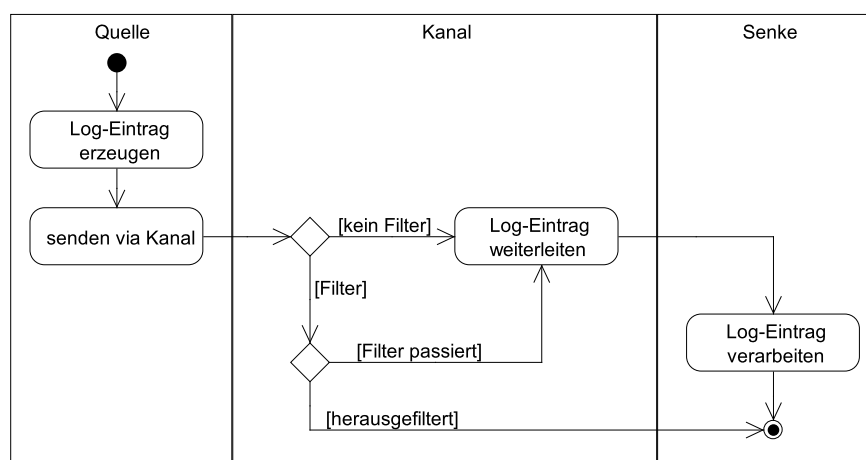


Abbildung 3.2: Ablauf eines Logging-Vorgangs mit Bezug auf die Rollenverteilung

In Abbildung 3.2 ist der Ablauf eines Logging-Vorgangs dargestellt. Eine Quelle erzeugt einen Log-Eintrag, der über einen Kanal transportiert werden soll. Hat der Kanal einen Filter, muss der Log-Eintrag diesen passieren. Vom Kanal wird der Log-Eintrag weitergeleitet an beliebig viele Senken, von denen jede einzelne selbst entscheidet, wie der Log-Eintrag verarbeitet wird. Insbesondere könnte dabei auch erneut lokal gefiltert werden.

3.2.4 Klassen und Assoziationen

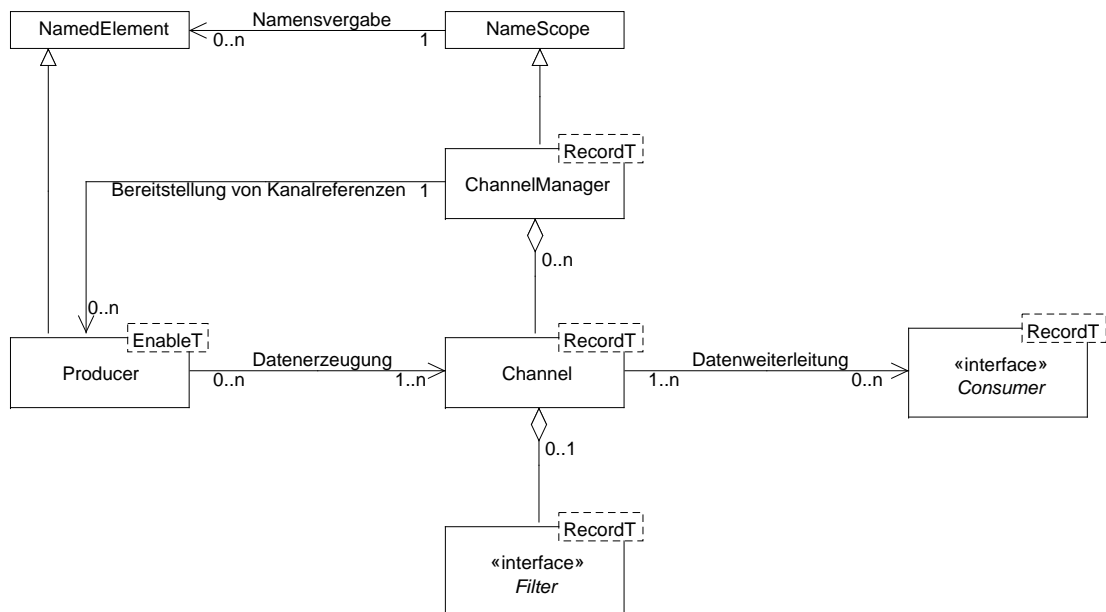


Abbildung 3.3: UML-Klassendiagramm des Logging-Konzepts

Die dem Logging-Konzept zugrundeliegende Klassenstruktur und ihre Assoziationen sind in Abbildung 3.3 dargestellt. Das Herzstück des Konzepts bilden die Klasse Channel und die Schnittstelle Consumer, da diese die Aktivitäten der Kategorisierung, Verteilung und Verarbeitung von Log-Daten umsetzen. Ein Kanal ist bezüglich seines Log-Datentyps (RecordT) parametrisiert. Ihm kann ein Filter zugeordnet werden, der im Diagramm durch die Schnittstelle Filter repräsentiert ist. An jedem Kanal können beliebig viele Konsumenten registriert werden. Zu beachten ist jedoch, dass sowohl der Filter als auch die Konsumenten mit dem gleichen Log-Datentyp wie der Kanal arbeiten müssen.

Durch die Klassen `NamedElement` und `NameScope` kann die eindeutige Namensgebung von Objekten kontrolliert werden. Objekte der Klasse `NamedElement` oder ihrer Spezialisierungen fordern unter Angabe eines Vorschlags einen eindeutigen Namen an, wobei die Klasse `NameScope` einen Gültigkeitsbereich repräsentiert, in dem die Eindeutigkeit der vergebenen Namen sichergestellt wird.

Die optional angebotene Klasse `Producer` unterstützt die eindeutige Bezeichnung ihrer Objekte durch Ableitung von `NamedElement`. Die Schaffung einer derartigen Produzenten-Klasse liegt in Erfahrungen mit ODEMX begründet. Wenn eine Software sehr viele Quellen von Log-Daten hat, so ist es häufig von Vorteil, als Quelle eines Log-Eintrags nicht nur eine Klasse oder Funktion lokalisieren zu können, sondern ein ganz bestimmtes Quell-Objekt.

Im Gegensatz zu den anderen generischen Klassen ist `Producer` jedoch nicht bezüglich des Log-Datentyps parametrisiert, sondern bezüglich der Log-Kanäle, auf die über Referenzen zugegriffen werden kann. Im Zusammenspiel mit der nachfolgend beschriebenen Klasse `ChannelManager` werden diese Kanalreferenzen automatisch initialisiert und eindeutige Bezeichner vergeben. Da Kanäle hinsichtlich des Log-Datentyps parametrisiert sind, ist es allerdings auch möglich, Produzentenklassen zu generieren, die unterschiedliche Kanalarten verwenden. Das kann sinnvoll sein, wenn bestimmte Kategorien verschiedenartige Log-Datentypen benötigen, die durch spezielle Konsumenten verarbeitet werden sollen. In einem solchen Szenario ist die Zusammenarbeit mit einem `ChannelManager` eingeschränkt, weil dieser nur gleichartige Kanäle verwalten kann.

Auch die Verwendung der Klasse `ChannelManager` ist optional. Als Spezialisierung von `NameScope` kann diese Verwaltungsklasse als Gültigkeitsbereich für Produzentennamen dienen. Ihre weitere Funktionalität betrifft die zentrale Erzeugung und Verwaltung von Kanal-Objekten, die den gleichen durch `RecordT` vorgegebenen Log-Datentyp transportieren. Die Nutzung eines `ChannelManager`-Objekts ist immer dann sinnvoll, wenn eine Software viele Produzenten enthält, welche alle auf dieselben Log-Kanäle zugreifen. Durch einen solchen Manager wird sichergestellt, dass alle Produzenten einen eindeutigen Namen haben, ihre Kanalreferenzen korrekt initialisiert sind und diese bei allen Produzenten auf dieselben Kanal-Objekte verweisen.

3.3 Softwaretechnische Umsetzung

Dieser Teil der Arbeit erklärt die Implementation des im vorigen Abschnitt beschriebenen generischen Logging-Konzepts in der Programmiersprache C++. Dazu erfolgt im nächsten Abschnitt eine Beschreibung der wichtigsten Techniken und Werkzeuge, die für die Umsetzung in C++ benötigt werden. Anschließend werden die unterschiedlichen Nutzungsebenen der Logging-Bibliothek sowie die im vorigen Abschnitt aufgeführten Klassen im Detail beschrieben, gefolgt von mehreren Default-Komponenten, die in die Bibliothek integriert werden, um ihre unkomplizierte Nutzung zu ermöglichen.

3.3.1 Essentielle Werkzeuge

Standardkonformer C++-Compiler für Generische Programmierung

Der Begriff **generische Programmierung** bezeichnet ein Programmierparadigma, dessen Ziel die Entwicklung von generalisiertem, wiederverwendbarem Code ist. Hauptaugenmerk im Entwicklungsprozess sind dabei Gemeinsamkeiten ähnlicher Implementationen von Datenstrukturen oder Algorithmen, die eine Abstraktion von verwendeten Datentypen oder Werten erlauben, so dass eine generische Datenstruktur oder ein generischer Algorithmus extrahiert werden kann. Durch Typ-Parametrisierung und Substitution werden auf diese Weise mehrere konkrete Implementationen durch eine generische Version abgedeckt [Step88].

In C++ wird dieses Programmierparadigma durch den Template-Mechanismus unterstützt, der auf parametrisierten Code-Schablonen für Funktionen und Klassen basiert. Erst bei ihrer Nutzung mit konkreten Typ- oder Werteparametern generiert der Compiler Code für die entsprechenden Instanzen der Funktions- oder Klassenschablonen. Aufgrund der Komplexität des Template-Mechanismus wird allerdings ein moderner C++-Compiler benötigt, da viele C++-Compiler erst in den letzten Jahren einen hohen Grad an Standardkonformität erreicht haben.

Die C++-Standardbibliothek macht ausgiebig Gebrauch von diesem Mechanismus und stellt eine große Zahl parametrisierter Datenstrukturen und Algorithmen bereit. Am bekanntesten sind die sogenannten Container, deren Funktionalität unabhängig vom Typ der gespeicherten Daten ist, sowie verschiedene Algorithmen,

die auf Container angewendet werden können, wie eine Sortierung oder Transformation der Elemente [Vande02].

Auch im Logging-Konzept spielen Templates eine wichtige Rolle, um die Nutzbarkeit mit verschiedenen Log-Datentypen sowie unterschiedlichen Kanälen zu ermöglichen, wie Abbildung 3.3 zeigt. Mehrere Komponenten sind mit dem Typparameter `RecordT` versehen, der bestimmt, mit welchem Typ von Log-Daten die aus den Schablonen generierten Klassen arbeiten können. Der Parameter `EnableT` der Klasse `Producer` bestimmt dagegen, auf welche Kanäle ein Produzent Zugriff haben soll.

Erweiterungen der C++-Standardbibliothek

Im Jahr 2005 wurde der sogenannte C++ Technical Report 1 (TR1) veröffentlicht. Bei diesem Dokument handelt es sich um Vorschläge zur Erweiterung der C++-Standardbibliothek. Die darin beschriebenen Komponenten sind bisher nicht Teil des offiziellen Standards, sollen es jedoch in der nächsten Version werden. Die Erweiterungen betreffen viele verschiedene Themengebiete wie Container, Funktionsobjekte, Numerik, reguläre Ausdrücke und Speicherverwaltung. Moderne Compiler bieten Implementationen dieser Erweiterungen der Standardbibliothek schon jetzt an.

Zur Speicherverwaltung dienen unter anderem sogenannte **Smart Pointer**. Dabei handelt es sich um Objekte, welche sich wie Zeiger verhalten und dazu weitere Funktionalität anbieten. Ein Beispiel ist das Klassentemplate `std::auto_ptr`, mit dessen Hilfe die Lebenszeit eines dynamisch erzeugten Objekts verwaltet werden kann. Die Logging-Bibliothek benötigt an verschiedenen Stellen die Garantie, dass Objekte wie Konsumenten oder Filter mindestens solange existieren, wie ein anderes Objekt darauf zugreifen könnte. Da `std::auto_ptr` allerdings mehrere Limitierungen hat³, verwendet die Logging-Bibliothek an vielen Stellen das TR1-Klassentemplate `std::tr1::shared_ptr`. Dieser Smart Pointer überwacht die Lebenszeit eines Objekts durch das Zählen von Referenzen. Solange ein verwaltetes Objekt durch mindestens ein `shared_ptr`-Objekt referenziert wird, bleibt es beste-

³ Objekte des Typs `std::auto_ptr` können wegen ihrer Besitz-Semantik nicht mit Standardcontainern verwendet werden, und mehrere `auto_ptr` dürfen nicht auf dasselbe verwaltete Objekt verweisen, weil sonst mehrfach `delete` für dieses gerufen wird.

hen – erst wenn der Referenzzähler den Wert null erreicht, wird das referenzierte Objekt vernichtet und sein Speicher wieder freigegeben.

Unter den TR1-Containern finden sich auch **Tupel**. Das entsprechende Klassentemplate `std::tr1::tuple` ist eine Generalisierung von `std::pair`, mit dem sich n -Tupel beliebiger Datentypen erstellen lassen. Solche n -Tupel werden von der Logging-Bibliothek zur Realisierung nutzerdefinierter Typlisten verwendet. Im Rahmen der generischen Programmierung ist `tuple` ein nützliches Werkzeug, um während des Kompiliervorgangs Typberechnungen nach dem Parameterindex oder Iterationen über die Liste der Parametertypen durchzuführen. Zu diesem Zweck werden von der TR1-Bibliothek die Hilfstemplates `std::tr1::tuple_size` und `std::tr1::tuple_element` bereitgestellt.

Zur Erzeugung von Funktionsobjekten werden in TR1 gleich mehrere Möglichkeiten angeboten. All jene C++-Typen, die links vom Funktionsaufruf-Operator `()` stehen dürfen, werden als **aufrufbare Typen** bezeichnet. Viele Algorithmen der Standardbibliothek basieren auf der Übergabe von Objekten aufrufbarer Typen. Ein Beispiel sind Prädikate, die für ein übergebenes Argument wahr oder falsch zurückgeben. Diese aufrufbaren Typen müssen allerdings eine bestimmte Signatur (bestehend aus Parameterliste und Rückgabety) haben. In manchen Situationen ist es jedoch erforderlich, einen aufrufbaren Typ zu verwenden, der diese Signatur nicht erfüllt, weil er mehr Parameter verlangt. Dieses Problem kann mit dem Funktionstemplate `std::tr1::bind` gelöst werden. Diesem wird als erstes Argument das Zielobjekt (ein Objekt aufrufbaren Typs) übergeben, gefolgt von einer Liste weiterer Argumente, die beim Aufruf des durch `bind` generierten Funktors an das Zielobjekt weitergereicht werden. Der Funktor verwaltet diese zusätzlichen Argumente intern und bietet nach aussen die geforderte Signatur. Standardmäßig ist diese parameterlos. Durch Einfügen von speziellen Platzhaltern in der Argumentliste von `bind` lässt sich diese Signatur variieren, so dass problemlos Funktoren mit der benötigten Parameterzahl generiert werden können.

Die Bibliothekensammlung POCO

POCO (*portable components*) ist eine Sammlung quelloffener C++-Klassenbibliotheken, die insbesondere für netzwerkorientierte Anwendungen plattformunabhängige Komponenten zur freien Verfügung stellen [POCO]. Darunter finden sich nützliche Abstraktionen für die Anwendungsgebiete Netzwerkkommunika-

tion, Dateisystemzugriff, Datenbankinteraktion oder auch XML-Verarbeitung, welche die C++-Standardbibliothek bisher nicht bereitstellt. Die Logging-Bibliothek nutzt die Datenbank-Komponenten der POCO-Bibliotheken. In ODEMX finden außerdem die Dateisystem- und XML-Komponenten Verwendung.

3.3.2 Nutzungsebenen

Da die Basis-Komponenten der Logging-Bibliothek generischer Art sind, aber auch einige Default-Komponenten bereitgestellt werden, ergeben sich in der Anwendung der Logging-Bibliothek verschiedene Nutzungsebenen. Entsprechend der gewählten Ebene wird der Anwender durch unterschiedliche bereitgestellte Komponenten unterstützt. Diese bestimmen den Implementationsaufwand für den Bibliotheksnutzer. Tabelle 3.2 listet die Ebenen auf und fasst die angebotene Unterstützung sowie den Implementationsaufwand zusammen.

Nutzungsebene	Unterstützung	Implementationsaufwand
Basis-Komponenten	keine	Log-Datentypen, Konsumenten, ggf. Filter und Produzenten
Default-Komponenten	Log-Datentyp, Filter, Konsumenten	ggf. Produzenten und weitere Konsumenten
Datenbank-Zugriff	Datenbank-Zugriffsschicht	Konsumenten, Datenbankstruktur, ggf. Filter, Produzenten und Log-Datentypen
Komplettintegration	Logging-Bibliothek komplett integriert, alle Komponenten vordefiniert	Logging-Anweisungen

Tabelle 3.2: Nutzungsebenen bei Verwendung der Logging-Bibliothek

Die Ebene mit dem geringsten Abstraktionsgrad ist die direkte Verwendung der Basis-Komponenten der Logging-Bibliothek, also der C++-Templates. Auf dieser Ebene ist der Implementierungsaufwand hoch, da von der Bibliothek ohne Kenntnis der Templateparameter keine unterstützenden Komponenten angeboten werden können. Der Anwender muss also einen oder mehrere Log-Datentypen festlegen oder neu schreiben. Als Log-Datentyp kann grundsätzlich jeder Datentyp verwendet werden, auch einfache C++-Datentypen. Für jeden Log-Datentyp wird mindestens eine passende Konsumentenklasse benötigt, welche die Verarbeitung der Log-Einträge übernimmt. Optional sind Produzentenklassen und Filter-Spezialisierungen für die Log-Datentypen zu implementieren.

Die zweite Nutzungsebene betrifft die Implementation von Logging-Funktionalität durch Verwendung der von der Logging-Bibliothek bereitgestellten Default-Komponenten. Es wird ein Default-Datentyp für das Logging angeboten, auf dessen Grundlage weitere Default-Komponenten aufbauen. So stellt die Logging-Bibliothek für diesen Datentyp auch eine Filter-Spezialisierung und typische Konsumentenklassen für die Text- und XML-Ausgabe bereit. Gegebenenfalls ist auf dieser Ebene die Implementation von Produzenten- sowie weiteren Konsumentenklassen notwendig, die den Default-Log-Datentyp verwenden.

Die dritte Nutzungsebene wird bei Anbindung einer Datenbank genutzt. Die bereitgestellte Zugriffsschicht zur Interaktion mit der Datenbank unterstützt den Default-Log-Datentyp der Bibliothek durch einige Methoden. Seine Verwendung ist jedoch nicht zwingend verlangt. Zur Weiterleitung von Log-Daten an eine Datenbank ist in jedem Fall die Definition einer Konsumentenklasse erforderlich, welche die Zugriffsschicht verwendet, um die gewünschte Tabellenstruktur in der Datenbank zu erzeugen und die Log-Daten einzupflegen. Die Zugriffsschicht erleichtert diese Aufgabe lediglich und abstrahiert von der direkten Datenbankinteraktion, damit verschiedene RDBMS auf einheitliche Weise angebunden werden können. Die Erzeugung von Produzentenklassen bleibt auch in dieser Ebene optional. Wegen der Erzeugung einer Konsumentenklasse und der Unterstützung des Default-Log-Datentyps ergeben sich in dieser Ebene Schnittmengen mit der ersten und zweiten Ebene.

Die Integration der Logging-Bibliothek in einem Software-Projekt bildet die Ebene mit der höchsten Abstraktionsstufe. Prinzipiell kann innerhalb des anderen Projekts komplett von der Implementation des Loggings abstrahiert werden, indem Komponenten geschaffen werden, die das Logging bereits als Grundfunktionalität anbieten. Im Abschnitt 3.4 wird dieser Ansatz anhand der Integration der Logging-Bibliothek in ODEMX demonstriert. Bei Verwendung der ODEMX-Komponenten ist nichts weiter zu tun, als Logging-Anweisungen im Quelltext einzufügen und eine der angebotenen Ausgabekomponenten zu nutzen.

In den nächsten Abschnitten folgt nun eine detaillierte Beschreibung aller Komponenten der Logging-Bibliothek. Zum besseren Verständnis der Komponenten wird ihre Verwendung in möglichst kurzen Beispielen demonstriert. Jedes dieser Programme wird dabei einer der Nutzungsebenen zugeordnet, um zu verdeutlichen, welche Form der Bibliotheksunterstützung jeweils in Anspruch genommen wird. Die Ausgaben aller Beispielprogramme dieses Kapitels sind in Anhang D zu finden.

3.3.3 Kanäle, Konsumenten und Filter

Die Kernfunktionalität des Logging-Konzeptes wird durch die Log-Kanäle und die Konsumenten erbracht. Dabei übernimmt das Klassentemplate `Channel` die zentralisierte Verteilung von Log-Daten einer Kategorie. Jedes `Channel`-Objekt verwaltet seine registrierten Konsumenten in einer Menge von `shared_ptr`-Objekten um sicherzustellen, dass die referenzierten `Consumer`-Objekte mindestens so lange existieren wie das `Channel`-Objekt selbst. Kanäle können durch eine Identifikationsnummer (ID) des Typs `ChannelId` unterschieden werden. Zudem besitzt jedes `Channel`-Objekt noch einen weiteren `shared_ptr`, mit dem ein Filter für Log-Daten verwaltet werden kann. Der als Templateparameter angegebene Log-Datentyp bestimmt, mit welchen Arten von Konsumenten und Filtern ein Kanal interagieren kann.

Das ebenfalls bezüglich des Log-Datentyps parametrisierte Klassentemplate `Consumer` gibt eine Schnittstelle für all die Klassen vor, die Log-Daten verarbeiten sollen. Konsumentenklassen müssen von dieser Schnittstelle unter Angabe des von ihnen akzeptierten Log-Datentyps ableiten und dann die rein virtuelle Methode `consume` implementieren. Mit dieser simplen Schnittstelle wird bewusst das in Abschnitt 3.2.1 beschriebene Problem der überdimensionierten `TraceConsumer`-Schnittstelle vermieden. Daraus resultiert allerdings ein potenzieller Verlust von Informationen, die durch den Aufruf verschiedener Methoden an einer Konsumentenschnittstelle übermittelt werden. Alle relevanten Informationen müssen stattdessen aus Log-Einträgen zu gewinnen sein. Da ein Konsument an mehreren Log-Kanälen registriert werden und von allen Daten erhalten kann, wird der Methode `consume` neben dem Log-Eintrag auch die ID des weiterleitenden Kanals übergeben, so dass eine einzelne Konsumentenklasse die Log-Daten verschiedener Kategorien gesondert verarbeiten kann.

Filter werden in diesem Konzept durch das Klassentemplate `Filter` repräsentiert, welches die Methode `pass` vorgibt, der ein Log-Eintrag als Parameter übergeben wird. Die Implementation der Methode obliegt Spezialisierungen dieses Klassentemplates, die definieren müssen, wann ein Log-Eintrag passieren darf und wann nicht. Im ersten Fall muss `pass` `true` zurückgeben, im zweiten `false`.

Der Ausgabe-Operator `<<` wird in Analogie zu den Standard-Datenströmen überladen, um Log-Einträge C++-typisch über Kanäle senden zu können. Diese Überladung ist in Quelle 3.1 dargestellt. Da Operatoren in C++ lediglich Funktionen mit

spezieller Syntax sind, kann man sie auch bezüglich ihrer Argumenttypen parametrisieren, wodurch in diesem Fall ein Funktionstemplate für den Log-Datentyp `RecordT` entsteht.

```
template < typename RecordT >
const Channel< RecordT >&
operator<<( const Channel< RecordT >& channel, const RecordT& record )
{
    if( ! channel.hasFilter() || channel.getFilter()->pass( record ) )
    {
        for( typename Channel< RecordT >::ConsumerSet::const_iterator
            iter = channel.getConsumers().begin();
            iter != channel.getConsumers().end();
            ++iter )
        {
            ( *iter )->consume( channel.getId(), record );
        }
    }
    return channel;
}
```

Quelle 3.1: Operatorüberladung für `Channel` und beliebige Log-Datentypen

Beim Aufruf des Operators wird entsprechend des in Abbildung 3.2 gezeigten Logging-Vorgangs zunächst getestet, ob der Kanal einen Filter hat. Gegebenenfalls muss der Log-Eintrag diesen passieren, um an Konsumenten weitergeleitet zu werden. Die Weiterleitung geschieht durch Aufruf der Methode `consume` an jedem registrierten Consumer.

Es existiert eine weitere Überladung desselben Operators für `shared_ptr`-verwaltete `Channel`-Objekte, die zunächst die Gültigkeit des Zeigers testet. So kann das Logging zur Laufzeit für einzelne Kanäle einfach deaktiviert werden, indem der Kanalzeiger auf null gesetzt wird. Die Reaktivierung zu einem späteren Zeitpunkt ist durch erneutes Setzen des Zeigers möglich. Ist bereits die Erzeugung von Log-Einträgen rechenintensiv, kann ein solcher Zeiger auch vor dem Operatoraufruf überprüft werden, so dass erst gar kein Objekt für den Log-Eintrag erstellt wird.

Beispiel 1: Kanal und Konsument

Dieses Beispiel zeigt die Anwendung der Logging-Bibliothek auf der Nutzungsebene der Basis-Komponenten. Es veranschaulicht das Zusammenspiel zwischen

einem Log-Kanal und einer neu definierten Konsumentenklasse, die als Log-Datentyp `std::string`-Objekte verarbeitet. Ziel ist die Protokollierung von Strings auf der Standardausgabe. Dazu wird `std::string` in Quelle 3.2 als Log-Datentyp `Rec` festgelegt und die Konsumentenklasse `ConsoleWriter` definiert, welche die Methode `consume` so implementiert, dass alle Log-Einträge direkt auf die Standardausgabe geschrieben werden.

```
1 #include <CppLog.h>
2 #include <iostream>
3
4 typedef std::string Rec;
5
6 class ConsoleWriter: public Log::Consumer< Rec > {
7 public:
8     virtual void consume( const Log::ChannelId channelId, const Rec& record ) {
9         std::cout << record << std::flush;
10    }
11 };
12
13 int main() {
14     typedef std::tr1::shared_ptr< Log::Channel< Rec > > ChannelPtr;
15     typedef std::tr1::shared_ptr< Log::Consumer< Rec > > ConsumerPtr;
16     ConsumerPtr consoleWriter = ConsumerPtr( new ConsoleWriter() );
17
18     Log::Channel< Rec > channel;
19     channel.addConsumer( consoleWriter );
20     channel << Rec( "Logging via Channel object\n" );
21
22     ChannelPtr p_channel( new Log::Channel< Rec >() );
23     p_channel->addConsumer( consoleWriter );
24     p_channel << Rec( "Logging via set Channel pointer produces output\n" );
25     p_channel.reset();
26     p_channel << Rec( "Logging via empty pointer prints nothing\n" );
27     if( p_channel )
28         p_channel << Rec( "Checking empty pointer avoids record creation\n" );
29 }
```

Quelle 3.2: Verwendung von Consumer und Channel

In der `main`-Funktion wird als Konsument zunächst ein `shared_ptr`-verwaltetes `ConsoleWriter`-Objekt erzeugt. Dieses wird durch Ruf der Methode `addConsumer` an einem default-konstruierten `Channel`-Objekt registriert. Danach wird eine Nachricht an den Kanal gesendet, die dann auf der Standardausgabe zu sehen ist. Um die Zerstörung des `ConsoleWriter`-Objekts muss sich der Anwender nicht kümmern, da der `shared_ptr` es automatisch freigibt, sobald die letzte Kopie des Zeigers und damit die letzte Objekt-Referenz vernichtet wird.

Im zweiten Teil der `main`-Funktion wird die Verwendung von `shared_ptr`-verwalteten Log-Kanälen demonstriert. Auch hier muss zunächst ein Konsument registriert werden, wobei das Objekt `consoleWriter` erneut verwendet wird. Die erste über den Kanal-Zeiger verschickte Nachricht erscheint ebenfalls auf der Standardausgabe. Anschließend wird der Zeiger zurückgesetzt, wodurch die folgende Nachricht nicht mehr weitergeleitet wird. Auf diese Weise kann also das Logging an der Quelle deaktiviert werden ohne den Kanal zu beeinflussen. Ohne diese Möglichkeit müsste stattdessen ein Filter verwendet oder der Konsument entfernt werden, um die Ausgabe der Nachricht zu unterdrücken. Dabei wird allerdings immer noch das `Rec`-Objekt für den Aufruf des Operators erzeugt. Vermeiden lässt sich dies, indem der `shared_ptr` vor Ausführung der Logging-Anweisung auf Gültigkeit getestet wird, so wie es am Ende der `main`-Funktion dargestellt ist.

3.3.4 Identifizierbare Produzenten und Verwaltung von Kanälen

Für die Auswertung von Log-Daten kann es von Bedeutung sein, dass alle Quellobjekte eindeutig identifizierbar sind. In `log4cxx` besitzen alle Logger einen Namen und werden in einem zentralen Repository verwaltet. Auch `ODEMx` unterstützt eine ähnliche Funktionalität bereits seit der ersten Version durch die Schnittstelle `LabeledObject` und die Verwaltungsklasse `LabelScope`.

Angelehnt daran werden von der Logging-Bibliothek die Klassen `NamedElement` und `NameScope` angeboten. Klassen, deren Objekte eindeutige Bezeichner erhalten sollen, werden von `NamedElement` abgeleitet und benötigen als Konstruktorparameter eine Referenz auf ein `NameScope`-Objekt, das einen Gültigkeitsbereich für Bezeichner repräsentiert, in dem deren Eindeutigkeit gesichert ist. Bei Aufruf der Methode `NameScope::createUniqueName` wird geprüft, ob der angeforderte Name eindeutig ist. Existiert bereits ein Eintrag zu diesem Namen, so wird die Eindeutigkeit durch Anfügen eines Zählers gewahrt.

Zur zentralen Verwaltung mehrerer Kanäle gleichen Typs und der Vergabe eindeutiger Objektnamen wird das von `NameScope` abgeleitete Klassentemplate `ChannelManager` von der Bibliothek bereitgestellt. Durch den Templateparameter wird der Log-Datentyp von Kanälen und Filtern festgelegt, mit denen der Manager arbeiten soll. Diese Limitierung auf einen Log-Datentyp pro Managerklasse ist notwendig, damit die `Channel`-Objekte von dieser Klasse konstruiert und in

einem Container vorgehalten werden können. Außerdem können nur für gleichartige Kanäle Methoden angeboten werden, welche Konsumenten oder Filter für den gegebenen Log-Datentyp gleichzeitig an allen verwalteten Kanälen registrieren. Channel-Objekte werden mittels `shared_ptr` in einer `std::map` verwaltet, deren Schlüsseltyp die `ChannelId` der Log-Kanäle ist. Wird durch die Methode `getChannel` eine Kanal-ID zum ersten Mal vom Manager angefordert, so erstellt dieser das dazugehörige Channel-Objekt und speichert es in seiner `map`, um fortan immer einen Zeiger auf dasselbe Objekt zurückzugeben. Zudem werden mit `addConsumer`, `removeConsumer`, `setFilter` und `resetFilter` Methoden für das Hinzufügen und Entfernen von Konsumenten oder Filtern an den verwalteten Channel-Objekten angeboten, die entweder einzelne – per ID spezifizierte – oder alle Kanäle beeinflussen.

Das Klassentemplate `Producer` dient der Konfiguration von Produzenten-Basisklassen. Durch Ableitung von `NamedElement` sind alle Produzentenobjekte eindeutig anhand ihres Namens identifizierbar, wenn sie dem gleichen Gültigkeitsbereich angehören. Daher wird im Konstruktor neben der Angabe des gewünschten Bezeichners auch eine Referenz auf den `NameScope` verlangt. Zugriff auf Log-Kanäle wird in `Producer`-Spezialisierungen über `shared_ptr`-Objekte gewährt, damit die Existenz von Channel-Objekten solange gesichert ist, wie ein Produzent diese nutzen könnte. Der Templateparameter `EnableT` kontrolliert dabei, welche der mit Hilfe des Makros `CPPLOG_DECLARE_CHANNEL` deklarierten Kanäle der jeweiligen Produzentenklasse als `shared_ptr`-Member zugänglich gemacht werden.

CPPLOG_DECLARE_CHANNEL(namespace, name, record_type)

Das Makro dient der Deklaration von Log-Kanälen für Produzentenklassen, die auf dem Klassentemplate `Log::Producer` basieren. Dafür wird in einem Namensraum (`namespace`) eine ID mit dem gegebenen Bezeichner (`name`) angelegt. Weiterhin erzeugt das Makro eine Spezialisierung des Hilfstemplates `ChannelField`, das für den deklarierten Log-Kanal einen `shared_ptr` als Member mit dem Bezeichner `name` bereitstellt. Die eigentliche Aktivierung erfolgt dabei durch die Übergabe des Templates `Log::Enable< ID-Liste >` als Parameter von `Producer`. Von den durch `ID-Liste` angegebenen `ChannelField`-Spezialisierungen wird der Produzententyp abgeleitet, womit er auf diese Kanäle dann Zugriff hat.

`Producer` sind auf die Zusammenarbeit mit einem `ChannelManager` ausgelegt. Als Spezialisierung von `NameScope` stellt dieser ihnen einerseits einen Gültigkeitsbe-

reich für Namen bereit, und andererseits dient er als Erzeuger und Verwalter der verwendeten Log-Kanäle. So erhalten Producer-Objekte bei ihrer Konstruktion automatisch Zugriff auf die aktivierten Kanäle, wenn anstelle eines NameScope-Objekts ein ChannelManager übergeben wird. Die Initialisierung der shared_ptr-verwalteten Log-Kanäle des Produzenten erfolgt während der Konstruktion durch Aufruf der Methode `getChannel` des Managers, so dass gegebenenfalls Channel-Objekte erzeugt werden, wenn für die angeforderte ID noch keine Instanz vorhanden ist. Auf diese Weise wird sichergestellt, dass die von Produzenten referenzierten Channel-Objekte existieren und dass alle Produzenten, die mit dem gleichen ChannelManager konstruiert werden, auch die gleichen Channel-Objekte verwenden.

Beispiel 2: Produzent und Kanal-Manager

Dieses Beispiel zeigt erneut eine Anwendung der Logging-Bibliothek auf der Nutzungsebene der Basis-Komponenten. Es verdeutlicht in den drei Quellen 3.3, 3.4 und 3.5 die Verwendung der beiden Klassentemplates `Producer` und `ChannelManager`. Dafür werden zwei Log-Kanäle deklariert, die von einer Produzentenklasse genutzt und von einem Manager verwaltet werden.

```
1 #include <CppLog.h>
2 #include <iostream>
3
4 typedef std::pair< std::string, std::string > Rec;
5
6 CPPLOG_DECLARE_CHANNEL( channel, info, Rec );
7 CPPLOG_DECLARE_CHANNEL( channel, debug, Rec );
8
9 typedef Log::Producer< Log::Enable< channel::info, channel::debug > > ProducerType;
```

Quelle 3.3: Deklaration eines Log-Datentyps, zweier Log-Kanäle und eines Basistyps für Produzentenklassen

Zur Demonstration der Verwendung von Produzentennamen wird in Quelle 3.3 zunächst mit `std::pair` ein neuer Log-Datentyp festgelegt, der zwei Strings transportieren kann, nämlich den Namen des Senders und eine Textnachricht. Anschließend werden die beiden Log-Kanäle `info` und `debug` deklariert. Die Deklaration von Kanälen mit Hilfe eines Makros erspart dem Anwender die Spezialisierung der

zugrundeliegenden Hilfstemplates, und sie ermöglicht eine einfache Konfiguration von Produzentenklassen. Die darauf folgende Typdefinition für `ProducerType` zeigt, wie das Klassentemplate `Producer` verwendet wird: als Templateparameter wird das Klassentemplate `Enable` mit den IDs der zu aktivierenden Kanäle `info` und `debug` übergeben.

```
10 class DataProducer: public ProducerType {
11 public:
12     DataProducer( Log::ChannelManager< Rec >& mgr, const std::string& name )
13         : ProducerType( mgr, name ) {}
14     void useInfo() {
15         info << log( "Info logging with Producer and ChannelManager" );
16     }
17     void useDebug() {
18         debug << log( "Debug logging with Producer and ChannelManager" );
19     }
20 private:
21     Rec log( const std::string& text ) const { return Rec( getName(), text ); }
22 };
```

Quelle 3.4: Implementation einer Produzentenklasse, die durch Ableitung Zugriff auf die aktivierten Log-Kanäle erhält

Quelle 3.4 zeigt die Klasse `DataProducer`, welche durch Ableitung von `ProducerType` die aktivierten Kanäle `info` und `debug` verwenden kann. Da eine Referenz auf den `ChannelManager` im Konstruktor an die Basisklasse übergeben wird, erfolgt eine automatische Initialisierung der beiden `shared_ptr`, welche die Kanäle referenzieren. Die Methoden `useInfo` und `useDebug` demonstrieren die Nutzung von Log-Kanälen innerhalb einer Produzentenklasse. Da alle Log-Einträge den Namen des Absenders enthalten sollen, wird für ihre Erzeugung die private Methode `log` verwendet, die automatisch den Namen hinzufügt.

```
23 class ConsoleWriter: public Log::Consumer< Rec > {
24 public:
25     virtual void consume( const Log::ChannelId channelId, const Rec& record ) {
26         switch( channelId ) {
27             case channel::info: std::cout << "INFO "; break;
28             case channel::debug: std::cout << "DEBUG "; break;
29             default: break;
30         }
31         std::cout << "(" << record.first << ") " << record.second << std::endl;
32     }
33 };
34
```

```

35 int main() {
36     typedef std::tr1::shared_ptr< Log::Consumer< Rec > > ConsumerPtr;
37     Log::ChannelManager< Rec > manager;
38     DataProducer p1( manager, "P1" ), p2( manager, "P2" );
39     manager.addConsumer( ConsumerPtr( new ConsoleWriter() ) );
40     p1.useInfo();
41     p2.useDebug();
42 }

```

Quelle 3.5: Verwendung von Producer und ChannelManager

Der abschließende in Quelle 3.5 gezeigte Teil des Beispiels ähnelt dem Code von Beispiel 1. Für den neuen Log-Datentyp `std::pair` wird eine Konsumentenklasse `ConsoleWriter` definiert, die je nach Kanal-ID eine entsprechende Ausgabe erzeugt und anschließend Sendername und Textnachricht ausgibt. In der `main`-Funktion wird zunächst ein `ChannelManager` erzeugt, der dann bei der Konstruktion von zwei Produzenten mit unterschiedlichen Namen verwendet wird. Über den Manager wird derselbe `ConsoleWriter` durch Ruf von `addConsumer` an beiden Kanälen als Konsument registriert. Die Log-Ausgaben der abschließenden Methodenrufe erscheinen dann mit Kanal- und Sender-Angabe auf der Standardausgabe.

3.3.5 Ein Log-Datentyp zum Transport beliebiger Daten

Die in den vorangegangenen Abschnitten gezeigten Beispiele 1 und 2 offenbaren einen Nachteil in der Verwendung der Bibliothek auf der Nutzungsebene der Basis-Komponenten. Wenn beliebige Log-Datentypen genutzt werden, erfordert dies für jeden einzelnen Typ mindestens die Implementation einer neuen Konsumentenklasse, wie die beiden Varianten der Klasse `ConsoleWriter` demonstrieren. Dem Problem kann jedoch durch die Bereitstellung von Default-Komponenten begegnet werden. Dieser Abschnitt beschreibt nun die Grundlage für die Nutzungsebene der Default-Komponenten.

Bereits in Abschnitt 3.3.2 wurde erwähnt, dass die grundlegende Default-Komponente ein zur Bibliothek gehöriger Log-Datentyp ist. Seine Umsetzung findet dieser in der Klasse `Record`, die im Einklang mit der Philosophie der Bibliothek den Transport beliebiger Daten unterstützt. Diese Einschränkung durch Vorgabe eines möglichen Log-Datentyps erlaubt im Gegenzug die Integration weiterer Default-Komponenten in der Bibliothek, wodurch der Nutzungskomfort erhöht wird.

Die Klasse `Record` transportiert üblicherweise eine im Konstruktor übergebene Textnachricht, verlangt diese Angabe jedoch nicht zwingend. Zur Speicherung häufig verwendeter Kontextinformationen werden die Methoden `file`, `line` und `scope` angeboten, mit denen sich die genaue Quellcode-Position der Erzeugung des Log-Eintrags anfügen lässt.

`Record` ist abgeleitet von der Basisklasse `InfoHolder`, welche in einer Hashtabelle Objekte sogenannter Infotypen verwaltet. Ein **Infotyp** ist eine Spezialisierung des Klassentemplates `InfoType`. Eine derartige Spezialisierung dient lediglich dem Transport von Daten eines bestimmten Wertetyps der als Templateparameter angegebenen wird. Da es `InfoType`-Spezialisierungen für beliebige Wertetypen geben kann, ist es also möglich, Daten beliebigen Typs in `Record`-Objekten zu transportieren. Zur vereinfachten Erzeugung von Infotypen wird das Makro `CPPLOG_DECLARE_INFO_TYPE` angeboten.

```
CPPLOG_DECLARE_INFO_TYPE( type_name, string_name, value_type )
```

Das Makro dient der Erzeugung von Infotypen, die Objekte des Typs `Log::Record` dazu befähigen, Daten jeglicher Art zu transportieren. Die Infotypen kapseln Werte eines vorgegebenen Typs (`value_type`), wobei hier neben Kopien der Objekte auch Referenzen oder Zeiger verwendet werden können. Der Parameter `type_name` gibt dabei den Namen des Infotyps vor, unter dem `Record`-Objekte die Daten vorhalten. Abweichend davon kann mit dem Parameter `string_name` ein anderer Bezeichner angegeben werden, der bei der Extraktion transportierter Daten Verwendung findet. So wird beispielsweise die Zuordnung von Tabellenspalten einer Datenbank zu den extrahierten Daten ermöglicht.

```
namespace Log {  
    CPPLOG_DECLARE_INFO_TYPE( FileInfo, "File", StringLiteral );  
    CPPLOG_DECLARE_INFO_TYPE( LineInfo, "Line", unsigned int );  
    CPPLOG_DECLARE_INFO_TYPE( ScopeInfo, "Scope", StringLiteral );  
}
```

Quelle 3.6: Deklaration von Infotypen für den Transport von Kontext-Informationen in Log-Einträgen

Quelle 3.6 zeigt die Deklaration jener Infotypen, die von der Logging-Bibliothek zur Implementation der `Record`-Methoden `file`, `line` und `scope` verwendet werden. Damit Konsumenten auch über die Infotypen eines Log-Eintrags iterieren können, bietet die Klasse `Record` das Methodentemplate `iterateInfo` an. Als

Funktionsparameter wird ein Funktor mit einer bestimmten Signatur übergeben, der die in Infotypen vorgehaltenen Daten verarbeitet. Der Templateparameter von `iterateInfo` ist dabei eine Liste von Infotypen, deren Erstellung mit dem Makro `CPPLOG_DECLARE_INFO_TYPE_LIST` erfolgt.

```
CPPLOG_DECLARE_INFO_TYPE_LIST( type_list_name, ... )
```

Mit Hilfe dieses Makros werden Listen von Infotypen deklariert, die der automatischen Extraktion von Infotyp-Werten aus Record-Objekten dienen. Der erste Parameter spezifiziert den Namen der Typliste, gefolgt von einer variablen Anzahl von Infotypen, die mit dem Makro `CPPLOG_DECLARE_INFO_TYPE` erzeugt wurden. Die Implementation der Typlisten wird durch das Klassentemplate `std::tr1::tuple` realisiert. Compilezeit-Iterationen über die Elemente der Liste sind durch die rekursive Anwendung der Hilfstemplates `std::tr1::tuple_size` und `std::tr1::tuple_element` realisiert.

Beispiel 3: Default-Log-Datentyp Record

Dieses Beispiel behandelt einen weiteren Aspekt der Nutzungsebene der Basis-Komponenten. Es illustriert die Datenextraktion aus Record-Objekten. Anwender können mit diesem Wissen eigene Konsumentenklassen auf der Grundlage des Log-Datentyps `Record` implementieren. Werden hingegen die Default-Ausgabekomponenten der Logging-Bibliothek genutzt, so muss sich der Anwender nicht mit dem hier dargestellten manuellen Zugriff auf die Record-Daten befassen. Die Quellen 3.7 und 3.8 zeigen die Verwendung der Klasse `Record` für den Transport und den Zugriff auf verschiedene Infotypen, wobei an dieser Stelle die von der Logging-Bibliothek angebotenen Kontext-Infotypen `FileInfo`, `LineInfo` und `ScopeInfo` Anwendung finden.

```
1 #include <CppLog.h>
2 #include <iostream>
3 using namespace Log;
4
5 CPPLOG_DECLARE_INFO_TYPE_LIST( Infos, FileInfo, LineInfo, ScopeInfo );
6
7 struct InfoWriter {
8     template < typename ValT >
```

```
9     void operator()( const StringLiteral& name, const ValT& value, bool found ) {  
10         if( found ) { std::cout << ", " << name << ": " << value; }  
11     }  
12 } writer;
```

Quelle 3.7: Deklaration einer Liste von Info-Typen und Definition eines Ausgabefunktors, der bei Iterationen über die Liste verwendet wird

In Quelle 3.7 wird zunächst durch ein Makro die Infotyp-Liste `Infos` deklariert, mit deren Hilfe die Kontextinformationen während der Verarbeitung von `Record`-Objekten automatisch extrahiert werden können. Die Verarbeitung der Daten geschieht dabei durch Funktoren wie `InfoWriter`, der auch die geforderte Signatur zeigt. Der Parameter `found` gibt an, ob ein Log-Eintrag gültige Daten für den Infotypen enthält, `name` ist die String-Repräsentation des Infotyps, und `value` ist der transportierte Wert.

```
13 int main() {  
14     Record rec( "Logging class Record example" );  
15     rec.file( __FILE__ ).line( __LINE__ ).scope( "main()" );  
16  
17     std::cout << rec.getText();  
18     if( FileInfo::Type const* fi = rec.get< FileInfo >() )  
19         std::cout << ", " << FileInfo::getName() << ": " << *fi;  
20     if( LineInfo::Type const* li = rec.get< LineInfo >() )  
21         std::cout << ", " << LineInfo::getName() << ": " << *li;  
22     if( ScopeInfo::Type const* si = rec.get< ScopeInfo >() )  
23         std::cout << ", " << ScopeInfo::getName() << ": " << *si;  
24     std::cout << std::endl;  
25  
26     std::cout << rec.getText();  
27     rec.iterateInfo< Infos >( writer );  
28     std::cout << std::endl;  
29 }
```

Quelle 3.8: Einfügen und Extrahieren von Daten die in Objekten der Klasse `Record` transportiert werden

Die `main`-Funktion in Quelle 3.8 zeigt, wie Daten durch Methodenverkettung von `file`, `line` und `scope` an einem `Record`-Objekt hinzugefügt werden und wie transportierte Daten auf verschiedene Weise aus diesem Objekt extrahiert werden können. Als eine Möglichkeit lassen sich mit dem Methodentemplate `InfoHolder::get` Zeiger auf die einzelnen Daten anfordern – für den Fall, dass ein `Record`-Objekt den verlangten Infotyp nicht besitzt, wird ein Nullzeiger zurückgegeben, so dass die Gültigkeit der Daten getestet werden kann. Die zweite

Möglichkeit ist die automatische Extraktion anhand der Typliste Infos durch Aufruf von `iterateInfo` mit einem Funktorobjekt, wie es hier mit einem `InfoWriter`-Objekt demonstriert wird.

3.3.6 Filtern von Log-Daten

Das Filtern von Log-Daten wird durch Registrieren eines `Filter`-Objekts an einem Log-Kanal ermöglicht. Für den Default-Log-Datentyp `Record` gibt es eine Spezialisierung des Klassentemplates `Filter`, welche auf der Verwendung von Filtermengen basiert, in denen die als Filterkriterium von Log-Einträgen verwendeten Werte gesammelt werden. Standardmäßig wird eine Filtermenge für die Textnachrichten bereitgestellt, die mit Hilfe der Methode `addRecordText` gefüllt wird. Da alle weiteren Daten eines `Record`-Objekts durch Infotypen gekapselt sind, können Anwender beliebige weitere Filtermengen für Infotypen über das Methodentemplate `add` hinzufügen und füllen, wobei der entsprechende Infotyp als Templateparameter angegeben wird. Der Wertetyp des jeweiligen Infotyps muss allerdings sortierbar sein, weil die Filter-Einträge in Objekten des Typs `std::set` vorgehalten werden, die eine effiziente Suche ermöglichen. Die Methode `resetFilter` löscht alle Filtermengen und setzt alle Einstellungen auf den Initialzustand des Filters zurück.

Ein globaler Filtermodus kann mit den Methoden `passAll` und `passNone` eingestellt werden. Diese Modi bestimmen, ob der Filter standardmäßig alle Log-Einträge passieren lässt oder alle abblockt. Im Fall von `passAll` werden dementsprechend nur jene Log-Einträge herausfiltert, für die ein Treffer in einer Filtermenge gefunden wird, im Fall `passNone` hingegen dürfen nur diejenigen Log-Einträge passieren, für die ein passender Wert in einer Filtermenge enthalten ist.

Die von der Filter-Schnittstelle vorgegebene Methode `pass` testet beim Aufruf zunächst, ob der Filter aktiv ist. Dies ist der Fall, wenn der Filtermodus verändert oder ein Wert hinzugefügt wurde. Anschließend wird der Text des übergebenen Log-Eintrags in der entsprechenden Filtermenge gesucht. Wurde keine Übereinstimmung festgestellt, so wird über die Filtermengen für Infotypen iteriert. Das Resultat der Methode hängt dann davon ab, ob in einer Filtermenge ein Treffer gefunden wird, und welcher Filtermodus gerade aktiv ist.

Beispiel 4: Default-Filter für Record

Dieses Beispiel demonstriert einen Aspekt der Verwendung der Logging-Bibliothek auf der Nutzungsebene der Default-Komponenten, für die die Logging-Bibliothek eine Filterimplementation und verschiedene Ausgabekomponenten bereitstellt. In Quelle 3.9 wird der Einsatz eines Filters an einem Log-Kanal gezeigt. Die Klassentemplates `Channel` und `Filter` werden mit ihrem Default-Templateparameter `Record` instanziiert. Wie oben beschrieben, können Log-Einträge dieses Typs mit Hilfe der Spezialisierung `Filter<Record>` anhand ihrer Textnachricht oder der Infotyp-Werte gefiltert werden. In diesem Beispiel wird dazu der von der Bibliothek vorgegebene Infotyp `ScopeInfo` genutzt.

```
1 #include <CppLog.h>
2 #include <iostream>
3 using namespace Log;
4
5 CPPLLOG_DECLARE_INFO_TYPE_LIST( Infos, ScopeInfo );
6 Channel<> channel;
7
8 void foo() {
9     channel << Record( "Logging filter example (foo)" ) + ScopeInfo( "foo()" );
10 }
11
12 int main() {
13     std::tr1::shared_ptr< Filter<> > filter = Filter<>::create();
14     filter->add< ScopeInfo >() << "foo()";
15     channel.setFilter( filter );
16     channel.addConsumer( OStreamWriter< Infos >::create( std::cout ) );
17
18     channel << Record( "Logging filter example (main)" ) + ScopeInfo( "main()" );
19     foo();
20 }
```

Quelle 3.9: Verwendung der Templatespezialisierung `Filter<Record>`

In den beiden Funktionen `foo` und `main` wird jeweils über ein globales `Channel`-Objekt ein Log-Eintrag mit der Kontextinformation `ScopeInfo` abgeschickt, die in diesem Fall den Funktionsnamen enthält. In der `main`-Funktion wird ein Filter erzeugt, der Log-Einträge mit dem `ScopeInfo`-Wert „foo()“ aussortieren soll. Dieser Filter wird am Log-Kanal registriert, ebenso wie ein Konsument, der Log-Einträge im Textformat an die Standardausgabe weiterleitet⁴. Als Resultat des Programm-

⁴ Die Beschreibung des Klassentemplates `OStreamWriter` erfolgt im nächsten Abschnitt

aufrufs erscheint auf der Standardausgabe lediglich der von der `main`-Funktion erzeugte Log-Eintrag, da der in `foo` erzeugte Log-Eintrag gefiltert wird.

3.3.7 Textbasierte Ausgabekomponenten

Auf der Grundlage der Klasse `Record` können mehrere textbasierte Ausgabekomponenten von der Logging-Bibliothek angeboten werden. Bei Nutzung des Default-Log-Datentyps kann die Logging-Bibliothek direkt und ohne großen Aufwand verwendet werden, anstatt wie in den Beispielen 1 und 2 die Definition einer Konsumentenklasse zu verlangen. Die Transportmöglichkeit für beliebige Daten durch `Record`-Objekte erfordert allerdings die Implementation dieser Komponenten als Klassentemplates. Mit dem Templateparameter `InfoListT` können Anwender eine Liste der Infotypen spezifizieren, die aus jedem Log-Eintrag zu extrahieren sind. Der Default-Templateparameter ist eine leere Liste, womit auch einfaches Loggen von Zeichenketten ermöglicht wird. Abbildung 3.4 zeigt die Hierarchie der bereitgestellten Ausgabekomponenten.

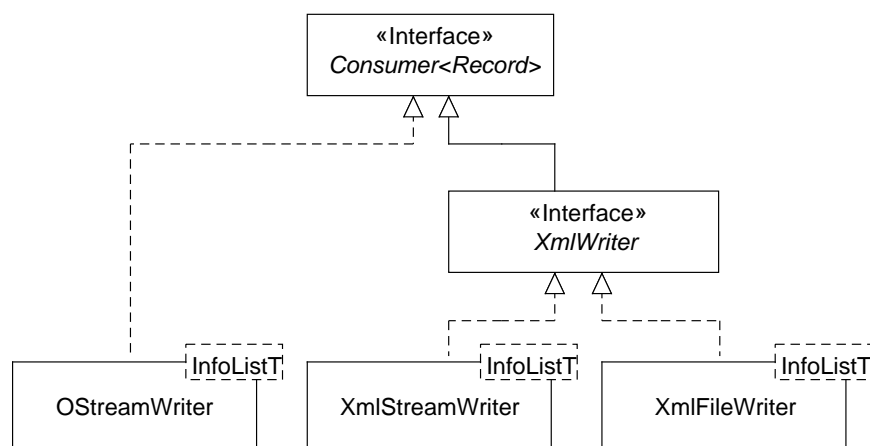


Abbildung 3.4: Hierarchie der textbasierten Ausgabekomponenten für den Log-Datentyp `Record`

Das Klassentemplate `OStreamWriter` implementiert direkt die Konsumentenschnittstelle `Consumer<Record>`. Zur Initialisierung dieser Ausgabekomponente verlangt der Konstruktor die Angabe einer Referenz auf ein `std::ostream`-Objekt.

Das kann beispielsweise ein geöffneter Dateiausgabestrom sein oder auch die Standardausgabe, wie das vorangegangene Beispiel 4 zeigt. Die Methode `consume` leitet dann die aus den Log-Einträgen extrahierten Informationen im String-Format an den Ausgabestrom weiter. Die Konvertierung der Daten geschieht durch die von der Standardbibliothek bereitgestellten Ausgabeoperatoren. Demzufolge muss für alle transportierten nutzerdefinierten Datentypen der Operator `<<` definiert sein. Anderenfalls lässt sich das Programm nicht übersetzen.

Weiterhin werden von der Logging-Bibliothek zwei Komponenten für die XML-Ausgabe angeboten. Als Zwischenschritt wird dafür die Schnittstelle `XmlWriter` in die Klassenhierarchie eingefügt, welche von der Schnittstelle `Consumer<Record>` abgeleitet ist, diese jedoch nicht implementiert. `XmlWriter` stellt lediglich Hilfsmittel zur Erzeugung des XML-Formats zur Verfügung. Zum einen sind dies statische Methoden und zum anderen ein Funktor analog zu dem in Beispiel 3 gezeigten `InfoWriter` (siehe Quelle 3.7), mit dessen Hilfe die Daten aus Infotypen extrahiert und formatiert werden.

Die Implementation der Konsumentenschnittstelle erfolgt erst durch die von `XmlWriter` abgeleiteten Klassentemplates `XmlStreamWriter` und `XmlFileWriter`. Der `XmlStreamWriter` ist dabei analog zum `OStreamWriter` implementiert, wobei die Textausgabe im XML-Format erfolgt. Allerdings werden nur die `record`-Elemente und ihre Kindelemente ausgegeben. Sollten XML-Header und ein Wurzelement benötigt werden, so können diese durch Aufruf der statischen Methoden von `XmlWriter` in den Datenstrom eingefügt werden.

Erweiterte Funktionalität bietet der `XmlFileWriter`, der immer abgeschlossene XML-Dateien mit Header und Wurzelement erzeugt. Im Konstruktor werden der Basisdateiname, der Name des Wurzelements und die maximale Anzahl von Log-Einträgen pro Datei festgelegt. Sobald dieses Limit erreicht ist, wird die aktuelle Datei abgeschlossen und eine neue begonnen. An den Basisdateinamen wird dabei jeweils ein Zähler angehängt, um die Dateireihenfolge und die Eindeutigkeit der Namen zu sichern.

Die drei beschriebenen Klassentemplates bieten alle eine `create`-Methode an, welche die gleichen Parameter erwartet wie der entsprechende Konstruktor. Damit wird ein `shared_ptr`-verwaltetes Objekt erzeugt, das direkt als Konsument an einem Kanal registriert werden kann.

Beispiel 5: Default-Ausgabekomponenten für Record

Dieses Beispiel demonstriert wie schon Beispiel 4 eine Anwendung der Logging-Bibliothek auf der Nutzungsebene der Default-Komponenten. Quelle 3.10 veranschaulicht die Verwendung der textbasierten Ausgabekomponenten der Logging-Bibliothek. Dabei wird auch die Ausgabe nutzerdefinierter Datenstrukturen behandelt.

```

1  #include <CppLog.h>
2  #include <iostream>
3
4  struct Point3d { float x, y, z; };
5
6  std::ostream& operator<<( std::ostream& os, Point3d const& p ) {
7      os << "(" << p.x << " " << p.y << " " << p.z << ")";
8      return os;
9  }
10
11  CPPLOG_DECLARE_INFO_TYPE( PointInfo, "point", Point3d );
12  CPPLOG_DECLARE_INFO_TYPE( DoubleInfo, "value", double& );
13  CPPLOG_DECLARE_INFO_TYPE_LIST( Infos, PointInfo, DoubleInfo );
14
15  int main() {
16      Log::Channel<> channel;
17      channel.addConsumer( Log::OStreamWriter< Infos >::create( std::cout ) );
18      channel.addConsumer( Log::XmlStreamWriter< Infos >::create( std::cout ) );
19      channel.addConsumer( Log::XmlFileWriter< Infos >::create( "Example", 1000 ) );
20
21      Point3d p = { 1, 2, 3 };
22      channel << Log::Record() + PointInfo( p ) + DoubleInfo( 123.45 );
23  }

```

Quelle 3.10: Verwendung der textbasierten Default-Ausgabekomponenten

Zunächst wird die Struktur `Point3d` und eine entsprechende Überladung des Ausgabeoperators `<<` definiert, die für die Konvertierung der Datenstruktur in einen String sorgt. Anschliessend folgt die Deklaration der Infotypen `PointInfo` und `DoubleInfo`, die in der Liste `Infos` zusammengefasst werden. Damit wird den Ausgabekomponenten signalisiert, welche Informationen aus Log-Einträgen extrahiert werden sollen. Danach wird in der `main`-Funktion ein `Channel`-Objekt für den Default-Log-Datentyp `Record` angelegt, an dem ein `OStreamWriter`, ein `XmlStreamWriter` und ein `XmlFileWriter` als Konsumenten registriert werden. Der durch die letzte Anweisung erzeugte Log-Eintrag erscheint dann als einfacher Text und als XML auf der Standardausgabe. Der `XmlFileWriter` erzeugt außerdem eine XML-Datei namens `Example_0.xml`, die den Log-Record enthält.

3.3.8 Anbindung an relationale Datenbanken

RDBMS bieten Programmierern für den Datenbankzugriff meist die Wahl zwischen einer systemspezifischen Programmierschnittstelle (API) oder einer standardisierten API wie zum Beispiel Open Database Connectivity (ODBC). Während die erste Lösung auf die Anbindung eines bestimmten RDBMS ausgelegt ist und dafür die Nutzung systemspezifischer Optimierungen erlaubt, bietet die zweite Variante Entwicklern die Möglichkeit, ihre Software an beliebige RDBMS anzubinden, wenn diese über einen ODBC-Treiber verfügen. Da bei der Logging-Bibliothek insbesondere auf die Austauschbarkeit von Komponenten Wert gelegt wird, sollte sie die Auswahl des RDBMS nicht beschränken und verwendet daher ODBC zur Kommunikation mit externen RDBMS.

Eingebettete RDBMS können dagegen direkt in andere Software integriert werden. So kann die Logging-Bibliothek bereits ein eigenes RDBMS anbieten, womit dem Anwender die Installation und Konfiguration eines RDBMS erspart werden kann. Ein populäres und weit verbreitetes RDBMS für diesen Zweck ist SQLite [SQLite], das sich bereits in vielen Anwendungsbereichen bewährt hat⁵. Damit kann die gesamte Datenbasis einer Anwendung innerhalb einer Datei verwaltet werden, die in einem portablen, plattformunabhängigen Format gespeichert wird. Die Wahl des eingebetteten RDBMS fiel zudem auch deshalb auf SQLite, da diese Bibliothek als quelloffene C-Bibliothek frei verfügbar ist, sich in aktiver Entwicklung befindet und häufige Aktualisierungen erfährt.

C++-Bibliotheken für den Zugriff auf Datenbanken

Zur vereinfachten Nutzung von RDBMS gibt es in der Programmiersprache Java Klassenbibliotheken, die eine objektrelationale Abbildung (*object-relational mapping*, ORM) von Objekten auf eine relationale Datenbank ermöglichen, wodurch ein Anwendungsprogramm seine Objekte transparent in einer relationalen Datenbank ablegen kann. Mit der Java Persistence API existiert eine standardisierte Schnittstelle dafür. In C++ gibt es bislang dagegen keine ausgereiften Bibliotheken, mit deren Hilfe sich ORM nutzen ließe. Daher wird an dieser Stelle diskutiert,

⁵ Die Skriptsprachen PHP und Python sind standardmäßig mit SQLite gebündelt. Namhafte Firmen und Entwicklergruppen wie Adobe, Apple, Google, Mozilla und Sun verwenden SQLite in verschiedenen Anwendungen zum Speichern von Konfigurations- oder Anwendungsdaten [SQLiteAnw].

durch welche C++-Bibliotheken eine Kommunikation mit externen RDBMS via ODBC und idealerweise auch mit SQLite umgesetzt werden kann.

Die bereits im Kapitel 3.1.3 erwähnte C++-Bibliothekensammlung Boost enthält aktuell keine Bibliothek für diesen Zweck. Die populärsten quelloffenen und vom RDBMS unabhängigen C++-Bibliotheken sind folgende:

- libODBC++ [LibODBC++]
- Database Template Library (DTL) [DTL]
- SOCI - The C++ Database Access Library [SOCI]
- Oracle, ODBC and DB2-CLI Template Library (OTL) [OTL]
- POCO Data [POCO]

Alle aufgelisteten Bibliotheken unterstützen mindestens die RDBMS-Anbindung via ODBC-API. Allerdings ist bei libODBC++, DTL und SOCI gemessen an der Häufigkeit der Aktualisierungen entweder eine Stagnation in der Entwicklung oder nur wenig Aktivität festzustellen. Sowohl OTL als auch POCO Data werden dagegen kontinuierlich aktualisiert, und es gibt in beiden Fällen aktive Foren, in denen die Entwickler um Rat gebeten werden können – ein klarer Vorteil, wenn einmal Probleme mit der genutzten Bibliothek auftreten sollten. Sowohl OTL als auch POCO Data unterstützen die Anbindung an externe RDBMS via ODBC-Schnittstelle. Allerdings enthält POCO Data standardmäßig SQLite, wodurch Anwendungen, die auf dieser Bibliothek aufsetzen, direkt SQLite-Datenbankdateien erstellen, lesen und schreiben können. Zudem bietet POCO Data eine einheitliche Abstraktionsschicht für die Verwendung von ODBC und SQLite. Für den Wechsel zwischen beiden APIs muss lediglich ein sogenannter Konnektor ausgetauscht werden. Die OTL hat dagegen den Vorteil, mit dem Format einer Header-Datei sehr leicht integrierbar zu sein. Ihre Nutzung hätte in diesem Projekt jedoch eine doppelte Implementierung der Datenbankbindung zur Folge, da nur die ODBC-Schnittstelle der OTL genutzt werden könnte. SQLite müsste separat über das native C-API eingebunden werden. Aufgrund dieser Betrachtungen bildet die Bibliothek POCO Data die Grundlage der Implementation der mit dieser Arbeit vorliegenden Datenbankbindung der Logging-Bibliothek.

Einen Nachteil haben allerdings alle aufgeführten Bibliotheken: In jedem Fall ist die direkte Eingabe von SQL-Kommandos im Quellcode notwendig. Das kann einerseits problematisch sein, wenn verschiedene RDBMS unterschiedliche Dialekte von SQL unterstützen, da der Code dann nicht unbedingt portabel ist. Andererseits

hat der C++-Compiler keine Kenntnis von SQL, weshalb der Anwender immer selbst dafür verantwortlich ist, dass SQL-Fragmente in C++-Strings syntaktisch korrekt sind. Durch eine einfache Abstraktionsschicht lassen sich beide Probleme eingrenzen.

Eine Zugriffsschicht für Datenbanken

Mit der Klasse `DatabaseAccessor` stellt die Logging-Bibliothek eine Zugriffsschicht bereit, die beiden im vorigen Abschnitt beschriebenen Problemen zumindest beim Schreiben von Log-Daten in eine Datenbank entgegenwirken soll. Wegen der unterschiedlichen SQL-Dialekte bietet diese Klasse dem Anwender eine einfache C++-Fassade auf der Grundlage von Tabellen, deren Struktur der Anwender vorgibt. So kann die Datenbankausgabe an beliebige Log-Datentypen angepasst werden. Da kein einheitliches Mapping zwischen C++-Datentypen und SQL-Datentypen existiert, bleibt es jedoch dem Nutzer überlassen, passende SQL-Datentypen für einzelne Tabellenspalten vorzugeben. Der eigentliche Datenbankzugriff via SQL wird innerhalb der Klasse `DatabaseAccessor` implementiert, so dass dem Anwender SQL-Kommandos in C++-Strings verborgen bleiben.

Obwohl ODBC für viele Systeme verfügbar ist, kann man nicht davon ausgehen, dass auch ODBC-Header-Dateien installiert sind, welche für die Übersetzung von Programmen mit ODBC-Komponenten benötigt werden. Daher wird der Wechsel zwischen ODBC und SQLite in der Implementation der Klasse `DatabaseAccessor` mit Hilfe von Präprozessor-Direktiven ermöglicht (siehe Anhang A). Standardmäßig wird SQLite als RDBMS verwendet, so dass ODBC-Komponenten nur dann auf einem System installiert sein müssen, wenn tatsächlich ein externes Datenbanksystem genutzt werden soll.

Die Klasse `DatabaseAccessor` implementiert eine auf das Logging ausgerichtete Zugriffsschicht für RDBMS. Sie ermöglicht im Wesentlichen die Erzeugung von Tabellen in einer Datenbank und das Einpflegen neuer Datensätze. Bei Erzeugung eines `DatabaseAccessor`-Objekts verlangt der Konstruktor die Angabe eines Verbindungsstrings, der zum Aufbau einer Datenbanksitzung verwendet wird. Die Umsetzung dieser Aufgabe wird an die Klassen `Poco::Data::Connector` und `Poco::Data::Session` delegiert. Mit Hilfe der ersten Klasse werden entsprechende Session-Objekte für SQLite- oder ODBC-Verbindungen erzeugt. Diese Aufgabe wird jedoch intern von der Bibliothek POCO Data erledigt.

Über das Session-Objekt interagiert ein DatabaseAccessor-Objekt mit dem RDBMS. Mittels Aufruf der Methode `createTable` kann in einer Datenbank eine Tabelle angelegt werden. Die Methode gibt ein temporäres Objekt der Klasse `SqlTableCreator` zurück, welches intern die SQL-Anweisung konstruiert und diese unter Verwendung der Klasse `Poco::Data::Statement` ausführt. Die nachfolgenden Beispiele 6 und 7 illustrieren die Verwendung.

SqlTableCreator

Diese Hilfsklasse kapselt die Erstellung von CREATE TABLE-Anweisungen. Ihre Nutzung erfolgt wie oben beschrieben indirekt durch Aufruf von `DatabaseAccessor::createTable`. Die interne SQL-Anweisung wird durch Methodenrufe an dem temporären Objekt konstruiert. Die Methode `column` fügt eine Spaltendefinition mit Name und SQL-Datentyp hinzu. Mit der Methode `primaryKey` kann der Primärschlüssel der Tabelle als Bedingung angefügt werden. Im Destruktor wird durch Ruf von `execute` die Ausführung der SQL-Anweisung eingeleitet. Bei Verwendung des Log-Datentyps `Record` können mehrere Spalten mit dem Methodentemplate `columns` eingefügt werden. Dies geschieht anhand einer Liste von Infotypen und eines Arrays mit den entsprechenden SQL-Datentypen. Dessen Erzeugung wird durch das nachfolgend beschriebene Makro `CPPLOG_DECLARE_SQL_COLUMN_TYPES` unterstützt.

CPPLOG_DECLARE_SQL_COLUMN_TYPES(list_type_name, array_name, ...)

Dieses Makro kommt bei der Erzeugung von Datenbanktabellen für den Log-Datentyp `Record` zur Anwendung. Zu der gegebenen Infotypliste (`list_type_name`) wird ein der Listenlänge entsprechendes Array von Strings erzeugt, dessen Bezeichner durch den zweiten Parameter (`array_name`) festzulegen ist. An letzter Stelle folgt eine variable Parameterliste, die analog zur Deklaration der Infotypliste die jeweils zugeordneten SQL-Datentypen aufnimmt und damit ein `std::tr1::array` initialisiert.

Das Einpflegen eines Datensatzes in eine Datenbank geschieht durch die Methode `insertIntoTable`. Sie arbeitet ähnlich wie `createTable`, denn hier wird ebenfalls ein temporäres Objekt – vom Typ `SqlInsertter` – zur Konstruktion und Ausführung einer SQL-Anweisung verwendet. Beispiel 6 zeigt die Anwendung. Diese komfortable Lösung verursacht allerdings Kosten zur Laufzeit des Programms, da bei hohem Datenaufkommen viele temporäre `SqlInsertter`-Objekte erzeugt werden und deren interne SQL-Anweisung immer neu geparkt werden muss. Daher gibt es mit der Methode `createInsertter` auch die Möglichkeit, `SqlInsertter`-Objekte für die mehrfache Verwendung zu erzeugen. Dies wird in Beispiel 7 demonstriert.

SqlInserter

Diese Hilfsklasse kapselt die Erstellung von INSERT-Anweisungen. Auch hier wird die interne SQL-Anweisung durch Methodenrufe konstruiert. Sowohl die indirekte als auch die direkte Verwendung werden unterstützt. Typisch für die indirekte Verwendung sind Rufe der Methode `columnValue`, welche ein Paar aus Spaltenname und zu speicherndem Wert zur SQL-Anweisung hinzufügt. Bei direkter Nutzung wird die SQL-Anweisung nur einmal mit Platzhaltern für die Werte konstruiert. Dazu dienen die Methoden `table` und `column`. Zur wiederholten Ausführung mittels `execute` müssen dann lediglich die Wertbindungen durch Rufe der Methode `value` ausgetauscht werden. Im Destruktor wird `execute` nur dann gerufen, wenn die vorgehaltene SQL-Anweisung noch nicht ausgeführt wurde. Für den Log-Datentyp `Record` werden die Methodentemplates `columns`, `values` und `columnValues` bereitgestellt, die auf der Grundlage von Infotyplisten mehrere Spalten oder Werte hinzufügen.

Die Klasse `SqlInserter` nutzt zur Kommunikation mit dem RDBMS eine vorbereitete SQL-Anweisung, ein sogenanntes **Prepared Statement**, welches von der Datenbank angefordert und mehrfach verwendet werden kann. Bei der erneuten Verwendung wird dann nur die Datenbindung ausgetauscht. In Laufzeittests wurden Versionen der Klasse mit und ohne Prepared Statement miteinander verglichen⁶. Dafür wurde das in Beispiel 7 dargestellte Programm zum Einfügen von 1.000.000 Log-Einträgen modifiziert. Die Laufzeiten betrugen 55,4 Sekunden ohne Prepared Statement verglichen mit 48,0 Sekunden unter Verwendung von einem Prepared Statement, was einer Laufzeitverbesserung von mehr als 13 Prozent entspricht.

Das Methodentemplate `DatabaseAccessor::storeData` bietet die Möglichkeit, den gesamten Inhalt eines Datenpuffers innerhalb einer Transaktion in die Datenbank zu schreiben. Dieses Vorgehen hat sich als sinnvoll erwiesen, da das Puffern von Log-Daten im Hauptspeicher die Zahl der Ein-/Ausgabeoperationen verringert und so erhebliche Geschwindigkeitsvorteile bringt. Umgesetzt wird dies, indem durch die SQL-Anweisung `BEGIN TRANSACTION` eine Transaktion gestartet wird. Darauf folgt das Einpflegen des gesamten Puffers durch den wiederholten Aufruf einer nutzerdefinierten Ausgabefunktion. Abgeschlossen wird die Transaktion durch die SQL-Anweisung `COMMIT`. Als erster Aufrufparameter muss dabei ein vorwärts-iterierbarer Container übergeben werden, und der zweite Parameter muss ein Funktionszeiger oder ein Funktor sein, der mit den Containerelementen als Parameter aufrufbar ist. Die Anforderungen an die Parameter erklären sich

⁶ Testsystem: Pentium M mit 2,1 GHz, Ubuntu-Linux, g++-Version 4.2.4

durch die interne Verwendung des C++-Standardalgorithmus `std::for_each`. Die Anwendung des Methodentemplates wird in Beispiel 6 gezeigt.

Zusätzlich verfügt die Klasse `DatabaseAccessor` noch über verschiedene Hilfsmethoden wie `tableExists`, `getMaxColumnValue` und `getCurrentDatabaseTime`, die im Hinblick auf ODEmx zum Abfragen essentieller Datenbank-Informationen bereitgestellt werden. Die Intention ist dabei nicht die Konstruktion einer umfangreichen Datenbankschnittstelle, die möglichst viel SQL-Funktionalität nachbildet, sondern eine Begrenzung auf das Wesentliche. So können die existierenden Methoden als Vorlage für Erweiterungen dienen, falls diese in der Zukunft erforderlich werden.

Beispiel 6: Verwendung der Datenbank-Zugriffsschicht

Dieses Beispiel zeigt die Nutzungsebene des Datenbank-Zugriffs. In den Quellen 3.11 und 3.12 wird demonstriert, wie die Datenbank-Zugriffsschicht zu verwenden ist. Zu diesem Zweck wird ein globales `DatabaseAccessor`-Objekt angelegt, das mit der SQLite-Datenbankdatei `example.db` interagiert.

```
1 #include <CppLog.h>
2 #include <string>
3 #include <utility>
4 #include <vector>
5 using namespace std;
6
7 Log::DatabaseAccessor db( "example.db" );
8
9 void createTable() {
10     if( ! db.tableExists( "example_table" ) ) {
11         db.createTable( "example_table" )
12             .column( "id", "INTEGER" )
13             .column( "text", "VARCHAR" )
14             .primaryKey( "id" );
15     }
16 }
17
18 void insertRow( const pair< int, string >& record ) {
19     db.insertIntoTable( "example_table" )
20         .columnValue( "id", record.first )
21         .columnValue( "text", record.second );
22 }
```

Quelle 3.11: Verwendung der Methoden `tableExists`, `createTable` und `insertIntoTable`

In Quelle 3.11 wird zunächst das globale Datenbank-Zugriffsobjekt `db` angelegt, welches als Verbindungsstring den Namen der Datenbankdatei erhält. Sollte diese nicht existieren, so wird sie automatisch erzeugt. Die danach definierte Funktion `createTable` überprüft zuerst, ob die Tabelle `example_table` bereits in der Datenbank existiert. Ist dies nicht der Fall, so wird sie mittels `DatabaseAccessor::createTable` angelegt. Die Tabellendefinition von `example_table` umfasst die Spalten `id` und `text`, denen jeweils ein adäquater SQL-Datentyp zugeordnet wird. Als Primärschlüssel der Tabelle wird das Attribut `id` festgelegt.

Es folgt die Definition der Funktion `insertRow`, welche die Methode `DatabaseAccessor::insertIntoTable` verwendet, um den als Parameter angegebenen Log-Eintrag des Typs `std::pair` als Datensatz in die Datenbank `example_table` einzupflegen. Mittels `columnValue` werden dabei jeweils der Spaltenname und der assoziierte Wert angegeben⁷.

```
23 int main() {  
24     createTable();  
25     int maxId;  
26     db.getMaxColumnValue( "example_table", "id", maxId );  
27  
28     vector< pair< int, string > > buffer;  
29     buffer.push_back( make_pair( maxId + 1, "first_example_string" ) );  
30     buffer.push_back( make_pair( maxId + 2, "second_example_string" ) );  
31  
32     db.storeData( buffer, &insertRow );  
33 }
```

Quelle 3.12: Verwendung der Methoden `getMaxColumnValue` und `storeData`

In der `main`-Funktion des Beispielprogramms wird als erstes die Funktion `createTable` gerufen, um sicherzustellen, dass die Tabelle `example_table` existiert. Anschließend wird der aktuell höchste Primärschlüsselwert mittels `getMaxColumnValue` abgefragt und in `maxId` gespeichert. Ein Vektor wird hier exemplarisch als Datenpuffer verwendet, und zwei Einträge mit Schlüsselwert und String-Nachricht werden hinzugefügt. Die letzte Anweisung demonstriert die Anwendung des Methodentemplates `DatabaseAccessor::storeData`. Dabei wird neben dem Puffer ein Zeiger auf die Funktion `insertRow` übergeben, welche in

⁷ Diese Methodenrufe modifizieren ein temporäres `SqlInserter`-Objekt, das die damit konstruierte SQL-Anweisung bei seiner Destruktion ausführt

storeData für jeden Eintrag des Puffers gerufen wird. Auf diese Weise werden die gepufferten Daten innerhalb einer Transaktion in die Datenbank eingepflegt.

Beispiel 7: Konsument mit Datenbankanbindung

Dieses Beispiel zeigt erneut die Nutzungsebene des Datenbank-Zugriffs. In den Quellen 3.13 bis 3.17 wird die Erstellung einer kompletten Konsumentenklasse demonstriert, welche die Klassen DatabaseAccessor und SqlInsertter zum Zugriff auf eine SQLite-Datenbank verwendet. Die empfangenen Log-Daten werden dabei in einem `std::vector` gepuffert und bei Erreichen einer Obergrenze automatisch in die Datenbank eingepflegt.

```

1  #include <CppLog.h>
2  #include <iostream>
3  using namespace Log;
4
5  CPPLOG_DECLARE_INFO_TYPE( I1, "info_1", std::string );
6  CPPLOG_DECLARE_INFO_TYPE( I2, "info_2", int );
7  CPPLOG_DECLARE_INFO_TYPE( I3, "info_3", float );
8  CPPLOG_DECLARE_INFO_TYPE_LIST( Infos, I1, I2, I3 );
9
10 class SqliteWriter: public Consumer<> {
11 public:
12     SqliteWriter( const std::string& dbName, std::size_t bufferLimit )
13         : db_( dbName ), bufferLimit_( bufferLimit ),
14           recordCount_( 0 ), initialized_( false )
15     {}
16     void flush();
17 private:
18     virtual void consume( const ChannelId channelId, const Record& record );
19     void initialize();
20     void insert( const Record& record );
21 private:
22     DatabaseAccessor db_;
23     std::auto_ptr< DatabaseAccessor::SqlInsertter > inserter_;
24     std::vector< Record > buffer_;
25     std::size_t bufferLimit_, recordCount_;
26     bool initialized_;
27 };

```

Quelle 3.13: Deklaration einer Konsumentenklasse mit Datenbankanbindung

Zunächst werden in Quelle 3.13 mehrere Infotypen deklariert und in der Infotypliste Infos zusammengefasst. Anschließend erfolgt die Deklaration der Konsumentenklasse `SqliteWriter`, die das Speichern von Log-Daten in einer SQLite-

Datenbank ermöglicht. Der Zugriff auf die Datenbank erfolgt dabei durch die privaten Member `db_` und `inserter_`. Zur Effizienzsteigerung werden Log-Daten in einem `std::vector` zwischengespeichert, wobei die Puffergrenze bei der Konstruktion festgelegt wird.

```
28 void SqliteWriter::consume( const ChannelId channelId, const Record& record ) {
29     if( ! initialized_ ) initialize();
30
31     if( recordCount_ < bufferLimit_ ) {
32         ++recordCount_;
33     } else {
34         flush();
35         recordCount_ = 0;
36     }
37     buffer_.push_back( record );
38 }
```

Quelle 3.14: Implementation der Methode `consume`

Die Implementation der Konsumentenschnittstelle ist in Quelle 3.14 dargestellt. Beim ersten Aufruf der Methode `consume` wird die Initialisierung des Konsumenten durch die Methode `initialize` vorgenommen. Neue Log-Einträge werden solange zum Puffer hinzugefügt, wie die Puffergrenze nicht erreicht ist. Der Zähler `recordCount_` überwacht dies, und bei Erreichen des Limits werden die vorgehaltenen Log-Daten durch Aufruf der Methode `flush` in die Datenbank eingepflegt und der Puffer geleert.

```
39 CPPLOG_DECLARE_SQL_COLUMN_TYPES( Infos, sqlTypes, "VARCHAR", "INTEGER", "REAL" );
40
41 void SqliteWriter::initialize() {
42     if( ! db_.tableExists( "log_record" ) ) {
43         db_.createTable( "log_record" )
44             .column( "id", "INTEGER" )
45             .column( "text", "VARCHAR" )
46             .columns< Infos >( sqlTypes )
47             .primaryKey( "id" );
48     }
49     inserter_ = db_.createInserter();
50     inserter_>table( "log_record" )
51         .column( "text" )
52         .columns< Infos >();
53     initialized_ = true;
54 }
```

Quelle 3.15: Implementation der Methode `initialize`

Die für die Initialisierung verantwortliche Methode `initialize` wird in Quelle 3.15 gezeigt. Das an dieser Stelle genutzte Präprozessor-Makro `CPPLOG_DECLARE_SQL_COLUMN_TYPES` wird für das automatische Erzeugen von Tabellenspalten anhand einer Infotypliste verwendet. Es deklariert das Array `sqlTypes` und initialisiert dieses mit den SQL-Datentypen der Tabellenspalten, die der Infotypliste Infos entsprechen. Verwendung findet dieses Array beim Aufruf von `createTable` am `DatabaseAccessor`-Objekt. Im Anschluss daran wird durch die Methode `DatabaseAccessor::createInserter` ein `SqlInserter`-Objekt erzeugt, dessen Prepared Statement für die Nutzung mit der Tabelle `log_record` konfiguriert wird.

```

55 void SqliteWriter::flush() {
56     using namespace std::tr1;
57     db_.storeData( buffer_, bind( &SqliteWriter::insert, this, placeholders::_1 ) );
58     buffer_.clear();
59 }
60
61 void SqliteWriter::insert( const Record& record ) {
62     inserter_>value( record.getText().c_str() )
63         .values< Infos >( record )
64         .execute();
65 }

```

Quelle 3.16: Implementation der Methoden `flush` und `insert`

Das Einpflegen der Log-Daten in die Datenbank geschieht mit Hilfe der in Quelle 3.16 dargestellten Methoden. Die öffentliche Methode `SqliteWriter::flush` delegiert diese Aufgabe unter Nutzung der Methode `SqliteWriter::insert` an `DatabaseAccessor::storeData`. Da Methoden einer Klasse einen `this`-Zeiger benötigen und `insert` somit nicht die passende Signatur für `storeData` hat, wird mit Hilfe von `std::tr1::bind` ein entsprechender Funktor generiert, der den Zeiger zwischenspeichert. Das letzte Argument von `bind` ist hier ein Platzhalter für den `Record`-Parameter von `insert`. Diese Methode nutzt den `SqlInserter`, um die Werte einzelner Log-Einträge an dessen internes Prepared Statement zu binden, so dass beim Aufruf von `execute` die entsprechenden Tabellenspalten mit diesen Werten gefüllt werden.

```

66 int main() {
67     typedef std::tr1::shared_ptr< SqliteWriter > SqlitePtr;
68     SqlitePtr sqlite( new SqliteWriter( "Example_Consumer_Sqlite.db", 1000 ) );
69
70     Channel<> channel;
71     channel.addConsumer( sqlite );

```

```
72     channel << Record( "Logging example: SQLite database access" )
73                     + I1( "string detail" ) + I2( 2 ) + I3( 3.0f );
74
75     try { sqlite->flush(); }
76     catch( const Poco::Exception& ex ) { std::cout << ex.displayText(); }
77 }
```

Quelle 3.17: Anwendung der Klasse SqliteWriter

Den Abschluss dieses Beispiels bildet Quelle 3.17, welche in der `main`-Funktion die Verwendung der Konsumentenklasse `SqliteWriter` demonstriert. Dafür wird ein `shared_ptr` mit einem `SqliteWriter`-Objekt initialisiert, dem der Name der verwendeten Datenbankdatei und eine Puffergröße vorgegeben werden. Weiterhin wird ein `Channel`-Objekt für den Default-Log-Datentyp `Record` erzeugt und der `SqliteWriter` daran als Konsument registriert. Der daraufhin über den Kanal versendete Log-Eintrag findet so seinen Weg in den Puffer des `SqliteWriter`-Objekts und wird durch expliziten Aufruf der Methode `flush` in die Datenbank geschrieben. Dieser letzte Aufruf wird durch eine `try-catch`-Anweisung gekapselt, da die verwendete Bibliothek `POCO Data` in Fehlersituationen Ausnahmen wirft.

3.4 Integration der Logging-Bibliothek in ODEMX

Eine Zielsetzung für die Logging-Bibliothek ist die leichte Integration in andere Software. In dieser Hinsicht kann man sich an Projekten wie `Boost` orientieren, welche heutzutage viele Komponenten als sogenannte **header-only Bibliotheken** bereitstellen, womit zur Verwendung lediglich die Header-Dateien benötigter Komponenten eingebunden werden müssen. Das separate Übersetzen des Quellcodes oder die Installation und das Verlinken von übersetzten Bibliotheken entfällt dabei komplett. Deshalb ist die Logging-Bibliothek ebenfalls in diesem Format verfügbar, so dass ihre Integration in ODEMX lediglich das Kopieren des Quellcodes in ein Unterverzeichnis erfordert – ein geeigneter Ort ist das Modul `External`, welches bereits in Abschnitt 2.1.2 Erwähnung fand. Des Weiteren wird mit `Data` ein neues ODEMX-Modul eingeführt, das alle Komponenten aufnehmen soll, die der Datenerzeugung, -erfassung und -auswertung dienen. Die Komponenten der Logging-Bibliothek bilden fortan die Grundlage für speziell an die Simulation mit ODEMX angepasste Logging- und Auswertungsklassen. In diesem Abschnitt werden nun die Spezialisierungen einzelner Klassen der Logging-Bibliothek sowie ihre Inte-

gration in die bestehende Klassenhierarchie von ODEMX dargelegt. Auf ähnliche Weise sind auch Anpassungen für andere C++-Projekte denkbar.

3.4.1 Der Log-Datentyp `SimRecord`

In ODEMX wird mit der Klasse `odemx::data::SimRecord` ein speziell auf Simulationsdaten zugeschnittener Log-Datentyp eingeführt. Als Quelle von Log-Daten sind in ODEMX Spezialisierungen einer Produzentenklasse vorgesehen, welche die Identifikation von Senderobjekten ermöglicht. ODEMX erlaubt außerdem die parallele Verwendung mehrerer Simulationskontexte innerhalb eines C++-Programms, so dass der Konstruktor der Klasse `SimRecord` neben der Beschreibung des Simulationsereignisses zusätzlich Referenzen auf den jeweiligen Produzenten und den aktuellen Simulationskontext verlangt. Diese simulationsspezifischen Log-Einträge transportieren daher immer mindestens folgende Informationen:

- ein String-Literal zur Beschreibung des Simulationsereignisses,
- einen Zeiger auf den Simulationskontext,
- einen Zeiger auf den Produzenten des Log-Eintrags und
- die Simulationszeit der Erstellung des Log-Eintrags.

Diese Informationen werden direkt im Konstruktor eines `SimRecord`-Objekts gesetzt. Die Methode `scope` erlaubt zudem die optionale Angabe eines Klassenkontextes, aus dem der Log-Eintrag versendet wurde. Diese Information ist insbesondere in größeren Klassenhierarchien hilfreich, da so die eindeutige Zuordnung von Log-Einträgen zu verschiedenen Basisklassen des Senderobjekts möglich ist.

Bereits in Abschnitt 3.2.1 wurde die Problematik der beliebig verschachtelbaren Detailhierarchien in Einträgen des bisherigen Trace-Mechanismus erörtert. Daher wird stattdessen für den Transport von weiteren Detailinformationen lediglich eine Detailebene erlaubt, die beliebig viele Informationen als Name-Wert-Paare in einem `std::vector` aufnehmen kann. Zum Hinzufügen von Detailinformationen werden die Methoden `detail` und `valueChange` angeboten, die ebenso wie `scope` die Methodenverkettung unterstützen, wie das folgende Beispiel illustriert. Der lesende Zugriff auf die oben aufgelisteten Daten eines `SimRecord`-Objekts erfolgt über die Methoden `getText`, `getSimulation`, `getSender` und `getTime`. Die Existenz optionaler Informationen wie Klassenkontext und Details kann mittels

hasScope und hasDetails überprüft werden. Der Zugriff erfolgt dann durch die Methoden getScope und getDetails.

3.4.2 Erzeugung und Verteilung von Log-Daten

Log-Kanäle und Logging-Management

Mit der Festlegung des Log-Datentyps auf die Klasse SimRecord wird folglich der Transport und die Verteilung von Log-Daten in ODEMX durch entsprechende Log-Kanäle des Typs `Log::Channel<SimRecord>` realisiert. Alle von ODEMX angebotenen Log-Kanäle haben eindeutige IDs, da sie mit Hilfe des Präprozessor-Makros `CPPLOG_DECLARE_CHANNEL` erzeugt werden. In Anlehnung an die in Abschnitt 3.2.2 aufgelisteten Logging-Kategorien werden in ODEMX folgende Kanal-IDs angeboten:

- `odemx::data::channel_id::trace`
- `odemx::data::channel_id::debug`
- `odemx::data::channel_id::info`
- `odemx::data::channel_id::warning`
- `odemx::data::channel_id::error`
- `odemx::data::channel_id::fatal`
- `odemx::data::channel_id::statistics.`

Eine String-Repräsentation der Kanalnamen kann bei Angabe der ID durch die Funktion `odemx::data::channel_id::toString` abgefragt werden. Da auch Anwender in ihren Programmen weitere Log-Kanäle für den Typ `SimRecord` definieren können, gibt diese Funktion bei unbekannten IDs „user-defined“ zurück.

Verwaltet werden diese standardmässig von ODEMX bereitgestellten Log-Kanäle von der Klasse `odemx::data::LoggingManager`, die eine Spezialisierung von `Log::ChannelManager<SimRecord>` ist. Die Klasse `LoggingManager` initialisiert mit ihrem Konstruktor die ODEMX-eigene Fehlerausgabe, indem an den Log-Kanälen `warning`, `error` und `fatal` standardmässig ein Konsument registriert wird, der Warnungen und Fehler an die Fehlerausgabe weiterleitet. Weiterhin bietet die Klasse die Methoden `enableDefaultLogging` und

`disableDefaultLogging` an, mit denen intern verwaltete Konsumenten an allen Log-Kanälen registriert beziehungsweise entfernt werden können. Durch Angabe eines Parameters ist die Wahl zwischen Standardausgabe, XML-Dateien oder Datenbankankbindung freigestellt. Der Wert `STDOUT` bestimmt dabei, dass Log-Daten an die Standardausgabe weitergeleitet werden sollen. Die Angabe der Werte `DATABASE` oder `XML` verlangt zusätzlich einen zweiten Parameterwert, der dem Verbindungsstring beziehungsweise dem Basisdateinamen entspricht. Wenn das Default-Logging aktiv ist, werden auch statistische Daten gesammelt, die mittels `reportDefaultStatistics` ausgegeben und durch `resetDefaultStatistics` zurückgesetzt werden können. Um die Einbettung der Klasse in ODEMX zu verdeutlichen ist der entsprechende Ausschnitt der Klassenhierarchie in Abbildung 3.5 dargestellt.

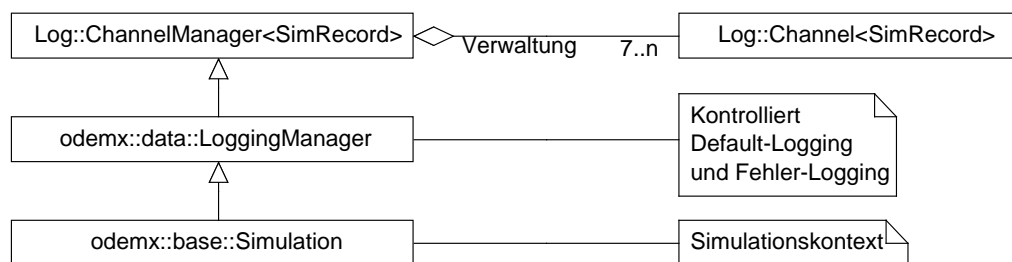


Abbildung 3.5: Ausschnitt der Klassenhierarchie von ODEMX, der die Einbettung von Logging-Manager und Log-Kanälen zeigt

Produzenten von Log-Daten

Bereits im vorangegangenen Abschnitt wurde erwähnt, dass ODEMX grundsätzlich Spezialisierungen einer Produzentenklasse als Quelle von Log-Daten verwendet. Die Klasse `odemx::data::Producer` ist die Basisklasse für alle ODEMX-Datenproduzenten, die ihrerseits auf dem Klassentemplate `Log::Producer` basiert. Dadurch erhalten alle Produzenten-Objekte eindeutige Bezeichner und verfügen über interne `shared_ptr` für die oben aufgelisteten Log-Kanäle. Bei Konstruktion der Klasse `Producer` werden diese automatisch durch Zugriff auf den `LoggingManager` initialisiert und können dann innerhalb aller abgeleiteten Klassen zum Versenden von Log-Einträgen des Typs `SimRecord` verwendet werden. Mit den Methoden `enableLogging` und `disableLogging` wird die Aktivierung beziehungsweise Deaktivierung des Loggings einzelner Objekte ermöglicht. Implementiert ist

dies durch Setzen oder Löschen der internen `shared_ptr`, welche die Log-Kanäle referenzieren (siehe Abschnitt 3.3.3 auf Seite 39). Lediglich die Kanäle `warning`, `error` und `fatal` bleiben immer aktiv, da sie für die Ausgabe ODEmX-interner Warnungen oder Fehler benötigt werden. So lässt sich eine genaue Kontrolle über die Erzeugung von Log-Daten einzelner Simulationselemente erreichen, ohne auf einen Filter angewiesen zu sein.

Die `SimRecord`-Objekte für Log-Einträge werden durch die Methode `log` erzeugt, welche lediglich die Angabe eines String-Literals als Parameter erwartet und automatisch die aktuelle Simulationszeit und Zeiger auf den Produzenten und den Simulationskontext zum Log-Eintrag hinzufügt. Zur einfachen Erzeugung von Log-Einträgen für die Statistik werden zusätzlich die Methoden `param`, `count`, `update` und `reset` angeboten, die Log-Einträge in einem festgelegten Format produzieren, das Statistik-Konsumenten verarbeiten können. Mit `param` wird ein Parameter mit Name und Wert protokolliert, `count` vereinfacht das Zählen eindeutig bezeichneter statistisch relevanter Ereignisse, mit `update` können Wertänderungen gemeldet werden, und `reset` zeigt das Zurücksetzen aller Statistikwerte eines Produzenten an. Diese letzte Methode ist nützlich, wenn statistische Daten gepuffert werden, weil so beispielsweise Einschwingphasen durch die Löschung irrelevanter Daten ignoriert werden können.

Die Methoden `getLabel`, `getType` und `getSimulation` implementieren die gleiche Funktionalität wie sie die Produzenten-Schnittstelle `TraceProducer` des bisherigen Trace-Mechanismus vorgibt. Sie dienen dem Zugriff auf den Bezeichner, den Klassentyp und den Simulationskontext eines Produzenten. Mit Version 3.0 der Bibliothek ODEmX ersetzt die Klasse `Producer` die alte Schnittstelle. Abbildung 3.6 zeigt einen Ausschnitt der Klassenhierarchie von ODEmX, um die Einbettung der Basisklasse `odemx::data::Producer` zu verdeutlichen. Die Klasse `odemx::sync::Queue` ist dabei ein Beispiel für eine Produzentenklasse.

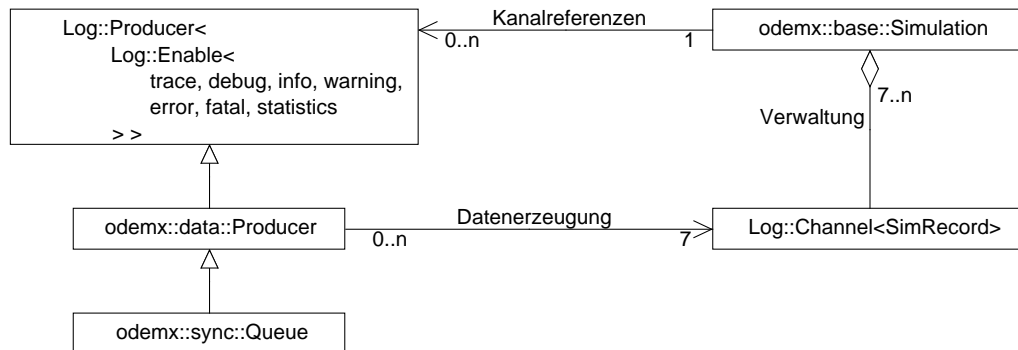


Abbildung 3.6: Ausschnitt der Klassenhierarchie von ODEmX, der die Einbettung der Produzentenbasisklasse `odemx::data::Producer` zeigt

Beispiel 8: ODEmX-basierter Produzent mit Logging-Nutzung

Dieses Beispiel zeigt die Verwendung der Logging-Bibliothek auf der Nutzungsebene der Komplettintegration. Es veranschaulicht in den Quellen 3.18 und 3.19 die Erstellung einer ODEmX-basierten Produzentenklasse sowie die typische Verwendung von ODEmX-Log-Kanälen anhand der Repräsentanten `info` und `error`. Dabei muss sich der Anwender mit keinerlei Implementationsdetails der zugrundeliegenden generischen Logging-Bibliothek auseinandersetzen.

```

1 #include <odemx/odemx.h>
2
3 class Stopwatch: public odemx::data::Producer {
4 public:
5     Stopwatch( odemx::base::Simulation& sim, const odemx::data::Label& label )
6         : Producer( sim, label ), running( false )
7     {}
8     void start() {
9         running = true;
10        startTime = getSimulation().getTime();
11        info << log( "started" );
12    }
13    void stop() {
14        if( running ) {
15            running = false;
16            info << log( "stopped" ).detail( "duration", getDuration() );
17        } else {
18            error << log( "StopWatch::stop(): stop watch is not running" );
19        }
20    }
21    odemx::base::SimTime getDuration() { return getSimulation().getTime() - startTime; }
    
```

```
22 private:
23     bool running;
24     odemx::base::SimTime startTime;
25 };
```

Quelle 3.18: Implementation einer nutzerdefinierten Produzentenklasse, die das Versenden von Log-Daten über zwei verschiedene Kanäle zeigt

In Quelle 3.18 wird die Produzentenklasse `StopWatch` definiert, deren Konstruktor die Angabe des Simulationskontextes und eines Bezeichners verlangt. Da die Klasse `Simulation` ein `Log::ChannelManager` ist, werden die Log-Kanäle der Basisklasse `Producer` automatisch initialisiert und die Eindeutigkeit des angeforderten Bezeichners innerhalb des Simulationskontextes gesichert. Die Methode `start` verwendet den Log-Kanal `info`, um den Beginn einer Zeitmessung anzuzeigen. Durch die Methode `stop` wird die Stoppuhr angehalten und das Ereignis via `info` protokolliert. Die gemessene Zeitdauer wird durch Aufruf von `SimRecord::detail` als Detail des Log-Eintrags festgehalten. Wird die Uhr gestoppt ohne aktiviert zu sein, so erfolgt die Anzeige einer Fehlermeldung über den Kanal `error`.

```
26 int main() {
27     odemx::base::Simulation& sim = odemx::getDefaultSimulation();
28     sim.enableDefaultLogging( odemx::data::output::STDOUT );
29
30     StopWatch watch( sim, "Stop Watch" );
31     watch.start();
32     sim.setCurrentTime( 10 );
33     watch.stop();
34
35     watch.disableLogging();
36     watch.stop();
37 }
```

Quelle 3.19: Nutzung der Klasse `StopWatch` mit `Simulation` als Manager

Quelle 3.19 zeigt die `main`-Funktion des Beispielprogramms, in der der Default-Simulationskontext verwendet wird. Da dieser eine Spezialisierung der Klasse `LoggingManager` ist, kann durch Aufruf der Methode `enableDefaultLogging` auch das Default-Logging aktiviert werden, welches in diesem Fall Daten an die Standardausgabe weiterleitet. Nach der Aktivierung des Loggings wird ein `StopWatch`-Objekt erzeugt, das mit dem Simulationskontext und einem Bezeichner initialisiert und sofort gestartet wird. Durch Erhöhung der Simulationszeit zwischen den Aufrufen von `start` und `stop` ergibt sich eine gemessene

Dauer von 10 Zeiteinheiten, die protokolliert wird. Der anschließende Aufruf von `disableLogging` deaktiviert an dem `StopWatch`-Objekt alle außer den fehlerbezogenen Log-Kanälen. Daher wird die Fehlermeldung, welche aus dem Aufruf der Methode `stop` an der inaktiven Stoppuhr resultiert, trotzdem angezeigt.

3.4.3 Filtern von Log-Daten

Grundlage für das Filtern des ODEMX-Log-Datentyps `SimRecord` ist eine Spezialisierung des Klassentemplates `Log::Filter`, deren Implementation analog zu der in Abschnitt 3.3.6 beschriebenen Spezialisierung `Filter<Record>` mit Filtermengen realisiert ist. So gibt es jeweils eine Filtermenge für Textnachrichten, Produzentennamen, Produzententypen und Klassenkontexte, die mit Hilfe der Methoden `addRecordText`, `addProducerLabel`, `addProducerType` und `addRecordScope` gefüllt werden können. Das Zurücksetzen des Filters auf den Initialzustand erfolgt durch die Methode `resetFilter`. Damit verfügt dieser Filter über eine ähnliche Funktionalität wie die im bisherigen Trace-Mechanismus verwendete Klasse `TraceFilter`.

Entsprechend der durch `Log::Filter` vorgegebenen Schnittstelle wird das eigentliche Filtern von der Methode `pass` umgesetzt, welche die Filtermengen auf Übereinstimmungen mit den im Log-Eintrag enthaltenen Daten durchsucht. Wird in einer Filtermenge der gleiche Wert wie im Log-Eintrag gefunden, so hängt die Weiterleitung des Log-Eintrags vom Filtermodus ab. Ist dieser auf `passAll` gesetzt, so werden alle Log-Einträge weitergeleitet, für die keine Übereinstimmung vorhanden ist. Bei `passNone` hingegen werden nur diejenigen Log-Einträge weitergeleitet, für die ein Treffer in einer Filtermenge gefunden wird. Standardmäßig ist im Konstruktor der Filtermodus `passAll` eingestellt. Zur Wahrung der Konsistenz in Namensgebung und Funktionalität ist die Klasse `odemx::data::SimRecordFilter` von der Template-Spezialisierung `Filter<SimRecord>` abgeleitet und ergänzt dabei eine `create`-Methode, die ein `shared_ptr`-verwaltetes Filterobjekt erzeugt, welches direkt an einem Kanal registriert werden kann.

Beispiel 9: Filtern des ODEMX-Log-Datentyps `SimRecord`

Dieses Beispiel zeigt wie schon Beispiel 8 die Verwendung der Logging-Bibliothek auf der Nutzungsebene der Komplettintegration. Anhand der Quellen 3.20 und

3.21 wird die Verwendung der Klasse `SimRecordFilter` verdeutlicht. Die in Beispiel 8 definierte Klasse `StopWatch` (siehe Quelle 3.18) wird an dieser Stelle als Basisklasse wiederverwendet, von der eine Spezialisierung `DebugStopWatch` definiert wird, um die Anwendung verschiedener Filter-Kriterien zu demonstrieren.

```
1  #include "Example_Producer_StopWatch.h"
2  #include <iostream>
3  using namespace odemx;
4
5  class DebugStopWatch: public StopWatch {
6  public:
7      DebugStopWatch( base::Simulation& sim, const data::Label& label ): StopWatch(sim,label) {
8          debug << log( "construction" ).scope( typeid( DebugStopWatch ) );
9      }
10     ~DebugStopWatch() {
11         debug << log( "destruction" ).scope( typeid( DebugStopWatch ) );
12     }
13     void start() {
14         debug << log( "started" ).scope( typeid( DebugStopWatch ) );
15         StopWatch::start();
16     }
17     void stop() {
18         debug << log( "stopped" ).scope( typeid( DebugStopWatch ) );
19         StopWatch::stop();
20     }
21 };
```

Quelle 3.20: Definition der Produzentenklasse `DebugStopWatch`, die zur Filter-Demonstration Log-Daten über den Kanal `debug` versendet

In Quelle 3.20 wird zunächst die Klasse `DebugStopWatch` gezeigt, welche die gleichen Methoden wie ihre Basisklasse `StopWatch` anbietet und diese um Debug-Ausgaben erweitert. Auf diese Weise können neben dem Beginn und dem Ende der Zeitmessung auch die Konstruktion und die Destruktion der Objekte protokolliert werden. Alle von der Klasse `DebugStopWatch` versendeten Log-Einträge transportieren zusätzlich zu der Textnachricht noch ihren Klassenkontext, welcher durch Aufruf der Methode `SimRecord::scope` hinzugefügt wird.

```
22 int main() {
23     typedef std::tr1::shared_ptr< data::SimRecordFilter > FilterPtr;
24     FilterPtr filter = data::SimRecordFilter::create();
25
26     base::Simulation& sim = getDefaultSimulation();
27     sim.enableDefaultLogging( data::output::STDOUT );
28     sim.setFilter( filter );
29
30     filter->addRecordText() << "construction";
```

```
31     DebugStopWatch watch( sim, "Stop Watch" );
32
33     filter->addRecordScope() << typeid( DebugStopWatch );
34     watch.start();
35     watch.stop();
36
37     filter->addProducerLabel() << "Stop Watch";
38     watch.start();
39
40     filter->resetFilter();
41     filter->addProducerType() << typeid( DebugStopWatch );
42     watch.stop();
43 }
```

Quelle 3.21: Verwendung der Klasse SimRecordFilter

In der in Quelle 3.21 dargestellten `main`-Funktion wird zunächst ein `shared_ptr`-verwaltetes `SimRecordFilter`-Objekt erzeugt, das mittels `setFilter` an allen vom Simulationskontext verwalteten Log-Kanälen registriert wird. Wie in Beispiel 8 kommt auch hier das Default-Logging von ODEMX zum Einsatz.

Bei Konstruktion des Filters ist der Modus `passAll` voreingestellt. Demnach werden Log-Einträge anhand der Filtereinträge herausgefiltert und nicht weitergeleitet. Der erste Filtereintrag betrifft die Textnachricht eines Log-Eintrags. Zum Hinzufügen des Strings „construction“ wird die Methode `addRecordText` verwendet. Da der Log-Eintrag, welcher bei der anschließenden Konstruktion eines `DebugStopWatch`-Objekts erzeugt wird, eben diese Textnachricht enthält, kann er den Filter nicht passieren.

Mit der Methode `addRecordScope` können dem Filter Typ-IDs von Klassenkontexten hinzugefügt werden, anhand derer Log-Einträge gefiltert werden sollen. Dies findet in der Regel bei Klassenhierarchien Verwendung. Der Filtereintrag `typeid(DebugStopWatch)` sorgt dafür, dass alle Log-Einträge, die mittels `SimRecord::scope` diesen Klassenkontext angeben, herausgefiltert werden. Aus diesem Grund werden bei den Aufrufen von `start` und `stop` nur die Log-Einträge der Basisklasse `StopWatch` weitergeleitet, nicht jedoch die Debug-Nachrichten aus dem Klassenkontext `DebugStopWatch`.

Das Filtern anhand der Bezeichner von Produzenten geschieht mit Hilfe der Methode `addProducerLabel`. Alle Log-Daten der dadurch spezifizierten Produzenten-Objekte werden durch den Filter geblockt. Dies geschieht unabhängig davon, aus welchem Klassenkontext die Log-Einträge stammen und betrifft damit auch die

Log-Daten von Basisklassenkomponenten wie `StopWatch`. Beim zweiten Aufruf von `start` werden daher gar keine Log-Daten weitergeleitet.

Danach wird der Filter durch `resetFilter` zurückgesetzt, um das Filtern anhand des Produzententyps zu demonstrieren. Dies ist allgemeiner als das Filtern anhand von Bezeichnern oder Klassenkontexten. Dadurch werden alle Log-Daten geblockt, die von Objekten einer angegebenen Klasse erzeugt werden. Folglich werden auch beim letzten Aufruf der Methode `stop` keinerlei Log-Daten erzeugt.

3.4.4 Verarbeitung und textbasierte Darstellung von Log-Daten

Durch die Vorgabe des Log-Datentyps `SimRecord` müssen auch die Log-Konsumenten für diesen Typ spezialisiert werden. Folglich wird die Konsumenten-Schnittstelle durch die Klasse `Log::Consumer<SimRecord>` vorgegeben. Da sie lediglich die Implementation der rein virtuellen Methode `consume` verlangt, unterscheidet sich die neue Schnittstelle stark von der bisher verwendeten `TraceConsumer`-Schnittstelle. Aus diesem Grund mussten die Ausgabekomponenten von ODEmx komplett überarbeitet werden.

Alle im Folgenden vorgestellten Konsumentenklassen haben eine statische `create`-Methode, mit der `shared_ptr`-verwaltete Objekte der jeweiligen Klasse erzeugt werden, da sich zur Sicherung der Lebensdauer nur solche Objekte an einem Log-Kanal als Konsument registrieren lassen. Zur Formatierung der Simulationszeit verfügen alle Ausgabekomponenten weiterhin über eine Methode `setTimeFormat`, deren Verwendung im Beispiel 10 demonstriert wird.

XML-Ausgabe und Anzeige im Browser

Zur Trace-Ausgabe wurden in Version 2.2 von ODEmx die Komponenten `HtmlTrace` und `XmlTrace` angeboten. Da moderne Browser in Verbindung mit Javascript auch XML verarbeiten können, ist diese Trennung nicht mehr notwendig. Die beiden Ausgabeklassen werden daher komplett durch die neue Klasse `XmlWriter` ersetzt, welche in ihrer Funktionalität auf der bisherigen `XmlTrace`-Implementation basiert. Bei Konstruktion eines solchen Objekts muss der Basisdateiname angegeben und eine Obergrenze der Anzahl von Log-Einträgen pro

XML-Datei festgelegt werden. Bei Überschreitung dieser Grenze wird die aktuelle XML-Datei erst dann abgeschlossen und eine neue geöffnet, wenn ein Log-Eintrag mit einer neuen Simulationszeit eintrifft. Dadurch werden in der Simulation gleichzeitig auftretende Ereignisse nicht über mehrere Dateien verteilt. An den Basisdateinamen wird zur Unterscheidung der Dateien und zur Anzeige der Reihenfolge eine laufende Nummer angefügt. Alle erzeugten Dateien werden zur Übersicht mit Name, Start- und Endzeit in einer separaten XML-Datei aufgelistet. Die XML-Schema-Definition für die von der Klasse `XmlWriter` erzeugten XML-Dokumente befindet sich in Anhang B.

Neben den XML-Dateien wird zur Anzeige im Browser eine HTML-Datei erzeugt, welche die XML-Daten mit Hilfe von Javascript parst und darstellt. Wie schon beim `XmlTrace` ist die Anzeige durch Verwendung von Cascading Style Sheets und Javascript dynamisch filterbar. Das betrifft den Nachrichtentext, den Sendernamen, den Sendertyp und den Klassenkontext. Neu ist das Filtern bezüglich der Log-Kanäle. Zwischen den einzelnen XML-Dateien, die das Ablaufprotokoll verschiedener Zeitperioden der Simulation enthalten, kann dynamisch umgeschaltet werden.

Einfache Textausgabe

Auch die einfache Textausgabe wird in Version 3.0 von ODEMX durch Konsumentklassen unterstützt. Dafür wurden mit `ErrorWriter` und `OStreamWriter` zwei neue Ausgabekomponenten implementiert. Die Klasse `ErrorWriter` dient dazu, Warnungen und Fehlermeldungen analog zum bisherigen Format auf die Fehlerausgabe zu schreiben. Das Ausgabeformat beinhaltet folgende Informationen:

- Log-Kanal (warning, error, fatal)
- Nachrichtentext
- Simulationszeit
- Objekt-Bezeichner
- Objekt-Typ
- zusätzliche Details, falls vorhanden
- Klassenkontext, falls vorhanden.

Verwendung findet diese Ausgabekomponente in der Klasse `LoggingManager`, die automatisch ein solches Objekt an allen fehlerbezogenen Log-Kanälen als Konsument registriert, um die Fehlerausgabe zu initialisieren.

Die Klasse `OStreamWriter` kann dazu verwendet werden, Log-Daten an einen beliebigen `std::ostream` weiterzuleiten. Sie dient in erster Linie dazu, auf einfache Weise Ausgaben auf der Standardausgabe erzeugen und überprüfen zu können. Damit soll vermieden werden, dass Anwender von ODEMX in Ermangelung einer solchen Möglichkeit auf die direkte Konsolenausgabe per `std::cout` zurückgreifen müssen, welche im Quellcode nicht gezielt deaktivierbar ist und daher meist auskommentiert wird. Die meisten von ODEMX angebotenen Basiskomponenten zur Modellbildung erzeugen Log-Daten zur Beobachtung interner Zustandsänderungen und zur Meldung von Fehlverhalten. Dazu nutzen sie die Log-Kanäle `trace`, `warning`, `error` und `fatal`. Verschiedene Komponenten nutzen auch den Statistik-Kanal `statistics`. Die Kanäle `debug` und `info` sind jedoch allein für Anwender der Bibliothek reserviert, so dass auf diesen nur nutzerdefinierte Log-Einträge zu beobachten sind. Im Gegensatz zu `std::cout` lässt sich so durch einen Filter oder das Registrieren beziehungsweise Entfernen eines `OStreamWriter`-Konsumenten leicht steuern, wann nutzerdefinierte Log-Daten auf die Standardausgabe oder in eine Datei geschrieben werden sollen.

Beispiel 10: Textbasierte Ausgabe von ODEMX-Log-Daten

Auch dieses Beispiel demonstriert die Verwendung der Logging-Bibliothek auf der Nutzungsebene der Kompletintegration. Es veranschaulicht die Nutzung der textbasierten ODEMX-Ausgabekomponenten `OStreamWriter` und `XmlWriter`. Dabei wird auch auf die Verwendung von Zeitformaten eingegangen. Im Rahmen dieses Beispielprogramms wird die in Beispiel 9 definierte Klasse `DebugStopWatch` wiederverwendet (siehe Quelle 3.20).

```
1 #include "Example_Producer_DebugStopWatch.h"
2 #include <iostream>
3 using namespace odemx::data::output;
4
5 int main() {
6     typedef std::tr1::shared_ptr< OStreamWriter > OStreamWriterPtr;
7     typedef std::tr1::shared_ptr< XmlWriter > XmlWriterPtr;
8
9     TimeBase startTime( 2009, 12, 30, 10*60*60*1000, TimeUnit::milliseconds );
10
```

```
11   OStreamWriterPtr os = OStreamWriter::create( std::cout );
12   os->setTimeFormat( new GermanTime( startTime ) );
13   XmlWriterPtr xml = XmlWriter::create( "Example_Consumer", 1000 );
14   xml->setTimeFormat( new Iso8601Time( startTime, 1, 0 ) );
15
16   odemx::base::Simulation& sim = odemx::getDefaultSimulation();
17   sim.addConsumer( odemx::data::channel_id::debug, os );
18   sim.addConsumer( odemx::data::channel_id::info, os );
19   sim.addConsumer( odemx::data::channel_id::info, xml );
20
21   DebugStopWatch watch( sim, "Stop Watch" );
22   watch.start();
23   sim.setCurrentTime( 42 );
24   watch.stop();
25 }
```

Quelle 3.22: Verwendung der textbasierten Ausgabekomponenten `OStreamWriter` und `XmlWriter` inklusive Zeitformatierung für die Simulationszeit

Das Programm in Quelle 3.22 illustriert die Verwendung der textbasierten Ausgabekomponenten mit den beiden von ODEMX unterstützten Zeitformaten `GermanTime` und `Iso8601Time`. Die Simulationszeit startet in ODEMX normalerweise mit dem Wert 0. Der Fortschritt der Simulationszeit erfolgt in abstrakten Zeiteinheiten, deren Interpretation dem Anwender obliegt. Zur Umwandlung der Simulationszeit in ein bestimmtes Ausgabeformat müssen folglich die Zeiteinheit und der Startzeitpunkt vorgegeben werden. Letzterer entspricht dabei dem Simulationszeitpunkt 0.

Gleich zu Beginn der `main`-Funktion werden sowohl die Startzeit als auch die Zeiteinheit festgelegt. Dafür wird ein Objekt der Klasse `TimeBase` erzeugt, welches den Zeitpunkt 30. Dezember 2009 um 10:00 Uhr repräsentiert. Als Einheit werden Millisekunden angegeben. Solch ein Objekt wird immer bei der Konstruktion eines Zeitformats benötigt. Daraufhin werden zwei Konsumenten durch Aufruf ihrer `create`-Funktion erzeugt, wobei dem `OStreamWriter` das deutsche Zeitformat⁸ zugeordnet wird. Der Konstruktor der Klasse `GermanTime` verlangt dabei lediglich die Übergabe eines `TimeBase`-Objekts. Der `XmlWriter` soll das Format ISO 8601⁹ zur Zeitausgabe verwenden. Dieses Format wird als zusammenhängende Datums- und Zeitangabe mit der Differenz zur Koordinierten Weltzeit (UTC) dargestellt. Daher muss bei Konstruktion eines `Iso8601Time`-Objekts neben `TimeBase` auch diese Differenz in Stunden und Minuten angegeben werden.

⁸ Ausgabeformat: YYYY-MM-DD / hh:mm:ss.f

⁹ Ausgabeformat: YYYY-MM-DDThh:mm:ss.f±HH:mm

Nach der Festlegung der Zeitformate folgt die Registrierung der Konsumenten an den Log-Kanälen. Der `OStreamWriter` dient hier zur Anzeige aller via `debug` und `info` versendeten Log-Daten. Der `XmlWriter` soll dagegen jedoch nur die Log-Daten des Kanals `info` in einer Datei festhalten. Auf diese Weise können die Debug-Ausgaben auf der Standardausgabe beobachtet werden, während die XML-Datei nur die simulationsrelevanten Daten aufzeichnet.

Im letzten Abschnitt der `main`-Funktion wird die in Quelle 3.20 definierte Klasse `DebugStopWatch` verwendet. Diese versendet Log-Einträge über den Kanal `debug`, während ihre Basisklassenkomponente `StopWatch` den Kanal `info` verwendet. Auf der Standardausgabe erscheinen nun sowohl die Informationen zu Konstruktion und Destruktion des Objekts als auch zu den Aufrufen von `start` und `stop`. In der erzeugten XML-Datei `Example_Consumer_0.xml` finden sich jedoch nur die `info`-Daten wieder. Daneben werden noch die Dateien `Example_Consumer_file_info.xml` und `Example_Consumer.html` angelegt. Erstere enthält eine Liste der erzeugten Log-Dateien, während letztere den Rahmen für die Anzeige der XML-Daten im Web-Browser bereitstellt.

3.4.5 Anbindung an relationale Datenbanken

Zur Anbindung von ODEMX an relationale Datenbanken ist es notwendig, ein relationales Datenbankschema zu definieren. Der erste Schritt der Datenmodellierung ist dabei die Erstellung eines ER-Modells. Aus diesem lässt sich leicht ein relationales Datenbankschema ableiten, welches in der Datenbank schließlich durch die entsprechende Tabellenstruktur realisiert wird.

Entity-Relationship-Modell

Da der Prozess der Typisierung von Entitäten nicht durch einen formalen Algorithmus vorgegeben ist, wird nach der von Schubert aufgestellten Regel zur Klassenbildung vorgegangen (siehe Abschnitt 2.2.2 auf Seite 15). Entitäten mit gleicher Attributskombination, die unter einem Oberbegriff zusammengefasst werden können, bilden demnach eine Entitätenklasse. Eine erste Klassifikation der von ODEMX generierten Daten in die Entitätenklassen `Simulationslauf`, und `Log-Eintrag` ist dabei offensichtlich, da `Simulationsläufe` und `Log-Einträge` jeweils durch eine bestimmte Kombination von Attributen beschrieben und daher als

Typen zusammengefasst werden können. Bei genauer Analyse des ODEMX-Log-Datentyps `SimRecord` findet sich mit Log-Eintragsdetail allerdings noch eine weitere Entitätenklasse, da Log-Einträge beliebig viele dieser Entitäten transportieren können.

Abbildung 3.7 zeigt das entsprechende ER-Diagramm in Chen-Notation inklusive aller den Entitätenklassen zugeordneten Attribute. Ein Simulationslauf wird demnach beschrieben durch den Zeitpunkt der Ausführung, einen nicht notwendigerweise eindeutigen Bezeichner, eine Beschreibung des Experiments und einen alphanumerischen Identifikator im standardisierten Format Universally Unique Identifier (UUID). Zur besseren Unterscheidung weist ODEMX jedem Simulation-Objekt bei Konstruktion automatisch eine derartige alphanumerische Kennzeichnung zu. Zwischen den Entitätenklassen Simulationslauf und Log-Eintrag besteht die 1:n-Beziehung erzeugt, da während eines Simulationslaufs beliebig viele Log-Einträge produziert werden können. Attribute von Log-Einträgen sind die Textnachricht, der Log-Kanal, der optionale Klassenkontext, Sendername und -typ, sowie die Simulationszeit der Erzeugung. Zur Entitätenklasse Log-Eintragsdetail wiederum besteht die 1:n-Beziehung transportiert, da Log-Einträge beliebig viele Details enthalten können. Die Attribute sind jeweils der Detailname und der Wert.

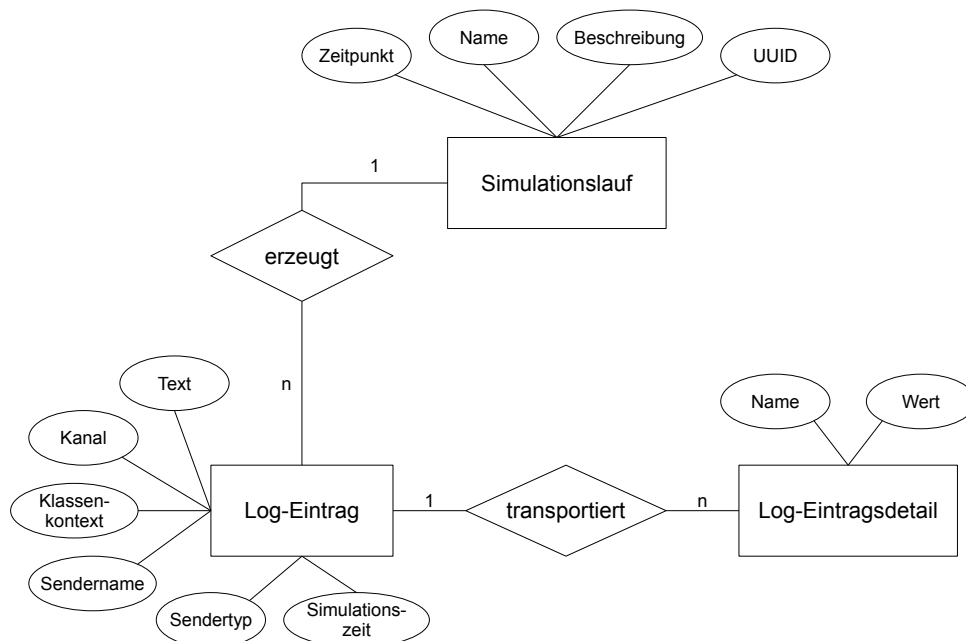


Abbildung 3.7: ER-Diagramm für die von ODEMX generierten Log-Daten

Relationales Datenbankschema und Tabellenstruktur

Zur Strukturierung von Datenbanken, die ODEMX-Simulationsdaten speichern, wird das im vorigen Abschnitt erstellte ER-Modell in ein relationales Datenbankschema überführt. Dieses ist in Abbildung 3.8 dargestellt.

Die Entitätenklassen Simulationslauf, Log-Eintrag und Log-Eintragsdetail sind dabei direkt durch die Relationsschemata `odemx_simulation_run`, `odemx_sim_record` beziehungsweise `odemx_sim_record_detail` repräsentiert. Da keine der drei beschriebenen Entitätenklassen Schlüsselattribute besitzt¹⁰, wird zu jedem Relationsschema ein weiteres Attribut `id` hinzugefügt, welches die Unterscheidbarkeit von Tupeln sichert und daher als Primärschlüssel verwendet wird.

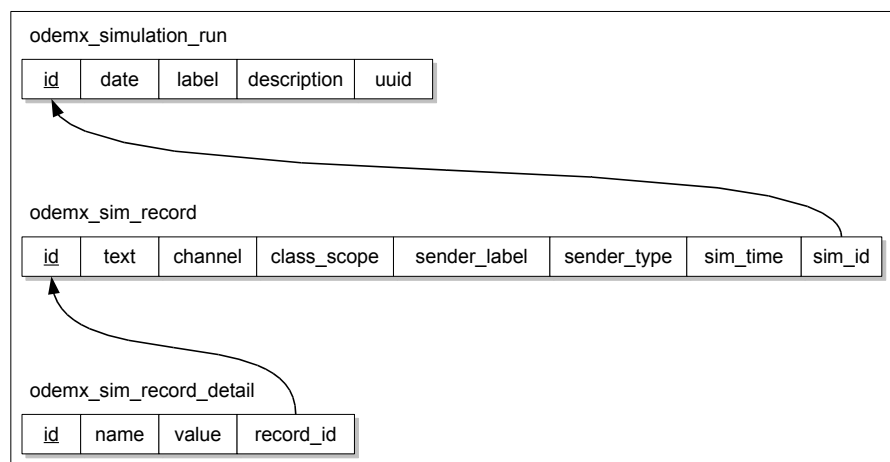


Abbildung 3.8: Relationales Datenbankschema für die von ODEMX generierten Log-Daten

Das ER-Modell beschreibt weiterhin die Beziehungen erzeugt und transportiert. Eine solche 1:n-Beziehung zwischen Entitätenklassen kann im Datenbankschema entweder durch ein weiteres Relationsschema mit zwei Fremdschlüsseln repräsentiert werden, oder es kann ein Fremdschlüssel-Attribut zu der Entitätenklasse auf

¹⁰ Diese Attribute müssen für unterschiedliche Tupel einer Relation garantiert eindeutige Werte haben. Die für Simulationen generierten UUIDs unterscheiden sich zwar mit sehr hoher Wahrscheinlichkeit, aber ihre Einmaligkeit ist nicht garantiert.

der n-Seite der Beziehung hinzugefügt werden. In diesem Fall wurde die zweite Lösung gewählt. So referenzieren die Fremdschlüssel-Attribute `sim_id` und `record_id` der Relationsschemata `odemx_sim_record` und `odemx_sim_record_detail` die Primärschlüssel der Relationsschemata `odemx_simulation_run` beziehungsweise `odemx_sim_record`.

Das aus der Datenmodellierung gewonnene relationale Datenbankschema gibt nun die Tabellenstruktur der Datenbank vor. Für jedes Relationsschema wird eine Tabelle angelegt, deren Spaltennamen genau den Attributen des Relationsschemas entsprechen. Dadurch ergibt sich eine Verteilung der Log-Daten von ODEMX auf die drei Tabellen `odemx_simulation_run`, `odemx_sim_record` und `odemx_sim_record_detail`, die im Folgenden näher betrachtet werden. Jede der drei Tabellen verfügt über eine Spalte `id`, die als Primärschlüssel der jeweiligen Tabelle verwendet wird und den anderen Tabellen zur Referenzierung einzelner Datensätze dient.

Feldname	Feldtyp	Primär-schlüssel	NULL-Werte	Standardwert	Fremd-schlüssel
<code>id</code>	Integer	Ja	Nein		
<code>date</code>	Timestamp	Nein	Nein		
<code>label</code>	Varchar	Nein	Nein		
<code>description</code>	Varchar	Nein	Nein		
<code>uuid</code>	Char(40)	Nein	Nein		

Tabelle 3.3: Struktur der Datenbanktabelle `odemx_simulation_run`

Tabelle 3.3 beschreibt die Struktur der Datenbanktabelle `odemx_simulation_run`. Das Feld `date` speichert den Zeitpunkt, zu dem der Simulationslauf durchgeführt wurde, die Felder `label` und `description` sind für den Namen des Simulationsobjekts sowie die Experimentbeschreibung vorgesehen, und im Feld `uuid` wird die von ODEMX erzeugte Simulations-ID eingetragen. Der SQL-Standard stellt bislang keinen Datentyp für UUIDs bereit, weshalb an dieser Stelle eine Zeichenkette fester Größe verwendet wird. Eine UUID kann im Stringformat nützlich sein, um Dateinamen zu erzeugen, die eine leichte Zuordnung zu dem erzeugenden Simulationslauf ermöglichen. Effizienter wäre das Speichern von UUIDs als 16 Bytes im Binärformat, wodurch der Wert in der Tabelle jedoch zur Darstellung immer eine Konvertierung verlangen würde und nicht mehr menschenlesbar wäre.

In Tabelle 3.4 ist die Struktur der Datenbanktabelle `odemx_sim_record` dargestellt, die vorwiegend Felder mit Strings enthält. Im Feld `text` wird die Nachricht

Feldname	Feldtyp	Primär-schlüssel	NULL-Werte	Standard-wert	Fremdschlüssel
id	Integer	Ja	Nein		
text	Varchar	Nein	Nein		
channel	Varchar	Nein	Nein		
class_scope	Varchar	Nein	Ja		
sender_label	Varchar	Nein	Nein		
sender_type	Varchar	Nein	Nein		
sim_time	Varchar	Nein	Nein		
sim_id	Integer	Nein	Nein		odemx_simulation_run.id

Tabelle 3.4: Struktur der Datenbanktabelle odemx_sim_record

eines Log-Eintrags festgehalten, channel beschreibt den Log-Kanal beziehungsweise die Kategorie und der optionale class_scope den Klassenkontext, in dem der Eintrag erzeugt wurde. Die Felder sender_label und sender_type geben Aufschluss über Namen und Klassentyp des Produzenten, und das Feld sim_time enthält die Simulationszeit als String, weil das gewählte ODEMx-Zeitformat nicht unbedingt dem SQL-Datentyp Timestamp entsprechen muss. Durch sim_id wird die Referenz zu einem Simulationslauf hergestellt, dessen Datensatz in der Tabelle odemx_simulation_run eingetragen ist.

Feldname	Feldtyp	Primär-schlüssel	NULL-Werte	Standard-wert	Fremdschlüssel
id	Integer	Ja	Nein		
name	Varchar	Nein	Nein		
value	Varchar	Nein	Nein		
record_id	Integer	Nein	Nein		odemx_sim_record.id

Tabelle 3.5: Struktur der Datenbanktabelle odemx_sim_record_detail

Die Struktur der Datenbanktabelle odemx_sim_record_detail wird in Tabelle 3.5 gezeigt. Da Details von Log-Einträgen immer Paare aus Name und Wert sind, erfolgt die Speicherung in den entsprechenden Feldern name und value. Das Feld record_id referenziert dabei den Datensatz des Log-Eintrags in Tabelle odemx_sim_record, dem die Detailinformationen zuzuordnen sind.

Normalformbestimmung

Die Normalisierung ist eine Analyse von Relationsschemata auf der Grundlage von funktionalen Abhängigkeiten und Schlüsselkandidaten. Sie dient der Sicherung wünschenswerter Eigenschaften von Relationsschemata. Einerseits betrifft dies eine minimale Redundanz und andererseits die Minimierung von Einfüge-, Löschen- und Modifikationsanomalien.

Um die erste Normalform zu erfüllen, müssen die Wertebereiche aller Attribute eines Relationsschemas nur atomare Werte enthalten. Alle Relationsschemata des in Abbildung 3.8 dargestellten relationalen Datenbankschemas für ODEMx-Simulationsdaten erfüllen diese erste Normalform, weil ihre Attributwerte nicht weiter zerlegbar sind.

Die Erfüllung der zweiten Normalform erfordert, dass ein Relationsschema sich in der ersten Normalform befindet und kein Nichtschlüsselattribut von einer echten Teilmenge eines Schlüsselkandidaten funktional abhängig ist. Diese Normalform ist automatisch erfüllt, wenn die Schlüsselkandidaten jeweils aus maximal einem Attribut bestehen. Dies ist bei allen beschriebenen Relationsschemata der Fall, da jeweils der Primärschlüssel `id` der einzige Schlüsselkandidat ist. Somit ist auch die zweite Normalform erfüllt.

Damit ein Relationsschema in der dritten Normalform ist, muss die zweite Normalform erfüllt sein, und kein Nichtschlüsselattribut darf transitiv von einem Schlüsselkandidaten abhängen. Diese Forderung wird allerdings von dem Relationsschema `odemx_sim_record` verletzt, da das Attribut `sender_type` von der Attributmenge `{sender_label, sim_id}` funktional abhängig und damit vom Primärschlüssel `id` transitiv abhängig ist. Folglich erfüllt das vorgestellte relationale Datenbankschema nicht die dritte Normalform. Da theoretisch verschiedene Arten von Anomalien in entsprechenden Datenbanken auftreten könnten, wird an dieser Stelle die Relevanz von Anomalien im Hinblick auf die Archivierung von ODEMx-Simulationsdaten bewertet.

In der Tabelle `odemx_sim_record` könnten alle drei in Abschnitt 2.2.3 beschriebenen Arten von Anomalien auftreten. Eine Einfügeanomalie würde auftreten, wenn bei einem neuen Eintrag in der Tabelle `odemx_sim_record` der Wert des Attributs `sender_type` von den Attributwerten existierender Datensätze eines bestimmten Senders abweicht. Auch ist es nicht möglich, einfach nur einen Sendertyp ohne weitere Attributwerte in der Datenbank zu speichern, weil dies höchstens

mit Nullwerten zu erreichen wäre, diese jedoch in fast allen Attributen unzulässig sind. Eine Löschanomalie kann auftreten, indem unbeabsichtigt die Information über einen bestimmten Sendertyp gelöscht wird, weil beispielsweise alle Sender eines Typs anhand ihrer Labels entfernt werden. Eine Modifikationsanomalie könnte während der Änderung des Typs eines Senders entstehen. Dabei muss das Attribut `sender_type` in allen Datensätzen dieses Senders aktualisiert werden. Sollten Datensätze bei der Aktualisierung übersehen werden, so würde die Datenbank inkonsistente Daten enthalten, da für diesen Sender auf einmal zwei verschiedene Typen gespeichert wären.

Relevant sind diese zugegebenermaßen konstruierten Beispiele für Anomalien im Fall von ODEMX deshalb nicht, weil Simulationsdaten lediglich durch ein Simulationsprogramm generiert werden, das selbst keine inkonsistenten Datensätze erzeugt, und im Nachhinein daran keine Änderungen vorgenommen werden. Die Auswertung der Daten benötigt lediglich lesenden Zugriff auf die Datenbank, wogegen die aufgeführten Anomalien ausschließlich bei schreibendem Zugriff auftreten können.

Es ist möglich, das vorliegende relationale Datenbankschema in die dritte Normalform zu bringen, indem weitere Tabellen für die Sender und ihre möglichen Typen angelegt werden, wobei die Zuordnung durch Fremdschlüssel umgesetzt wird. Zur Minimierung des Speicherplatzes wäre dies sicher die bevorzugte Lösung. In Bezug auf die Vermeidung von Redundanz kommt an dieser Stelle aber der Kostenfaktor ins Spiel, da für jeden Log-Eintrag eine Prüfung vorgenommen werden müsste, ob der Sender und dessen Typ für den aktuellen Simulationskontext bereits eingetragen ist, oder ob er hinzugefügt werden muss. Dazu benötigt man zusätzliche Datenbankabfragen während des Simulationslaufes, die eine nicht zu vernachlässigende Zeitperiode benötigen würden. Da komplexe Simulationsexperimente ohnehin eine lange Laufzeit haben, wird an dieser Stelle der Leistung des Programms der Vorrang gegeben.

Die Klasse DatabaseWriter

Die Logging-Bibliothek stellt auf der Basis von POCO Data eine Zugriffsschicht bereit, um mit dem eingebetteten RDBMS SQLite oder via ODBC mit externen RDBMS zu interagieren. Darauf kann nun in ODEMX aufgebaut werden. Die Anbindung an relationale Datenbanken erfolgt durch die Ausgabekomponente

DatabaseWriter, welche eine Instanz der Klasse `Log::DatabaseAccessor` verwendet, um die Verbindung herzustellen und mit Hilfe von `SqlInsertter`-Objekten Daten in die drei aufgeführten Tabellen zu schreiben. Sollten die Tabellen noch nicht in der Datenbank angelegt sein, so geschieht dies automatisch in der Initialisierungsphase. Ob die Klasse `DatabaseWriter` ODBC oder `SQLite` unterstützen soll, muss allerdings schon während des Kompiliervorgangs festgelegt werden. Ein Wechsel verlangt demnach das erneute Übersetzen der Klasse.

Bereits in Abschnitt 3.3.8 wurde erwähnt, dass das Puffern von Daten bei hohem Datenaufkommen sinnvoll ist, und so verlangt der Konstruktor die Angabe einer Pufferobergrenze, bis zu der Daten im Hauptspeicher vorgehalten werden. Umgesetzt wird diese Funktionalität durch die Klasse `SimRecordBuffer`, die alle Log-Daten in einem `std::vector` zwischenspeichert. Dabei muss allerdings beachtet werden, dass manche in Log-Einträgen enthaltenen Informationen nicht dauerhaft sind. So ist beispielsweise bei Zeigern auf Produzenten nicht gesichert, dass die referenzierten Objekte bei der Leerung des Puffers noch existieren. Die relevanten Daten – in diesem Fall Name und Typ des Produzenten – müssen daher anstelle der Zeiger kopiert und gespeichert werden. Im Puffer werden die Log-Daten durch die von `Log::Record` abgeleitete interne Klasse `StoredRecord` verwaltet, die vergängliche Daten in Infotypen kopiert.

Das Entleeren des Puffers der Klasse `DatabaseWriter` geschieht automatisch durch die Methode `flush`, die in der Methode `consume` immer dann gerufen wird, wenn der Puffer seine Obergrenze erreicht hat. Im Destruktor der Klasse `DatabaseWriter` wird `flush` ein letztes Mal gerufen, um sicherzustellen, dass bei der Zerstörung des Objekts keine ungeschriebenen Daten im Puffer verbleiben. Auch ein manueller Aufruf von `flush` ist zu beliebigen Zeitpunkten möglich. Die Methode `insertRecord` bestimmt dabei, wie die Daten eines Log-Eintrags in die Datenbank eingefügt werden. Auch in der Klasse `DatabaseWriter` wird das Methodentemplate `DatabaseAccessor::storeData` verwendet, um die gepufferten Simulationsdaten einzupflegen. Dafür werden analog zu Quelle 3.16 der Datenpuffer und die Methode `insertRecord` als Parameter übergeben.

Beispiel 11: Archivieren von Log-Daten in einer Datenbank

An dieser Stelle wird die Logging-Bibliothek auch wieder auf der Nutzungsebene der Komplettintegration verwendet. Nach den in Beispiel 10 vorgestellten

ten textbasierten Ausgabekomponenten wird an dieser Stelle die Datenbank-
bindung von ODEMX-Simulationsprogrammen mit Hilfe der Konsumentenklasse
DatabaseWriter illustriert. Die in Quelle 3.18 definierte Klasse Stopwatch wird
dabei erneut als Produzentenklasse eingebunden.

```
1 #include "Example_Producer_StopWatch.h"
2
3 int main() {
4     using namespace odemx;
5     using namespace odemx::data::output;
6
7     typedef std::tr1::shared_ptr< DatabaseWriter > DatabaseWriterPtr;
8     DatabaseWriterPtr dbWriter = DatabaseWriter::create(
9         "Driver=PostgreSQL_Driver;"
10        "UID=username;"
11        "PWD=password;"
12        "Database=odemx_example_database;"
13        "Server=localhost;"
14        "Port=5433;", 10000 );
15
16    /* DatabaseWriterPtr dbWriter = DatabaseWriter::create(
17        "example_databasewriter.db", 10000 ); */
18
19    dbWriter->setTimeFormat( new GermanTime(
20        TimeBase( 2009, 4, 4, 16*60*60, TimeUnit::seconds ) ) );
21
22    base::Simulation& sim = getDefaultSimulation();
23    sim.addConsumer( dbWriter );
24
25    Stopwatch watch( sim, "Stop Watch" );
26    watch.start();
27    sim.setCurrentTime( 10 );
28    watch.stop();
29
30    try { dbWriter->flush(); }
31    catch( Poco::Exception& ex ) { std::cout << ex.displayText(); }
32 }
```

Quelle 3.23: Verwendung der Klasse DatabaseWriter zur Datenbank-
bindung von ODEMX-Simulationsprogrammen

Quelle 3.23 zeigt die main-Funktion des Beispielsprogramms. Darin wird zuerst
einmal ein DatabaseWriter-Objekt durch Aufruf der statischen Methode create
erstellt. Als Argumente werden dabei ein Verbindungsstring und die Pufferober-
grenze angegeben. Da in diesem Beispiel via ODBC die Verbindung zu einer
PostgreSQL-Datenbank hergestellt werden soll, muss der Verbindungsstring ei-
ne Reihe von Angaben enthalten: den ODBC-Treiber, Nutzernamen und Passwort,
den Namen der Datenbank sowie Adresse und Port des Datenbank-Servers. Die

Pufferobergrenze wird hier auf 10.000 Log-Einträge gesetzt. Die auskommentierte Anweisung zeigt, wie ein `DatabaseWriter` zur Verwendung mit SQLite zu erstellen wäre. Wie schon bei den textbasierten Ausgabekomponenten kann auch bei der Datenbankausgabe eine Formatierung der Simulationszeit vorgenommen werden – in diesem Fall durch ein Objekt der Klasse `GermanTime`.

Anschließend wird der Default-Simulationskontext angefordert und der `DatabaseWriter` an diesem als Konsument registriert. Dem folgt die Konstruktion eines `StopWatch`-Objekts, das hier analog zu Beispiel 8 als Produzent verwendet wird. Die in den Funktionen `start` und `stop` erzeugten Log-Einträge werden an den `DatabaseWriter` weitergeleitet und von diesem gepuffert.

Der Aufruf von `flush` am Ende der Funktion sorgt dann für das Archivieren der Log-Daten in der Datenbank. Dabei werden, falls nötig, die drei oben beschriebenen Tabellen angelegt und die erzeugten Log-Daten eingetragen. Ein manueller Aufruf von `flush` ist nicht zwingend notwendig, da die Methode auch im Destruktor der Klasse gerufen wird, falls der Puffer noch Daten enthält. Allerdings werden dann die von POCO Data geworfenen Ausnahmen vom Destruktor nicht nach außen weitergereicht¹¹.

Logging von parallel ausgeführten Simulationsprogrammen

ODEMX-basierte Systemmodelle bilden parallele Prozesse auf Koroutinen ab, so dass nebenläufige Aktionen in ihrer Ausführung sequentialisiert werden. Aus diesem Grund werden ODEMX-Simulationsprogramme typischerweise in einem einzigen Thread ausgeführt. Moderne Mehrkern-Prozessoren können allerdings mehrere Threads parallel ausführen. Um diese Leistungsfähigkeit auszunutzen liegt es nahe, mehrere Simulationsexperimente gleichzeitig zu starten, die dann parallel auf den verfügbaren Prozessor-Kernen abgearbeitet werden.

Die in Abschnitt 2.2.4 erläuterten ACID-Eigenschaften von RDBMS bilden die Grundlage für die fehlerfreie gleichzeitige Interaktion mehrerer Anwendungen mit derselben Datenbank. Die durch das RDBMS bereitgestellte Infrastruktur übernimmt die Synchronisation der Anfragen, wenn parallel laufende Anwendungen zur gleichen Zeit mit derselben Datenbank arbeiten wollen.

¹¹ Destruktoren sollten in C++ grundsätzlich keine Ausnahmen werfen, um nicht den vorzeitigen Programmabbruch zu verursachen. Dies geschieht nämlich automatisch, wenn eine zweite Ausnahme geworfen wird, bevor die erste gefangen wurde.

Ein Problem, das in diesem Zusammenhang auftaucht, ist jedoch die Vergabe der Primärschlüssel für die einzupflegenden Datensätze. Die Simulationsprogramme können bei paralleler Ausführung nicht garantieren, dass nie der gleiche Wert mehrfach verwendet wird. Bei Eintreten dieses Falls wird aber eine Integritätsbedingung der Datenbank verletzt, weshalb die Transaktion mit einer Fehlermeldung abgebrochen wird, um die Konsistenz des Datenbestands zu wahren.

Die Lösung des Problems besteht darin, dass die Datenbank den entsprechenden Primärschlüssel für jeden neuen Datensatz selbst erzeugt. Portabel ist dieser Ansatz allerdings nicht zu implementieren, da der SQL-Standard keinen Datentyp vorgibt, der automatisch für jeden Eintrag einzigartige Werte erzeugt. Die Umsetzung muss daher RDBMS-spezifisch erfolgen. PostgreSQL bietet für diesen Zweck den SQL-Datentyp `SERIAL`, Microsoft SQL Server hingegen erlaubt mit `IDENTITY` die Angabe einer Spalteneigenschaft, die einzigartige Werte sichert. Ähnlich geht dies bei MySQL mit dem Schlüsselwort `AUTO_INCREMENT`, wogegen das Problem bei Oracle mit Sequenzen und Triggern gelöst werden muss. Die mit dieser Arbeit vorliegende Version 3.0 von ODEMx enthält zunächst nur die Unterstützung für PostgreSQL, da das EMS dieses RDBMS verwendet. Aber auch für andere RDBMS sollte diese Funktionalität analog zu implementieren sein.

Im Gegensatz zu den oben aufgeführten RDBMS darf bei SQLite maximal ein Nutzer schreibend auf die Datenbank zugreifen. Parallel ablaufende Simulationsprogramme können somit nicht gleichzeitig dieselbe Datenbank-Datei verwenden, weil das erste erfolgreich verbundene Programm den Zugriff für andere Anwendungen blockiert und deren Zugriffe in Fehlermeldungen resultieren. Es ist zwar möglich, für eine bestimmte Zeit zu warten und die Daten erst dann zu schreiben, wenn der Zugriff wieder freigegeben ist. Allerdings würde dies zu einer Sequenzialisierung der parallel gestarteten Simulationsprogramme führen und den erhofften Leistungsgewinn negieren. Daher ist die gemeinsame Nutzung von SQLite-Datenbanken durch mehrere Simulationsprogramme nicht vorgesehen.

3.4.6 Statistikerfassung und Berichterzeugung

Mit der Einführung eines allgemeinen Logging-Konzeptes in ODEMx geht auch eine Veränderung im Bereich Statistik einher. Im bisherigen Report-Konzept sammeln alle `ReportProducer` ihre statistischen Daten selbst und speichern diese in aggregierter Form, so dass statistische Kenngrößen auf Anfrage in Tabellenform

ausgelesen und durch Report-Ableitungen in einem Bericht dargestellt werden können.

Bei der Archivierung von Simulationsdaten in einer Datenbank ist es sinnvoller, alle statistisch relevanten Ereignisse in nicht-aggregierter Form zu versenden und einzeln zu speichern. Die Berechnung statistischer Kenngrößen muss in diesem Fall in der Auswertungsphase vorgenommen werden. Das Vorhalten aller Statistik-Einträge hat dabei den Vorteil, dass im Nachhinein beliebige Zeitintervalle und Zusammenhänge zwischen den Daten untersucht werden können.

Statistische Log-Daten werden in Version 3.0 von ODEMX durch eine eigene Logging-Kategorie repräsentiert. Die Basisklasse `odemx::data::Producer` gibt allen ODEMX-Datenproduzenten Zugriff auf den zu diesem Zweck geschaffenen Log-Kanal `statistics`, der wie die anderen Kanäle bezüglich des Datentyps `SimRecord` parametrisiert ist. Eine Analyse der statistikerzeugenden Klassen in Version 2.2 von ODEMX führte zu der Feststellung, dass sich ODEMX-relevante Statistikdaten durch vier Arten von Log-Einträgen beschreiben lassen: Parameterwerte, Anzahl bestimmter Ereignisse, Wertänderungen und Rücksetzzeit. Dementsprechend stellt `odemx::data::Producer` vier Methoden zur einfachen Erzeugung statistischer Log-Daten bereit (siehe Abschnitt 3.4.2). Alle Statistikproduzenten sind nun von dieser Klasse abgeleitet statt wie bisher von `ReportProducer`. Das Report-Konzept findet aber weiterhin seine Verwendung im Zusammenhang mit den statistikberechnenden Komponenten von ODEMX, die im Modul `Statistics` angeboten werden.

Damit auch ohne Datenbankanbindung statistische Kenngrößen berechnet werden können, wird die spezielle Konsumentenklasse `StatisticsBuffer` eingeführt. Durch Registrierung eines solchen Objekts am Log-Kanal `statistics` werden für alle Produzenten die gleichen statistischen Kenngrößen berechnet, wie sie zuvor von den `ReportProducer`-Klassen intern vorgehalten wurden. Dies geschieht auf der Grundlage der Klasse `odemx::stats::Accumulate`. In der Klasse `StatisticsBuffer` assoziiert eine `std::map` die Bezeichner der Produzenten mit ihren gesammelten statistischen Daten. Lesenden Zugriff auf diesen Container gewährt die Methode `getStatistics`. Gesendete Parameter- und Zählerwerte werden als Paare aus Name und Wert vorgehalten. Neben den aggregierten Daten puffert die Klasse auf Wunsch auch alle einzelnen Wertänderungen mit ihrer entsprechenden Simulationszeit – ein Konstruktorparameter aktiviert diese Funktionalität. Die vorgehaltenen Daten lassen sich dann zu einem späteren Zeitpunkt

für weitere Berechnungen verwenden. So kann beispielsweise ein Histogramm erstellt oder eine lineare Korrelationsanalyse durchgeführt werden. Mit der Methode `isBufferingUpdates` lässt sich feststellen, ob das Zwischenspeichern aktiviert ist. Der Zugriff auf gepufferte Wertänderungen eines Produzenten erfolgt durch die Methode `getUpdates`. Ein Aufruf von `reset` löscht alle berechneten und gespeicherten Daten – nur die meist während der Konstruktion übermittelten Parameter der Statistikproduzenten bleiben davon unberührt.

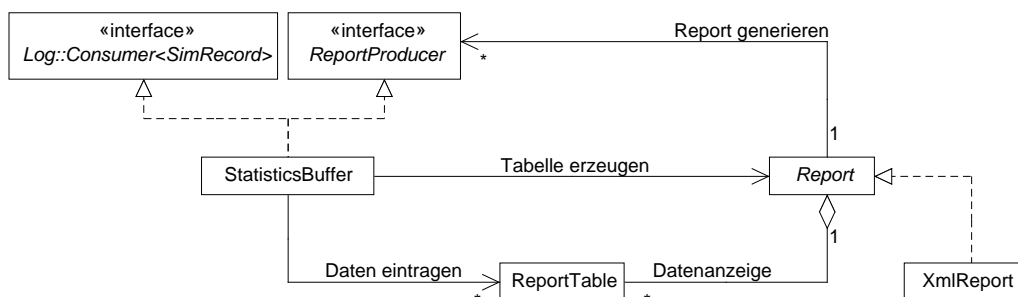


Abbildung 3.9: Einbettung der Klasse `StatisticsBuffer` und Zusammenspiel mit den Klassen des Report-Konzepts

Durch die Implementation der `ReportProducer`-Schnittstelle ist die Konsumentenklasse `StatisticsBuffer` mit dem Report-Konzept kompatibel. Abbildung 3.9 zeigt die Einbettung in die Klassenhierarchie von ODEMX. Report-Spezialisierungen stellen statistische Daten in tabellarischer Form dar. Die von der Klasse `XmlReport` erzeugten XML-Dateien werden dabei mittels XSL-Transformation direkt in modernen Web-Browsern angezeigt. Die XML-Schema-Definition für die von der Klasse `XmlReport` erzeugten XML-Dokumente befindet sich in Anhang B. Mit `OStreamReport` wird in Version 3.0 von ODEMX eine weitere Ausgabeklasse angeboten, die Daten im einfachen Textformat direkt an einen `std::ostream` weiterleitet, womit Statistik-Berichte beispielsweise auf der Standardausgabe dargestellt werden können. Die bisher angebotene Klasse `HtmlReport` wird nicht mehr benötigt, da die Klasse `XmlReport` ihre Funktionalität abdeckt.

Beispiel 12: Statistik-Berechnung ohne Datenbankanbindung

In diesem Beispiel wird die Logging-Bibliothek ebenfalls auf der Nutzungsebene der Kompletintegration verwendet. Das Beispiel illustriert in den Quellen 3.24 und 3.25 die Erzeugung statistischer Daten mit Hilfe des Log-Kanals `statistics` sowie die Verwendung der Konsumentenklasse `StatisticsBuffer`. Dabei wird auch auf das Zwischenspeichern von Wertänderungen eingegangen, aus denen ein Histogramm generiert wird. Erneut findet hier die in Beispiel 8 definierte Klasse `StopWatch` (siehe Quelle 3.18) Verwendung als Basisklasse.

```
1 #include "Example_Producer_StopWatch.h"
2 using namespace odemx;
3
4 class StatisticsStopWatch: public StopWatch {
5 public:
6     StatisticsStopWatch( base::Simulation& sim, const data::Label& label )
7         : StopWatch( sim, label )
8     {}
9     void start() {
10         StopWatch::start();
11         statistics << count( "uses" );
12     }
13     void stop() {
14         StopWatch::stop();
15         statistics << update( "duration", StopWatch::getDuration() );
16     }
17 };
```

Quelle 3.24: Definition der Klasse `StatisticsStopWatch` zur Demonstration der Verwendung des Log-Kanals `statistics`

Quelle 3.24 zeigt die Klasse `StatisticsStopWatch`, welche die Klasse `StopWatch` um das Logging statistisch relevanter Werte erweitert. In der Methode `start` wird mittels `count` ein Log-Eintrag zum Zählen der Aufrufe erzeugt und via `statistics` versendet. Beim Stoppen der Uhr mittels `stop` wird dann jeweils die gemessene Zeitdauer protokolliert. Die Log-Einträge für diese Werte werden durch die Methode `update` erzeugt, die als Argumente einen Bezeichner und einen Wert erwartet.

```
18 int main() {
19     using data::buffer::StatisticsBuffer;
20     typedef std::tr1::shared_ptr< StatisticsBuffer > BufferPtr;
21
22     base::Simulation& sim = getDefaultSimulation();
23     BufferPtr buf = StatisticsBuffer::create( sim, "Statistics Buffer", true );
24     sim.addConsumer( data::channel_id::statistics, buf );
25
26     odemx::random::Normal normal( sim, "Normal Distribution", 42, 5 );
27     StatisticsStopWatch watch( sim, "Statistics Stop Watch" );
28     int i = 10000;
29     while( i-- ) {
30         watch.start();
31         sim.setCurrentTime( sim.getTime() + normal.sample() );
32         watch.stop();
33     }
34
35     const StatisticsBuffer::UpdateDeque& updates =
36         buf->getUpdates( watch.getLabel(), "duration" );
37     stats::Histogram histogram( "StopWatch Duration", 20, 70, 25 );
38     for( StatisticsBuffer::UpdateDeque::size_type i = 0; i < updates.size(); ++i ) {
39         histogram.update( updates[ i ].second );
40     }
41
42     data::output::XmlReport report( "Example_Statistics.xml" );
43     report.addReportProducer( *buf );
44     report.addReportProducer( histogram );
45     report.generateReport();
46 }
```

Quelle 3.25: Verwendung der Klasse StatisticsBuffer im Zusammenspiel mit den Klassen Histogram und XmlReport zur Erstellung eines Berichts, der auch ein Histogramm enthält

Die Quelle 3.25 zeigt die main-Funktion des Beispielprogramms, in der zunächst ein StatisticsBuffer-Objekt erzeugt und als Konsument am Log-Kanal statistics registriert wird. Der dritte Parameter der statischen Methode create gibt dabei an, dass Daten zwischengespeichert werden sollen. Anschließend wird ein Zufallszahlengenerator des Typs odemx::random::Normal erzeugt, der für die Erhöhung der Simulationszeit genutzt wird. Die Werte 42 und 5 geben im Konstruktor den Mittelwert und die Standardabweichung vor.

Das nachfolgend erstellte StatisticsStopWatch-Objekt wird in einer Schleife wiederholt aufgerufen. Dabei werden die Statistik-Einträge für die gemessene Zeitdauer erzeugt, zu denen im nächsten Schritt ein Histogramm erstellt wird. Die gepufferten Daten werden mit getUpdates angefordert und dann an das Histogramm-

Objekt übermittelt. Da hierbei die in `updates[i].first` gespeicherten Zeitpunkte irrelevant sind, wird immer nur die in `updates[i].second` festgehaltene Zeitdauer verwendet. Abschließend wird ein `XmlReport`-Objekt konstruiert, dem der `StatisticsBuffer` und das Histogramm für die Erzeugung der XML-Datei `Example_Statistics.xml` übergeben werden. Beim Aufruf von `generateReport` werden die akkumulierten Daten von Histogramm und Statistik-Puffer in Tabellen eingetragen, welche im XML-Format ausgegeben werden.

3.4.7 Laufzeitvergleich des bisherigen Trace-Mechanismus und des Loggings via Kanal trace

Da die generische Logging-Bibliothek in Version 3.0 die Grundlage der ODEMx-Datenerfassung bildet und damit den bisherigen Trace-Mechanismus ersetzt, sind Laufzeitvergleiche zwischen beiden Konzepten angebracht. Der hauptsächliche Unterschied besteht dabei in der Implementation der Informationsweitergabe. Im Trace-Mechanismus wurden bisher in jeder Klasse für alle von ihr erzeugten Trace-Einträge statische `MarkType`-Objekte erstellt. Diese wurden bei jedem neuen Trace-Eintrag unverändert wiederverwendet. Die Weiterleitung dynamischer Informationen wie die Änderung von Variablenwerten wurde hingegen durch die Aufrufe vieler verschiedener Methoden an der `TraceConsumer`-Schnittstelle realisiert.

Durch die Vereinfachung der Konsumentenschnittstelle auf nur eine Methode ist es notwendig, alle durch unterschiedliche Methodenrufe bereitgestellten Informationen in den Log-Einträgen zu übermitteln. Die Konsequenz im neuen Logging-Konzept ist, dass nicht einfach unveränderliche statische Objekte verwendet werden können, sondern dass für jeden Log-Eintrag die Erzeugung eines `SimRecord`-Objekts notwendig ist. Da die Erzeugung von Objekten deutlich kostenintensiver ist als Methodenrufe an einer Schnittstelle, ist nicht zu erwarten, dass der neue Mechanismus kürzere Laufzeiten erreicht als der bisherige.

Die Laufzeitmessungen¹² wurden anhand der ODEMx-Implementation einer Variante des Philosophenproblems durchgeführt (siehe Anhang C). Dabei sitzen 800 Philosophen an einer Tafel, und ihre Denkperioden sind auf 10 Zeiteinheiten begrenzt. Die Simulationsdauer ist auf 10.000 Zeiteinheiten festgelegt. Beide Versionen der Bibliothek sowie die Simulationsprogramme wurden mit der Release-

¹² Testsystem: Pentium M mit 2,1 GHz, Ubuntu-Linux, g++-Version 4.2.4

Konfiguration kompiliert. Zu beachten ist, dass der Laufzeitvergleich lediglich die Erzeugung von Trace-Daten betrifft, weil diese bei allen Simulationsexperimenten generiert werden. In den Testprogrammen finden außerdem keine Konsumentenklassen Verwendung, weil deren unterschiedliche Implementationen und Ausgabeformate die Messungen beeinflussen würden. Die Zeitmessungen erfolgten mit dem Linux-Kommandozeilenprogramm `time`.

Messwert	ODEMx 2.2	ODEMx 3.0 (Trace aktiviert)
Reale Zeit	0m5.544s	0m13.214s
CPU-Zeit (User-Modus)	0m5.308s	0m12.605s
CPU-Zeit (Kernel-Modus)	0m0.004s	0m0.352s

Tabelle 3.6: Laufzeitmessung von Simulationsläufen mit den Versionen 2.2 und 3.0 der Bibliothek ODEMx

Tabelle 3.6 zeigt die Resultate der Programmläufe. Dabei wurde die Erwartung bestätigt, dass die Erzeugung der Trace-Daten aufgrund der objektbasierten Log-Einträge des neuen Konzepts mehr Zeit benötigen würde. Diesen Daten zufolge ist die Vorgängerversion der Bibliothek mehr als doppelt so schnell wie die aktuelle. Eine derartige Verlangsamung ist natürlich inakzeptabel, insbesondere weil die gesammelten Trace-Daten lediglich Detailinformationen zu den Interna von ODEMx enthalten und der Fehlersuche dienen.¹³

Nach bisherigen Erfahrungen werden diese Daten aber nur selten benötigt. Aufgrund der im Test beobachteten Laufzeitunterschiede ist es daher sinnvoll, dem Anwender die Möglichkeit zu geben, die automatische Erzeugung von Trace-Daten zu deaktivieren. Zur Konfiguration der Datenbankansbindung von ODEMx bezüglich ODBC oder SQLite bietet Version 3.0 bereits eine Möglichkeit, um den Kompilierungsvorgang mit Präprozessor-Direktiven zu steuern. An dieser Stelle wird nun auch ein Schalter zur Aktivierung der Trace-Einträge hinzugefügt (siehe Anhang A). Bei Deaktivierung der Trace-Funktionalität wird die gesamte Bibliothek ohne Trace-Einträge übersetzt.

In Tabelle 3.7 ist nun die Zeitmessung des Simulationslaufs nach Deaktivierung der Trace-Funktionalität dargestellt. Erneut ist ein großer Unterschied in der Laufzeit erkennbar. Das Abschalten der Trace-Einträge führt in Version 3.0 zu einer

¹³ Wegen des gewaltigen Unterschieds in der Laufzeit wurde testweise ein Zähler als Konsument registriert, der nach Ablauf der Simulation 7.202.190 Trace-Einträge vermeldete.

Messwert	ODEMx 2.2	ODEMx 3.0 (Trace deaktiviert)
Reale Zeit	0m5.544s	0m2.181s
CPU-Zeit (User-Modus)	0m5.308s	0m1.884s
CPU-Zeit (Kernel-Modus)	0m0.004s	0m0.256s

Tabelle 3.7: Laufzeitmessung nach Deaktivierung der Trace-Einträge von ODEMx Version 3.0

deutlich verbesserten Simulationslaufzeit, auch im Vergleich zur Vorgängerversion. Der Leistungsgewinn hat allerdings seinen Preis darin, dass die Bibliothek neu übersetzt werden muss, wenn die Trace-Ausgaben zur Fehleranalyse benötigt werden.

3.5 Zusammenfassung

Die bei der Konzeption der generischen Logging-Bibliothek angestrebten Zielsetzungen wurden weitestgehend erreicht. Die Grundstruktur des ODEMx-Trace-Mechanismus konnte erfolgreich in einer generalisierten Form implementiert werden. Dadurch bietet die im Rahmen dieser Arbeit geschaffene Logging-Bibliothek die Möglichkeit, neben einer Trace-Protokollierung auch beliebige weitere Logging-Kategorien zu definieren. Weiterhin wurden Basiskomponenten für Produzenten- und Konsumentenklassen erstellt, wobei auch die angestrebte Vereinfachung der Konsumentenschnittstelle umgesetzt wurde. Durch die Vorgabe einer einzigen rein virtuellen Methode wird die Definition neuer Ausgabekomponenten deutlich erleichtert.

Das Ziel der Unterstützung beliebiger Log-Datentypen konnte in der Implementation des Logging-Konzeptes durch die Verwendung von C++-Templates realisiert werden. Die gewählte Lösung ermöglicht eine vielseitige Verwendung der Logging-Bibliothek auf typsichere Art und Weise, weil der Compiler Fehlermeldungen erzeugt, wenn der Log-Datentyp kombinierter Komponenten nicht kompatibel ist.

Ein hoher Grad an Generizität kann aber auch Nachteile haben, da für den Nutzer oft ein Mehraufwand in der Konfiguration solcher Komponenten entsteht und auch die Einarbeitung in bestimmte Konzepte aufwändiger sein kann. Ein generelles Problem, auf das Bibliotheksentwickler bei generischen Komponenten stoßen, sind fehlende Informationen über die vom Anwender festzulegenden Datentypen. Bezüglich der Templateparameter sollten so wenige Annahmen wie möglich gemacht

werden. In manchen Fällen werden daher auch nur Schnittstellen vorgegeben, die der Anwender implementieren beziehungsweise spezialisieren muss. Beispiele dafür sind die Filter- und die Konsumentenschnittstelle der Logging-Bibliothek. Zur Verringerung der Komplexität und des Aufwands wurde mit der Logging-Bibliothek jedoch ein Mittelweg gewählt, indem sie die Verwendung auf mehreren Nutzungsebenen unterstützt. So werden neben den generalisierten Komponenten auch Default-Komponenten angeboten, mit deren Hilfe die Verwendung der Bibliothek erleichtert wird.

Neben den standardmäßig für den Log-Datentyp Record angebotenen Komponenten wird auch eine Zugriffsschicht für RDBMS bereitgestellt, welche die Implementation von Konsumentenklassen mit Datenbankbindung vereinfacht und gleichzeitig vom SQL-Dialekt des spezifischen RDBMS abstrahiert. Auf dieser Grundlage wurde in ODEMx die Klasse `DatabaseWriter` implementiert, mit der ODEMx-Simulationsprogramme eine Datenbankbindung erhalten. Leider mussten an dieser Stelle Einschränkungen für spezifische RDBMS gemacht werden, da die parallele Nutzung einer bestimmten Datenbank durch mehrere Simulationsprogramme nicht plattformübergreifend durch die gleichen SQL-Anweisungen realisierbar ist. Bei der Anbindung via ODBC muss daher die spezifische Datenbank durch eine Präprozessor-Direktive bekannt gemacht werden.

Eine weitere Zielstellung bei der Konzeption der Logging-Bibliothek war die möglichst einfache Einbindung und Verwendung in beliebigen C++-Projekten. Der Anwendungsfall ODEMx hat gezeigt, dass das Format einer header-only Bibliothek die Integration der Logging-Bibliothek extrem vereinfacht, da neben simplem Kopieren und Einbinden der Header-Dateien keine weiteren Arbeitsschritte notwendig sind. Auch die erwähnten Default-Komponenten tragen ihren Teil zur unkomplizierten Verwendung bei, da die Bibliothek durch ihren eigenen Log-Datentyp gleich Konsumenten für die einfache Text- und XML-Ausgabe anbieten kann.

In ODEMx ersetzt die Logging-Bibliothek den bisher verwendeten Trace-Mechanismus und fügt einige Erweiterungen hinzu. Auf der Grundlage einer speziellen Produzentenbasisklasse stellen nahezu alle ODEMx-Komponenten in Version 3.0 die Logging-Funktionalität in sieben Kategorien bereit. Damit können Anwender von ODEMx in ihren abgeleiteten Klassen auf einfache Art und Weise Simulationsdaten protokollieren, wobei erstmals auch die Fehlerausgabe und die Statistiksammlung durch die Logging-Komponenten übernommen werden.

Aufgrund der einfachen Konsumentenschnittstelle der Logging-Bibliothek ver-

lief die Reimplementation der Trace-Konsumentenklasse `XmlTrace` in Form der Klasse `XmlWriter` problemlos. Auch die neu geschaffenen Konsumentenklassen `OStreamWriter`, `DatabaseWriter` und `StatisticsBuffer` verlangten relativ geringen Implementationsaufwand.¹⁴ Etwas größere Umbauten waren im Bereich der Statistik notwendig, da die Statistikproduzenten zuvor nur aggregierte Daten gesammelt haben. Durch Ableitung von `odemx::data::Producer` erhalten diese Klassen nun Zugriff auf den Log-Kanal `statistics`, über den jede einzelne statistisch relevante Wertänderung versendet werden kann. Statistische Kenngrößen können damit in einer separaten Auswertungsphase aus archivierten Simulationsdaten gewonnen werden. Diese Berechnung kann aber auch wie bisher während der Laufzeit geschehen, indem beispielsweise ein `StatisticsBuffer`-Objekt als Konsument verwendet wird.

Auch wenn die konzeptuellen Zielsetzungen mit der Implementation der Logging-Bibliothek erreicht wurden, so gibt es dennoch Verbesserungsmöglichkeiten. Dies wurde während der im Abschnitt 3.4.7 beschriebenen Laufzeitanalyse deutlich. Offensichtlich stellt die Erzeugung sehr vieler `SimRecord`-Objekte einen Kostenfaktor dar, der möglicherweise in der Zukunft noch etwas optimiert werden kann.

¹⁴ Die Implementationen dieser Klassen umfassen jeweils weniger als 400 Zeilen Quellcode.

4 Protokollsimulation

4.1 Einführung

4.1.1 Begriffsklärung

Ein Großteil der Kommunikation zwischen Menschen wird heutzutage über viele verschiedene Rechner- und Kommunikationsnetzwerke abgewickelt. Computer mit Internet-Anschluss und Mobiltelefone sind für viele Menschen mittlerweile unentbehrliche Werkzeuge zur Erledigung ihrer täglichen Arbeit. Die Basis für die Realisierung dieser allgegenwärtigen Kommunikationssysteme sind Kommunikationsprotokolle.

Ein **Kommunikationsprotokoll** beschreibt Verhaltenskonventionen, die vorgeben, auf welche Weise Kommunikationssysteme miteinander kommunizieren. Dies geschieht durch Festlegung von Datenformaten und einer zeitlichen Abfolge der Interaktionen zwischen Kommunikationspartnern. Zur Erbringung der durch ein Kommunikationsprotokoll beschriebenen Funktionalität sind allerdings eine Vielzahl unterschiedlicher Funktionen notwendig: das Senden und Empfangen von Daten, ihre Kodierung und Dekodierung, das Erkennen und Beheben von Übertragungsfehlern, die Steuerung der Datenflüsse zwischen Kommunikationspartnern und die Sicherung einer bestimmten Übertragungsqualität. Zur besseren Handhabung der hohen Komplexität werden in Kommunikationssystemen typischerweise aufeinander abgestimmte Protokollhierarchien eingesetzt, in denen jedes einzelne Protokoll lediglich eine Teilfunktion erbringt [König03].

4.1.2 Modellierung von Kommunikationsprotokollen

Nach König [König03] umfasst der allgemein verwendete Protokollbegriff zwei Konzepte. Das erste ist der sogenannte **Dienst** (*service*), eine Funktion eines

Rechnernetzwerks oder Kommunikationssystems zur Erbringung einer Dienstleistung im Netz. Dienste können von Anwendungen oder auch Protokollen genutzt werden. Abbildung 4.1 zeigt das Modell eines Kommunikationsdienstes.

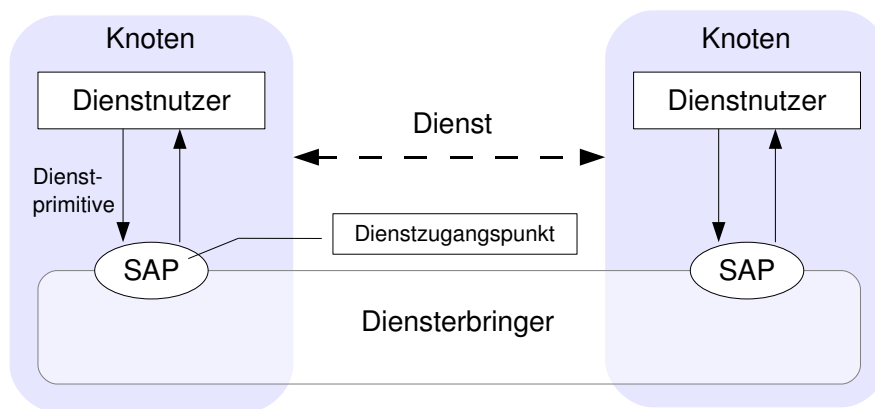


Abbildung 4.1: Modell eines Kommunikationsdienstes

Die Verwendung eines Dienstes erfordert mindestens zwei Dienstanwender, die den Dienst über eine vordefinierte Dienstschnittstelle, den sogenannten **Dienstzugangspunkt** (*service access point, SAP*), in Anspruch nehmen. SAPs sind asynchrone Kommunikationsschnittstellen, welche die Interaktion zwischen Dienstanwender und Dienstbringer über Warteschlangen realisieren. Der Datenaustausch zwischen Dienstanwender und Dienstbringer wird anhand verschiedener sogenannter **Dienstprimitive** realisiert, die von der konkreten Implementation des Dienstes abstrahieren. Dienstprimitive werden durch eine Bezeichnung, ihren Typ sowie die zugehörigen Parameter spezifiziert. Ein Dienstanwender verwendet einen Dienst, indem sogenannte **request**-Primitive an den entsprechenden SAP übergeben werden. Der Dienstbringer realisiert die Kommunikation zwischen Dienstanwendern durch die Übergabe von **response**-Primitiven an SAPs der Kommunikationspartner. Die Kommunikationsabläufe an der Dienstschnittstelle sowie die verfügbaren Dienstprimitive und deren Parameter werden durch eine Dienstspezifikation vorgegeben.

Das zweite Konzept ist das **Protokoll**, welches die Art und Weise der Erbringung eines angebotenen Dienstes beschreibt. Umgesetzt wird diese Aufgabe von den sogenannten **Instanzen** (*entities*). Dies sind aktive Objekte des Dienstbringers, welche durch Nachrichtenaustausch mit ihrer Umgebung interagieren. Instanzen

können untereinander kommunizieren und bedienen die SAPs. Ein SAP ist dabei immer genau einer Instanz zugeordnet, wogegen eine Instanz auch mehrere SAPs bedienen darf. Zur Erbringung des Dienstes nehmen Instanzen die anliegenden Dienstprimitiven entgegen, analysieren sie und kommunizieren mit der sogenannten **Partnerinstanz** (*peer entity*), die dem SAP des Kommunikationspartners zugeordnet ist. Diese Wechselwirkung zwischen den Instanzen folgt festen Regeln, die durch das Protokoll vorgegeben sind. Es beschreibt eine zeitliche Abfolge der Interaktion zwischen den dienstbringenden Instanzen sowie die Syntax und Semantik der auszutauschenden Nachrichten. Abbildung 4.2 zeigt die Konzepte Dienst und Protokoll vereint in einem Modell.

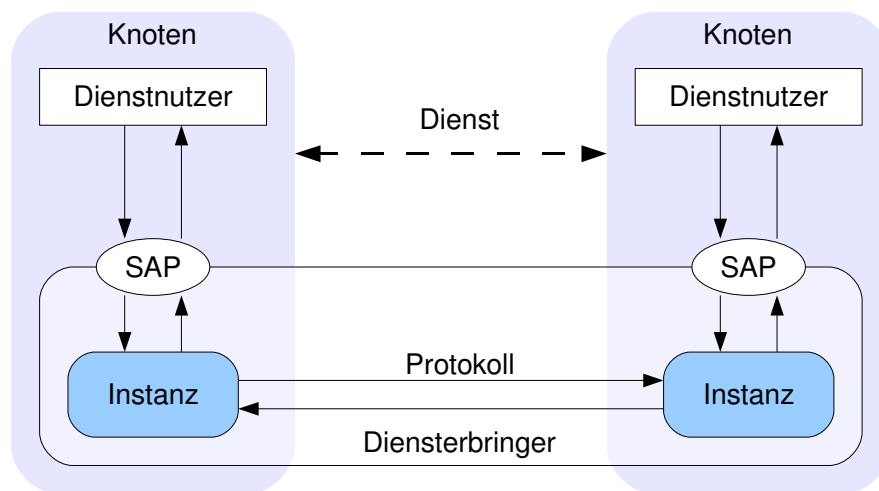


Abbildung 4.2: Kombiniertes Dienst- und Protokollmodell (eine Schicht)

Die zwischen Partnerinstanzen übermittelten Nachrichten werden als **Protokoll-dateneinheiten** (*protocol data unit, PDU*) bezeichnet. Meist werden in einem Protokoll verschiedene PDUs verwendet – beispielsweise für Verbindungsaufbau, Datentransfer und Verbindungsabbau. Eine **Verbindung** ist eine logische Beziehung zwischen den Dienstanwendern an eindeutig adressierten SAPs, die eine sichere Übertragung von Daten und die Wahrung der Übertragungsfolge gewährleistet. Die Verwaltung der Verbindungen übernehmen die zu den SAPs gehörigen Instanzen. Über jede Verbindung läuft nebenläufig das gleiche Protokoll ab, so dass sich diese nicht beeinflussen.

Wie bereits erwähnt, findet bei Kommunikationsprotokollen meist eine Aufteilung der Teilfunktionen auf mehrere Protokolle statt, die in einer Hierarchie angeord-

net werden. Für diesen Zweck hat sich eine Schichtenstruktur durchgesetzt. Eine solche **Schicht** (*layer*), wie sie in Abbildung 4.2 dargestellt ist, kann einen oder mehrere Dienste bereitstellen, und sie umfasst alle Instanzen, die die gleiche funktionale Zielstellung haben, also die gleichen Dienste erbringen. Zur Kommunikation untereinander greifen Instanzen einer Schicht jeweils auf die Dienste der darunter liegenden Schicht zu. Der Nachrichtenaustausch zwischen dienstbringenden Instanzen eines Dienstes verläuft also nur theoretisch auf derselben Ebene, denn auf Senderseite werden PDUs von oben nach unten über SAPs durch die Schichten gereicht, während sie auf Empfängerseite von unten nach oben über SAPs weitergegeben werden. Innerhalb jeder Schicht erfüllt die dem SAP zugeordnete Instanz ihre Funktion und reicht die Nachricht dann weiter. Die komplette Protokollhierarchie, welche alle an der Kommunikation beteiligten Schichten umfasst, wird als **Protokollstack** bezeichnet.

4.1.3 Simulation von Kommunikationsprotokollen

Aufgrund der rasanten Entwicklung drahtloser Kommunikationssysteme ist ein ganzes Forschungsgebiet entstanden, das sich mit der Analyse des Verhaltens und der Leistungsfähigkeit solcher Systeme befasst. Durch zunehmende Komplexität und gesteigerte Anforderungen erhöht sich auch der Aufwand für den Entwurf und die Analyse von Kommunikationssystemen. Aus diesem Grund hat sich die Simulation in den vergangenen Jahren auch in dieser Branche zu einem alltäglichen Werkzeug entwickelt. Tabelle 4.1 zeigt, dass die Simulation von Kommunikationssystemen je nach Detailgrad durch verschiedene Arten von Simulatoren umgesetzt wird [Jeruch00].

Detailgrad	Untersuchungsziel	Art des Simulators
Netzwerk	Protokollverhalten und Nachrichtenfluss	ereignisgetriebene Simulatoren
Links	Wellenformen, Verzerrungen, Rauschen und Interferenzen	zeitgetriebene Wellenform-Simulatoren
Signalverarbeitung	Implementierungsdetails von Algorithmen	zeitgetriebene Simulatoren mit begrenzter Genauigkeit
Schaltkreisanalyse	Schaltkreissimulation	Schaltkreissimulatoren

Tabelle 4.1: Verschiedene Detailgrade bei der simulativen Untersuchung von Kommunikationssystemen

Auf der Ebene des Netzwerks, welches aus kommunizierenden Knoten besteht, werden das Protokollverhalten und der Nachrichtenfluss typischerweise mit ereignisgetriebenen Simulationswerkzeugen wie ODEMX untersucht. Meist wird dabei von den weiteren aufgelisteten Detailebenen abstrahiert, oder es werden Ergebnisse von Simulationsexperimenten anderer Detailebenen in die Netzwerksimulation mit einbezogen. Da das Protokollmodul von ODEMX auf die Simulation der Netzwerkebene ausgelegt ist, finden die anderen Detailebenen in dieser Arbeit keine weitere Betrachtung. Ausführliche Abhandlungen dieser Themen findet der geeignete Leser in [Jeruch00] oder auch [Tranter04].

Wie bereits oben erwähnt, ist die Entwicklung von Kommunikationsprotokollen ein sehr aufwändiger Prozess, bei dem die Simulation in verschiedenen Entwicklungsstadien als wertvolles Werkzeug dienen kann, um Fehler im Entwurf oder unerwartetes Verhalten aufzudecken. Neben der Validierung des Protokollverhaltens kann mit Hilfe von Simulationsexperimenten auch die Leistungsfähigkeit bereits existierender oder geplanter Netzwerke untersucht werden.

Die Auswahl an Simulationswerkzeugen in diesem speziellen Bereich ist trotz steigenden Interesses recht überschaubar – prominente kommerzielle Vertreter dieser Kategorie sind OPNET Modeler [OPNET] und QualNet Developer [QualNet]. Unter den quelloffenen Simulationswerkzeugen ist ns-2 (Network Simulator Version 2) die mit Abstand meistgenutzte Simulationssoftware für Kommunikationsnetzwerke. Im letzten Jahrzehnt haben viele verschiedene Forschungsgruppen Erweiterungsmodule zu diesem Projekt beigetragen [Issari08]. Allerdings hat auch dieses Simulationswerkzeug mehrere Schwächen, weshalb bereits seit dem Jahr 2006 eine Nachfolgeversion unter dem Namen ns-3 in Entwicklung ist [Hender06]. Eine weitere populäre quelloffene Simulationsbibliothek ist OMNeT++, deren Schwerpunkt auf der Simulation von Kommunikationsnetzwerken und verteilten Systemen liegt. Domänenspezifische Modelle und Komponenten sind jedoch nicht Teil von OMNeT++, denn diese werden durch andere Forschungsgruppen in Eigenregie entwickelt und beigesteuert [Varga08].

4.2 Das ODEmX-Protokollsimulationsmodul

Angestoßen wurde die Entwicklung eines ODEmX-eigenen Protokollmoduls im Jahr 2007. Es sollte eine Möglichkeit geschaffen werden, eigene Ergebnisse von Protokollsimulationen mit jenen zu vergleichen, die mit anderen Simulationswerkzeugen gewonnen wurden. Ausgangspunkt für die erste Implementation des Protokollmoduls war die Portierung einer in der Spezifikationssprache SDL vorliegenden Protokollbeschreibung nach ODEmX. Aufbauend auf diesen Erfahrungen wurden unter Verwendung der in Abschnitt 4.1.2 aufgeführten Prinzipien des Protokollentwurfs Basiskomponenten für ODEmX entwickelt, welche eine schichtorientierte Protokollmodellierung unterstützen. Dieser Abschnitt beschreibt zunächst das in der ersten Version des Protokollmoduls verwirklichte Konzept und skizziert die ursprüngliche Implementation.

4.2.1 Konzept

Aufgabe des Moduls war und ist die Bereitstellung von Bausteinen für die Simulation des Verhaltens von Kommunikationsprotokollen in beliebig strukturierten Netzwerken. Die bereitgestellten Basisklassen unterstützen den Anwender bei der Definition von Protokollinstanzen, der Erzeugung von Protokollstacks und dem Aufbau der Netzwerktopologie. Zudem wird durch das sogenannte Übertragungsmedium eine mögliche Implementation für die Kommunikation der Knoten untereinander angeboten, so dass sich der Anwender nur dann damit befassen muss, wenn eine andere Funktionalität gewünscht ist. Abbildung 4.3 zeigt die Grundidee für den Ablauf von Kommunikationsvorgängen zwischen Netzwerkknöten.

Die Anwendungsschicht eines Netzwerkknöten greift auf den Protokollstack zu, um Daten zu versenden. Die Protokollinstanz der obersten Schicht des Stacks erhält die Dateneinheit, fügt ihre eigenen Protokollinformationen hinzu, und übergibt diese modifizierte Protokolldateneinheit an die nächsttiefere Schicht. Dies wiederholt sich bis zur untersten Schicht im Protokollstack, deren Protokollinstanz ihre Dateneinheit automatisch an das Übertragungsmedium weiterleitet. Das Medium erzeugt in Abhängigkeit von den in der Dateneinheit enthaltenen Informationen und der Topologie des Netzwerks Simulationsergebnisse entsprechend der Anzahl der Empfänger. Diese Simulationsergebnisse realisieren die Übertragung

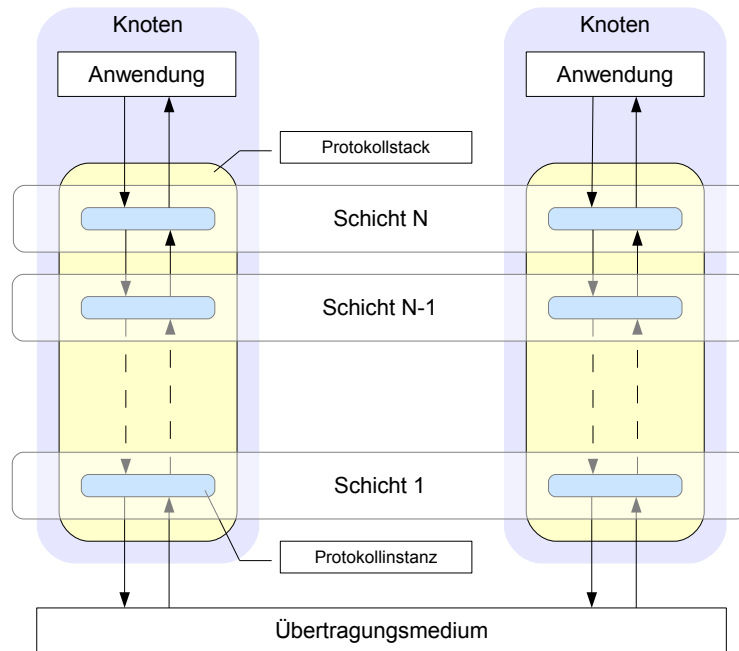


Abbildung 4.3: Kommunikation zwischen Anwendungen, die auf zwei unterschiedlichen Netzknoten laufen

der Dateneinheit, indem sie nach Ablauf der Übertragungsdauer ausgeführt werden und dabei die versendete Dateneinheit zum Empfangspuffer des adressierten Protokollstacks hinzufügen. Im empfangenden Knoten durchläuft die Dateneinheit erneut eine oder mehrere Schichten, wobei die einzelnen Instanzen die Dateneinheit ihrem Protokoll entsprechend bearbeiten. Durch die oberste Schicht des Protokollstacks können dann die empfangenen Anwendungsdaten an die auf diesem Knoten aktive Anwendung übergeben werden.

4.2.2 Softwaretechnische Umsetzung in Version 2.2 von ODEmX

Dieser Abschnitt beschreibt die erste Implementation des ODEmX-Protokollsimulationsmoduls, welche das im vorigen Abschnitt beschriebene Konzept in ODEmX-Komponenten umsetzt. Zu allen von dem Modul bereitgestellten Komponenten wird an dieser Stelle eine kurze Erläuterung gegeben, um die Grundlagen für die im Anschluss dargelegte Problemanalyse dieser Implementation zu vermitteln.

Simulationskontext

In Version 2.2 wird für die Protokollsimulation mit ODEmX die Nutzung eines speziellen Simulationskontextes verlangt, der durch die von `Simulation` abgeleitete Klasse `ProtocolSimulation` gegeben ist. Diese Klasse ermöglicht anderen Komponenten den Zugriff auf einzelne Schichten, das Übertragungsmedium und die Netzwerktopologie. Da ODEmX-Protokollimplementationen in Schichten gegliedert sein müssen, gibt es das Methodentemplate `createLayer`, mit dem sich neue Schichten erstellen lassen. Jede Protokollsimulation verwendet nur ein Übertragungsmedium, das die gesamte Kommunikation steuert. Erzeugt wird ein Medium nutzerdefinierten Typs durch Aufruf des Methodentemplates `createMedium`. Sowohl das Medium als auch die Schichten werden nach ihrer Erzeugung durch den Simulationskontext verwaltet.

Schicht

Da die Protokollinstanzen der Schichten die aktiven Komponenten sind, welche die Dienstfunktionalität erbringen, muss der Protokollstack mindestens eine Schicht enthalten. Diese Schichten werden durch die Klasse `ProtocolLayer` implementiert, welche Instanzen mit der Methode `createNewEntity` erzeugt und in einer Liste verwaltet. Durch Aufruf des Methodentemplates `createLayer` des Simulationskontextes werden neue Schichten erzeugt. Der zugehörige Instanztyp muss dabei als Templateparameter angegeben werden, damit die Schicht neue Instanzen dieses Typs erzeugen kann. Auf diese Weise können gleichartige Protokollstacks automatisch von ODEmX konstruiert werden, indem die Erzeugung der Instanzen durch die registrierten `ProtocolLayer`-Objekte vorgenommen wird.

Protokollstack

Die Knoten eines Netzwerks nutzen Objekte der Klasse `ProtocolStack` als Kommunikationsschnittstelle zu weiteren im Netzwerk befindlichen Knoten. Die Topologie des Netzwerks wird in Version 2.2 von ODEmX durch Nachbarschaftsbeziehungen zwischen den `ProtocolStack`-Objekten gespeichert, wobei jeder Stack eine ID erhält. Entsprechend der vom Nutzer angegebenen Schichtenstruktur erzeugt ein `ProtocolStack`-Objekt bei seiner Konstruktion für jede gegebene Schicht eine Protokollinstanz und konfiguriert diese so, dass die vertikale Kommunikation

innerhalb des Stacks funktioniert. Über die Methode `inputData` werden Anwendungsdaten gesendet, und über die Methode `receivedTransmission` wird eine Übertragung vom Netzwerk empfangen. Die Ausgabe der empfangenen Daten an eine Anwendung erfolgt über die rein virtuelle Methode `receivedData`. Daher muss der Anwender in Version 2.2 von ODEMX Spezialisierungen dieser Klasse definieren, welche durch die Implementation der Methode `receivedData` bestimmen, was mit den von der obersten Schicht des Stacks erhaltenen Anwendungsdaten zu tun ist.

Protokollinstanz

Instanzen sind die aktiven Komponenten des Protokollmodells, welche die von einer Schicht angebotenen Dienste erbringen. Die entsprechende Basisklasse `ProtocolEntity` ist daher von der Klasse `Process` abgeleitet. Die Repräsentation der von der Instanz bedienten SAPs erfolgt durch jeweils einen Datenpuffer für Dateneinheiten von der oberen sowie der unteren Schicht. Das Prozessverhalten ist definiert als dauerhafter Wartezustand, solange keine Daten verfügbar sind. Bei Datenerhalt wird der Prozess sofort aktiviert und abhängig davon, an welchem Puffer gerade Daten eingegangen sind, wird eine der beiden rein virtuellen Methoden `handleUpperLayerInput` oder `handleLowerLayerInput` gerufen. Die Verhaltensdefinition der Instanzen obliegt damit dem Anwender, der diese beiden Methoden in abgeleiteten Klassen implementieren muss.

Topologie

In Version 2.2 von ODEMX wird wie Topologie eines Netzwerks mit Hilfe der Klasse `NetTopology` beschrieben. Jede Protokollsimulation verfügt über genau eine Topologie, an der alle Netzwerkknotten durch Aufruf von `registerNode` registriert werden. Die Verbindungen der Netzwerkknotten untereinander werden als **Links** bezeichnet. Alle Knotten werden in einer `std::map` vorgehalten, in der die IDs der Protokollstacks als Schlüssel verwendet werden. Die zu jedem Knoten gespeicherten Informationen umfassen neben der ID auch eine Positionsangabe, einen Zeiger auf das `ProtocolStack`-Objekt und eine weitere `std::map` mit den Links zu benachbarten Knotten. Die mittels `setLink` gespeicherten Links zwischen Knotten verfügen jeweils über eine Zufallsvariable, mit deren Hilfe der Verlust von

Nachrichten durch Angabe einer bestimmten Wahrscheinlichkeit modelliert werden kann.

Übertragungsmedium

Eine zentrale Komponente des Protokollmoduls ist das durch die Klasse `TransmissionMedium` implementierte Übertragungsmedium, wovon es in jeder Protokollsimulation genau eines gibt. Die gesamte Kommunikation zwischen Netzwerknoten läuft über dieses Objekt, weil die Instanzen der untersten Schicht eines Protokollstacks automatisch die zu übertragenden Daten an das Medium weiterreichen. Aus diesem Grund wird hier auch durch Ableitung von der Klasse `NetTopology` die Topologie des Netzwerks verwaltet, da die Verbindungen zwischen den Knoten bekannt sein müssen, um die Weiterleitung von Nachrichten zu bewerkstelligen. Der Vorteil eines solchen zentralen Verteilers ist die vereinfachte Protokollierung der Kommunikation und die Sammlung statistischer Daten.

Protokolldateneinheiten

Die in Protokollsimulationen verwendeten Nachrichtentypen müssen die vorgegebene Schnittstelle `ProtocolMessage` implementieren, damit das Übertragungsmedium mit ihnen arbeiten kann. Die dadurch bereitgestellten Informationen umfassen Sender und Empfänger einer Nachricht, die Unterscheidung nach Broadcast (Nachricht an alle Nachbarknoten) oder Unicast (Nachricht an einen Nachbarknoten), die Übertragungsdauer sowie eine String-Repräsentation. Besonders wichtig ist auch die Möglichkeit, eine komplette Kopie anzufordern, damit eine Nachricht gleichzeitig durch verschiedene Knoten bearbeitet werden kann, ohne dass diese Seiteneffekte auslösen.

4.3 Ansatz zur Erweiterung des Protokollsimulationsmoduls

Die erste Version des Protokollsimulationsmoduls findet in verschiedenen Projekten Verwendung. In der aktuellen Forschung des Lehrstuhls Systemanalyse werden

beispielsweise ODEmx-Simulatoren zur Untersuchung eines Alarmierungsprotokolls in selbstorganisierenden Sensornetzwerken eingesetzt [Fischer08]. Das Protokollmodul wird dabei zur Simulation einer Kommunikationsschicht zwischen den Netzknoten verwendet. Aus dieser Anwendung ergab sich die Erkenntnis, dass eine Modellbildung mit verschiedenen Abstraktionsgraden wünschenswert wäre, damit die Erweiterung des Protokollstacks iterativ vollzogen werden kann.

Im Rahmen eines Projektpraktikums, in dem ein TCP/IP-Stack nachgebildet werden sollte, wurden verschiedene weitere Defizite des Protokollsimulationsmoduls lokalisiert. Dabei stellte sich heraus, dass die Modellierungsfreiheit für Protokollentwickler unnötig eingeschränkt wurde. Analog zur Gliederung des Abschnitts 4.2.2 werden die festgestellten Mängel in diesem Abschnitt beschrieben und entsprechende Lösungsansätze vorgestellt. Ihre softwaretechnische Umsetzung wird dann im anschließenden Abschnitt dargelegt.

4.3.1 Problemanalyse der bisherigen Implementation

Simulationskontext

Hauptgründe für die Schaffung eines eigenen Simulationskontextes für Protokollsimulationen waren das Versenden von Nachrichten via Übertragungsmedium sowie die automatische Initialisierung von Protokollstacks. Der Simulationskontext stellte bisher alle Schichten bereit, wobei diese auf Anforderung eines Protokollstacks eine neue Instanz erzeugten. Instanzen der untersten Schicht wiederum nutzten den Simulationskontext, um Zugriff auf das Übertragungsmedium zu erhalten. Durch diese interne Automatik ergab sich allerdings eine stark eingeschränkte Modellierungsfreiheit in der Konfiguration der Protokollstacks, wie im Folgenden noch dargelegt wird. Als Lösungsansatz wird auf derartige interne Automatismen verzichtet, womit die Notwendigkeit für einen speziellen Simulationskontext entfällt. In der Folge sind in Version 3.0 von ODEmx nun auch Protokollsimulationen mit beliebigen Simulationskontexten und insbesondere auch dem Default-Simulationskontext realisierbar.

Schicht

Durch das Methodentemplate `ProtocolSimulation::createLayer` wurde bisher jeder Schicht bei ihrer Erzeugung der Klassentyp ihrer zugehörigen Protokollinstanzen übergeben. Aus diesem Grund konnten Schichten allerdings nur gleichartige Instanzen dieses einen Typs erzeugen, womit die Modellierungsfreiheit stark eingeschränkt wurde. Jede Schicht konnte so zwar immer noch mehrere Dienste anbieten, jedoch musste die gesamte Funktionalität durch den einen Instanztyp der Schicht erbracht werden. Als Lösungsansatz für dieses Problem wird die gemeinsame Basisklasse `ServiceProvider` für Dienstbringer eingeführt, so dass unter Ausnutzung von Polymorphie Objekte verschiedener Typen in einer Schicht verwaltet werden können, die jeweils bestimmte Dienste implementieren.

Die Erzeugung der Protokollinstanzen durch die Schichten führte darüber hinaus noch zu anderen Beschränkungen, da die interne Realisierung dieses Mechanismus auf C++-Templates basierte und Konstruktoren mit einheitlicher Signatur verlangte. So wurde die Konfiguration der Protokollinstanzen unnötig erschwert, weil ein Zugriff auf einzelne Instanzen durch den Anwender nicht vorgesehen war. Als Lösungsansatz wird dem Anwender daher von vornherein die Erzeugung der Protokollinstanzen nach seinen eigenen Vorstellungen überlassen. Die initialisierten Objekte kann er dann an der jeweiligen Schicht registrieren.

Protokollstack

Die Initialisierung der Instanzen des Protokollstacks wurde bislang automatisch mit Hilfe der vom Simulationskontext bereitgestellten Schichten umgesetzt, wodurch es innerhalb einer Protokollsimulation immer nur eine Art von Stack geben konnte, die alle Knoten des Netzwerks für die Kommunikation verwenden mussten. Gemischte Netzwerke mit unterschiedlich strukturierten Protokollstacks ließen sich so nicht modellieren. Als Lösungsansatz wird daher auf die automatische Initialisierung verzichtet und dem Anwender die Kontrolle über die Konfiguration der Stacks gegeben. Anwender erstellen dann die Schichten, konfigurieren sie wie oben beschrieben mit den entsprechenden Dienstbringern und registrieren die Schichten am Stack. So können beliebig strukturierte Protokollstacks in einem Simulationsprogramm verwendet werden.

In Version 2.2 von ODEmx war der Protokollstack durch eine abstrakte Klasse mo-

dellet, die dem Anwender das Schreiben einer eigenen Ableitung abverlangte. Es musste die rein virtuelle Methode `receivedData` implementiert werden, um dem Empfang von Nutzerdaten eine Aktion zuzuordnen. Als typische Implementation hat sich dafür die Weitergabe der Nachrichten an einen Empfangspuffer herausgestellt. Als Lösungsansatz wird die Klasse dahingehend umgestaltet, dass der Stack fortan die Möglichkeit bietet, für den Empfang von Daten weitere SAPs mit eigenem Puffer anzulegen. An diesen SAPs kann ein Prozess der Anwendungsschicht dann auf die Ankunft von Daten warten. Dadurch wird die Methode `receivedData` nicht mehr benötigt, und die Notwendigkeit einer Spezialisierung entfällt.

Protokollinstanz

Im ursprünglichen Entwurf des Protokollmoduls konnte eine Protokollinstanz lediglich zwei Dienstzugangspunkte bedienen, nämlich einen von der oberen und einen weiteren von der unteren Schicht. Diese SAPs wurden durch zwei zur Instanz gehörige Nachrichtenpuffer modelliert. Aufgrund dieser Limitierung von Instanzen erfolgte zwischenzeitlich eine Erweiterung des Konzepts, bei der SAPs durch Bezeichner repräsentiert und diese anhand einer `std::map` mit den jeweiligen Puffern der Protokollinstanzen assoziiert wurden. So konnten auch mehrere SAPs auf den gleichen Puffer verweisen, was der Protokollinstanz die Möglichkeit gab, dynamisch SAPs zu erzeugen und mehrere Dienste bereitzustellen. Da die Nachrichten verschiedener SAPs so allerdings über den gleichen Puffer empfangen wurden, war die Identifikation des genutzten SAPs umständlich und erforderte mehr Aufwand vom Anwender. Als Lösungsansatz wird für Dienstzugangspunkte eine eigene Klasse `Sap` eingeführt, deren Objekte neben dem Bezeichner des SAPs jeweils einen eigenen Datenpuffer enthalten. Dadurch wird konzeptuell eine bessere Trennung von SAP und Protokollinstanz erreicht und die Identifikation von SAPs beim Nachrichtenempfang erleichtert.

Ferner fehlte dem Protokollmodul in Version 2.2 von ODEMX eine Möglichkeit der Abstraktion von Protokollinstanzen, Protokollstacks und der Struktur des Netzwerks. All diese Komponenten wurden selbst dann benötigt, wenn lediglich ein einfacher Dienst mit Direktverbindung zwischen allen Knoten modelliert werden sollte. Als Lösungsansatz wird daher von den Instanzen abstrahiert und die oben schon erwähnte Basisklasse `ServiceProvider` für Dienstleister eingeführt. Diese Grundlage bietet eine bessere Unterstützung bei der Entwicklung von Proto-

kollmodellen mit unterschiedlichem Detailgrad. In Version 3.0 von ODEMX werden dafür drei konkrete Ableitungen dieser Klasse vorgegeben: die von der Netzwerkstruktur abstrahierende Dienstklasse `Service`, die Klasse `Entity` für beliebige Protokollinstanzen und die Klasse `Device` für Netzwerkschnittstellen, die mit dem Übertragungsmedium kommunizieren.

Topologie

In der bisherigen Repräsentation der Netzwerktopologie wurden Protokollstacks mit den einzelnen Knoten gleichgesetzt, weshalb die Links als Verbindungen zwischen `ProtocolStack`-Objekten gespeichert wurden. Dieses Modell ist allerdings nur dann korrekt, wenn jeder Knoten genau eine Netzwerkschnittstelle hat, über die er mit anderen Knoten kommunizieren kann. Im bisherigen Konzept des Protokollstacks mit einer Instanz pro Schicht war dies auch der Fall. In realen Netzwerken können jedoch durchaus mehrere Netzwerkschnittstellen an einem Knoten vorhanden sein. Als Lösungsansatz hilft auch an dieser Stelle die zuvor beschriebene Unterstützung für mehrere Diensterbringer pro Schicht weiter. So können an der untersten Schicht einfach mehrere `Device`-Objekte registriert werden, die unterschiedliche Übertragungsmedien nutzen. Links in der Topologie werden dabei als Verbindungen zwischen den Netzwerkschnittstellen registriert.

Die Topologie unterstützte schon in Version 2.2 eine einfache Form der Fehlermodellierung auf der Basis von Zufallszahlengeneratoren. Zu diesem Zweck enthielt jede Link-Beschreibung ein `ODEMX-Draw`-Objekt, das bei jeder Nachrichtenübertragung abgefragt wurde, um den zufälligen Verlust von Nachrichten zu modellieren. In anderen Simulationsbibliotheken wie `ns-2` gibt es jedoch deutlich mehr Fehlermodelle, die auch noch austauschbar sind [Issari08]. Als Lösungsansatz wird daher eine Verallgemeinerung des Konzepts vorgenommen, indem die Schnittstelle `ErrorModel` für die Fehlermodellierung bereitgestellt wird. Anstelle des `Draw`-Objekts verweist die Link-Beschreibung dann auf eine `ErrorModel`-Spezialisierung. Da diese Schnittstelle auf vielfältige Weise implementiert werden kann, sind so auch komplexere Fehlverhalten modellierbar.

Übertragungsmedium

Da das Übertragungsmedium bisher vom Simulationskontext verwaltet wurde, gab es in früheren Protokollmodellen immer nur ein Medium. Reale Netzwerke nutzen jedoch häufig mehr als ein Medium (Kabel, Funkwellen) zur Nachrichtenübertragung. Als Lösungsansatz wird das Übertragungsmedium vom Simulationskontext entkoppelt. So kann es mehrere verschiedene Übertragungsmedien im gleichen Simulationsprogramm geben, und es können komplexere Netzwerkstrukturen nachgebildet werden. Jedes Medium besitzt dabei seine eigene Topologie.

Die Übertragung der Nachrichten via Medium wurde bislang nur durch ein einziges Simulationseignis für die Nachrichtenankunft modelliert. Das erschwerte jedoch die Modellierung von Kollisionen, welche von den Netzwerkschnittstellen festgestellt werden müssen. Als Lösungsansatz bietet sich hier die Verwendung von zwei Simulationseignissen an, welche den Netzwerkschnittstellen Beginn und Ende einer Übertragung anzeigen. Auf dieser Basis kann in der Klasse `Device` ein Mechanismus zur Erkennung von Kollisionen integriert werden.

Protokolldateneinheiten

In Version 2.2 von ODEMX war die Schnittstelle `ProtocolMessage` für Nachrichtentypen mit ihren acht rein virtuellen Methoden schon sehr spezifisch und grenzte dadurch die Modellierungsfreiheit ein. Ein Problem war auch die unrealistische Abfrage der Übertragungsdauer an der Nachricht selbst. Dieser Wert hängt schließlich maßgeblich von der jeweiligen Netzwerkschnittstelle und ihrer zum Senden verfügbaren Bandbreite ab. Als Lösungsansatz wird die vorgegebene Schnittstelle verkleinert auf zwei Methoden. Ermöglicht wird dies durch das Zusammenspiel der Klassen `Medium` und `Device`. Anstelle der Übertragungszeit wird nun mit `getSize` die Größe einer Nachricht abgefragt, so dass jede Netzwerkschnittstelle selbst die Übertragungsdauer in Abhängigkeit ihrer Konfiguration berechnen kann. Zudem wird von den PDU-Klassen wie bisher die Implementation der Methode `clone` verlangt, welche komplette Kopien von Nachrichten erstellt, um die Seiteneffektfreiheit zu sichern. Da sich die Schnittstelle auf Protokolldateneinheiten bezieht, wird der Name in `Pdu` abgeändert.

4.3.2 Konzept für die Erweiterung der Implementation

Im letzten Abschnitt wurde bereits erwähnt, dass das erweiterte Protokollsimulationsmodul die Modellbildung mit unterschiedlichem Abstraktionsgrad durch einige neue Klassen unterstützt. Dabei kann der Anwender die Verhaltensbeschreibung der einzelnen Komponenten nach wie vor beliebig komplex gestalten. Die Unterscheidung auf Klassenebene bestimmt lediglich, welche Komponenten des Protokollsimulationsmoduls miteinander interagieren sollen. Die mit dieser Arbeit vorliegende Implementation ermöglicht die Modellbildung mit drei verschiedenen Detailgraden.

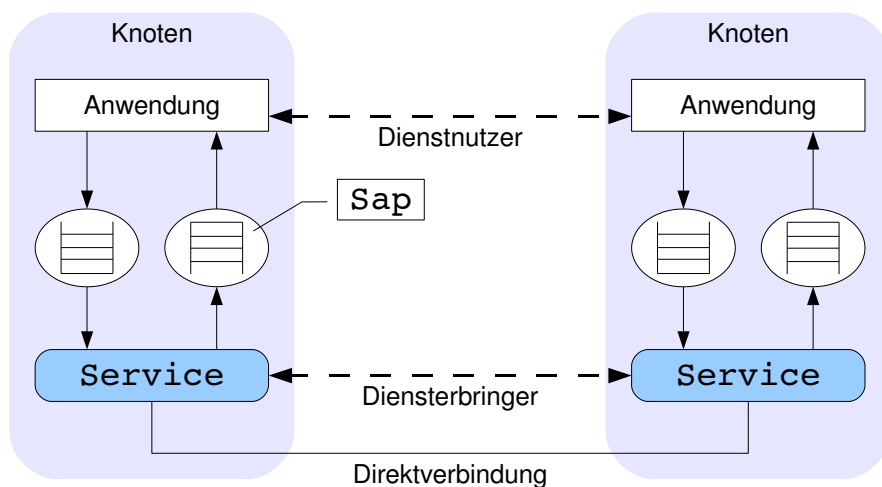


Abbildung 4.4: Modellierung von Netzwerkknoten unter Abstraktion von Protokollstacks und Topologie durch Nutzung der Klasse Service

Analog zu dem in Abbildung 4.1 gezeigten Modell eines Kommunikationsdienstes befasst sich der erste Detailgrad lediglich mit dem Dienst und dessen Bereitstellung durch einen Dienstbringer. Von der Struktur des Netzwerks wird komplett abstrahiert, indem alle Knoten direkt miteinander verbunden werden. Das Konzept ist in Abbildung 4.4 dargestellt. Ein Service-Objekt wird über ein pufferndes Sap-Objekt angesprochen, bearbeitet die übergebenen Daten und leitet sie weiter an den Empfänger-Service, der seinerseits die erhaltenen Daten bearbeitet und über einen Sap an den Dienstanwender weiterreicht. Wie der Bearbeitungsschritt aussieht, ist durch den Anwender zu definieren.

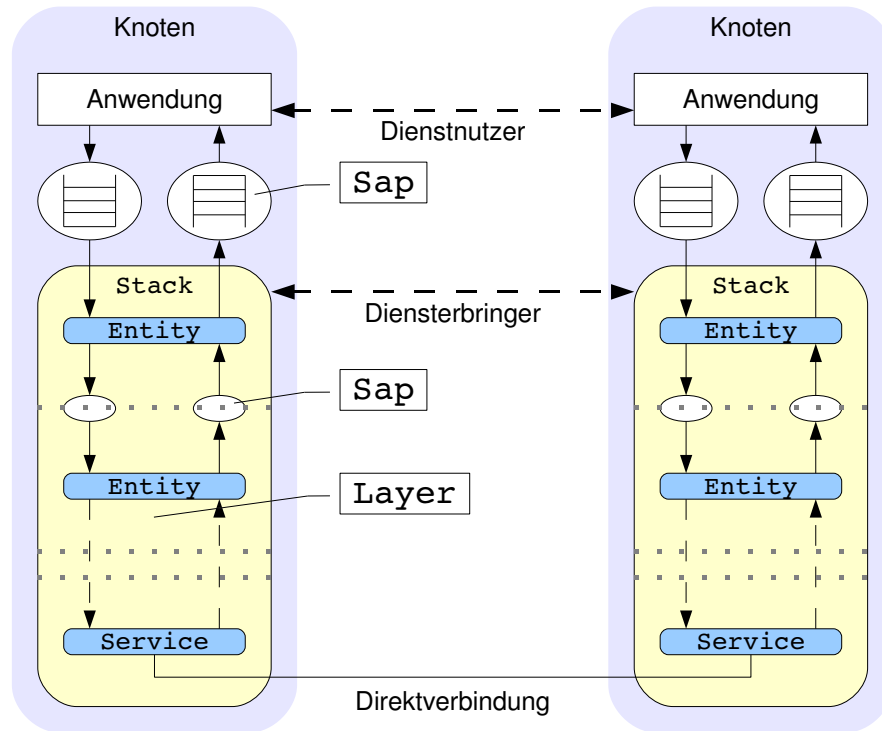


Abbildung 4.5: Vereinfachte Darstellung eines Protokollstacks mit Schichten und Instanzen, der von der Netzwerkstruktur abstrahiert

Der zweite Detailgrad ergänzt einen Stack mit Schichten und Protokollinstanzen, welche durch die Klassen Layer beziehungsweise Entity repräsentiert werden. Abbildung 4.5 illustriert die Idee, macht jedoch aus Gründen der Übersichtlichkeit ein paar Vereinfachungen. Tatsächlich können Protokollinstanzen beliebig viele SAPs bedienen, und jede Schicht darf mehr als eine Protokollinstanz enthalten, die jeweils ihre eigenen SAPs anbietet. Bei diesem Detailgrad fungiert nun der Stack als Diensterbringer für die Anwendungsschicht. Aber auch die Entity-Klassen und die Service-Klasse sind Diensterbringer – ihre Dienstnutzer sind jeweils die Diensterbringer der darüberliegenden Schicht. Außerdem ist in der Abbildung nur angedeutet, dass der Stack beliebig viele Schichten enthalten darf und dass die Kommunikation zwischen Schichten grundsätzlich über SAPs verläuft.

Bei diesem mittleren Detailgrad wird weiterhin von der Netzwerkstruktur abstrahiert, indem die Funktionalität der untersten Schicht durch eine Service-Klasse erbracht wird. Hier zeigt sich eine Besonderheit der Basisklasse Service, da diese einerseits als Ersatz für einen Stack verwendet werden kann, andererseits aber

auch mit der Klasse Layer kompatibel ist. So kann sie auch in der untersten Schicht des Stacks zum Einsatz kommen. Für den Dienstanutzer auf oberster Ebene, also die Anwendung, ändert sich beim Austausch des Service-Objekts durch ein Stack-Objekt nichts. Er greift wie zuvor auf SAPs zu, welche ihm nun von der obersten Schicht des Protokollstacks angeboten werden.

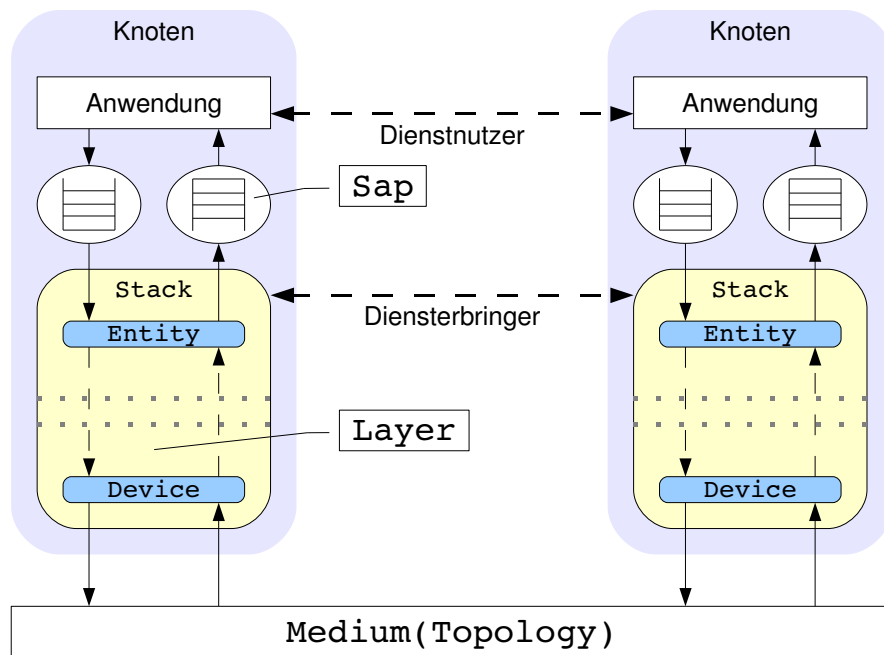


Abbildung 4.6: Vereinfachter Protokollstack mit Schichten und Instanzen, der eine Netzwerkschnittstelle (Device) und ein Übertragungsmedium zur Kommunikation mit anderen Knoten verwendet

Abbildung 4.6 zeigt den dritten und höchsten Detailgrad, bei dem alle vom Protokollsimulationsmodul angebotenen Klassen zum Einsatz kommen. Auch in dieser Abbildung wurden die oben genannten Vereinfachungen vorgenommen, um die Übersichtlichkeit zu gewährleisten. Der wesentliche Unterschied zum mittleren Detailgrad liegt nun in der Ersetzung der Service-Klasse durch eine Device-Klasse. Mit dieser Struktur kann eine beliebige Netzwerktopologie modelliert werden, indem dem Medium die Links zwischen den Netzwerkschnittstellen der Knoten bekannt gemacht werden.

Objekte von Entity-Spezialisierungen sind in dieser neuen Version des Protokollsimulationsmoduls allerdings nicht mehr für die Verwendung in der untersten

Schicht des Protokollstacks vorgesehen, denn im Gegensatz zur Klasse `Service` ist in `Entity` kein Mechanismus zur Direktverbindung von Netzwerkknoten integriert. Die Klasse `Medium` kann wiederum nur mit `Device`-Objekten interagieren, weshalb die Dienstbringer der untersten Schicht des Stacks je nach Abstraktionsgrad `Service`- oder `Device`-Spezialisierungen sein müssen.

Die beschriebenen Detailgrade sind insbesondere für die inkrementelle Entwicklung von Protokollmodellen geeignet, da zunächst die Funktionalität der Anwendungsschicht in Verbindung mit einer einfachen `Service`-Implementation getestet werden kann. Anschließend kann Schicht um Schicht der Protokollstack modelliert werden, bis im finalen Schritt noch die Netzwerkschnittstellen und das `Medium` hinzukommen und schließlich Simulationsexperimente mit realistischen Topologien durchgeführt werden können.

4.4 Softwaretechnische Umsetzung des erweiterten Protokollsimulationsmoduls

Das Thema dieses Abschnitts ist die Implementation des im letzten Abschnitt konzeptuell vorgestellten erweiterten Protokollsimulationsmoduls. Alle nachfolgend beschriebenen Klassen befinden sich, sofern nicht anders gekennzeichnet, im Namensraum `odemx::protocol`. Aus Gründen der Lesbarkeit wird dieses Präfix bei der Verwendung von Klassennamen im Text weggelassen.

Die Ausführungen zur Implementation der Komponenten werden von einem einfachen Beispiel begleitet, welches die Umsetzung eines simplen Kommunikationsdienstes für Netzwerkknoten anhand der Konzepte des vorangegangenen Abschnitts illustriert. Dabei werden etappenweise die Detailgrade der Modellierung sowie die Verwendung der bereitgestellten Klassen und Konzepte demonstriert.

4.4.1 Protokolldateneinheiten

Damit das Protokollsimulationsmodul mit beliebigen Nachrichtentypen arbeiten kann, ist es notwendig, eine Schnittstelle für Protokolldateneinheiten vorzugeben. Zu diesem Zweck wird die abstrakte Klasse `Pdu` bereitgestellt, welche die Implementation der Methoden `getSize` und `clone` verlangt. Erstere muss die Größe der

Dateneinheit angeben, welche für die korrekte Berechnung der Übertragungsdauer benötigt wird, und letztere muss eine Kopie des Objekts erzeugen, die ohne Seiteneffekte von mehreren Empfänger-knoten bearbeitet werden kann. Diese Funktionalität wird vom Übertragungsmedium verwendet, um die gleiche Dateneinheit an mehrere Empfänger zu übermitteln. Der Zugriff auf abgeleitete Nachrichtentypen erfolgt mit Hilfe polymorpher Zeiger. Da die manuelle Speicherverwaltung dieser Objekte in Protokollsimulationen sehr komplex werden kann, kommt auch an dieser Stelle das Klassentemplate `std::tr1::shared_ptr` zum Einsatz. Alle Klassen des Protokollmoduls arbeiten mit `shared_ptr`-verwalteten Pdu-Objekten, indem sie die Typdefinition `PduPtr` verwenden.

4.4.2 Dienstzugangspunkte und Diensterbringer

Der Zugriff auf Dienste ist im Protokollsimulationsmodul ausschließlich über Dienstzugangspunkte realisiert. Ihre Repräsentation finden diese in der Klasse `Sap`. Dienstanutzer und Diensterbringer verwenden derartige Objekte als Schnittstelle zum Nachrichtenaustausch. Der Zugriff auf SAPs von Diensterbringern erfolgt über einen Bezeichner, der als Konstruktorparameter anzugeben ist und mittels `getName` abgefragt werden kann. Die Methoden `read` und `write` bieten lesenden und schreibenden Zugriff auf einen internen Datenpuffer, welcher die Basis für die asynchrone Kommunikation zwischen Dienstanutzer und Diensterbringer ist. Der interne Puffer ist durch das Klassentemplate `odemx::sync::PortT` implementiert, weil dieses mit dem Memory-basierten Wartekonzept zur Prozesssynchronisation [Kluth07] kompatibel ist. Für den Aufruf mit `odemx::base::Process::wait` liefert die Methode `getBuffer` einen Zeiger auf den Puffer. Dies ist immer dann nötig, wenn ein Prozess auf eines von mehreren Ereignissen warten soll und der Dateneingang an einem SAP dazugehört.

Diensterbringer werden durch die abstrakte Basisklasse `ServiceProvider` repräsentiert. Schon in Abschnitt 4.1.2 wurde erwähnt, dass die Diensterbringer die aktiven Komponenten in dem hier verwendeten Protokollmodell sind. Aus diesem Grund ist die Klasse `ServiceProvider` von `odemx::base::Process` abgeleitet. Ihr Prozessverhalten ist definiert als andauernder Wartezustand, der immer dann unterbrochen wird, wenn an einem der SAPs eine Nachricht eintrifft. Dadurch wird der Prozess sofort aktiviert und ruft die rein virtuelle Methode `handleInput`, die in Spezialisierungen zu überschreiben ist.

Wie in Abschnitt 4.1.2 erwähnt, sind SAPs immer genau einem Dienstbringer zuzuordnen. Daher werden die Sap-Objekte von dieser Basisklasse verwaltet. Erzeugt werden kann ein Sap mit der Methode `addSap`, welche die Angabe des Namens verlangt. Bei Dopplungen wird eine Warnung erzeugt und das bereits existierende Objekt zurückgegeben, da die SAPs vom Dienstbringer eindeutig identifizierbar sein müssen. Mit `getSap` und `getSaps` kann auf einen beziehungsweise alle registrierten SAPs zugegriffen werden. Durch `removeSap` ist auch das Entfernen von SAPs möglich. Die Klasse `ServiceProvider` stellt ausserdem einen Zeiger auf die umgebende Schicht bereit, der mit `setLayer` und `getLayer` gesetzt oder gelesen werden kann.

Das Protokollsimulationsmodul bietet mit `Service`, `Entity` und `Device` verschiedene Spezialisierungen der Klasse `ServiceProvider` an, die unterschiedliche Aufgabenbereiche abdecken. Alle drei folgen aber einem einheitlichen Namensschema für Methoden, die das Weiterleiten von Nachrichten implementieren. Die Methode `send` repräsentiert dabei das Versenden von Nachrichten an ein Partner-Objekt derselben Schicht. Implementiert ist dies durch Weitergabe von Nachrichten an die nächsttiefere Schicht oder das Übertragungsmedium. Die Methode `pass` reicht dagegen Nachrichten an die nächsthöhere Schicht weiter. Die Signaturen der Methoden unterscheiden sich jedoch in den einzelnen Spezialisierungen, weshalb hier keine virtuellen Methoden vorgegeben sind.

4.4.3 Kommunikationsdienste und Fehlermodellierung

Die Klasse `Service` ist eine Spezialisierung von `ServiceProvider`, welche die Abstraktion von Modellierungsdetails wie Protokollstack und Netzwerktopologie ermöglicht. Ziel ist es, durch die Verwendung einer einfachen Implementation eines Kommunikationsdienstes schnell zu einem lauffähigen Modell zu kommen, an dem erste Experimente bezüglich der Funktionalität der Anwendungsschicht durchführbar sind. Wenn es in einem Modell nur darum geht, dass Anwendungen, die auf verschiedenen Knoten laufen, miteinander kommunizieren können, reicht diese Vereinfachung mit direkter Verbindung der Knoten völlig aus.

Für den Empfang von Dateneinheiten aus dem Netzwerk besitzt `Service` einen internen Netzwerk-SAP, dem PDUs durch Aufruf der internen Methode `receive` übergeben werden. Die Methode `ServiceProvider::handleInput` wird von `Service` daher so implementiert, dass eine Unterscheidung zwischen Senden und

Empfangen stattfindet. Falls eine Dateneinheit vom Netzwerk empfangen wird, erfolgt der Aufruf der rein virtuellen Methode `handleReceive`. Anderenfalls ist eine PDU der darüberliegenden Schicht zu verarbeiten, weshalb dann die rein virtuelle Methode `handleSend` aufgerufen wird. Beide Methoden sind vom Anwender in einer Spezialisierung der Klasse zu implementieren. Typischerweise finden dabei die Methoden `send` und `pass` ihre Verwendung. So wird in `handleSend` die Methode `send` genutzt, um PDUs an einen Partnerknoten im Netzwerk zu versenden, während in `handleReceive` empfangene PDUs mittels `pass` an die darüberliegende Schicht weitergeleitet werden. Wird die Klasse `Service` dem einfachsten Detailgrad entsprechend direkt ohne Protokollstack verwendet, so können mit der Methode `addReceiveSap` Ausgabe-SAPs für die Anwendungsschicht hinzugefügt werden, von denen die erhaltenen Daten gelesen werden können.

Von der Modellierung des Netzwerks wird abstrahiert, indem alle registrierten `Service`-Objekte anhand einer Adresse im String-Format ansprechbar sind. So können alle Knoten eines Netzwerks direkt miteinander kommunizieren. Implementiert ist diese Direktverbindung durch eine statische `std::map`, welche die Adressen mit Zeigern auf `Service`-Objekte assoziiert. Die Manipulation dieser Adresstabelle erfolgt mit den statischen Methoden `registerService` und `removeService`. Lesender Zugriff wird durch die statische Methode `getAddressMap` gestattet. Zeiger auf einzelne `Service`-Objekte können mittels `getPeer` unter Angabe der Adresse angefordert werden.

Einem `Service` kann durch die statische Methode `setErrorModel` ein Fehlermodell zugeordnet werden, welches in der Methode `receive` direkt vor dem Nachrichtenempfang abgefragt wird. Die Fehlermodellierung wird wie oben erwähnt durch die Schnittstelle `ErrorModel` unterstützt. Diese verlangt lediglich die Implementation der rein virtuellen Methode `apply`, der als Parameter ein PDU-Zeiger übergeben wird. Dadurch kann das Fehlermodell auch leicht den Inhalt der Dateneinheit verändern. Soll die PDU anschließend weitergeleitet werden, so muss diese Methode `true` zurückgeben. Um bibliotheksseitig mindestens die gleiche Funktionalität zur Fehlermodellierung wie bisher anzubieten, wurde exemplarisch die Klasse `ErrorModelDraw` implementiert, welche durch Abfrage eines `odemx::random::Draw`-Objekts `true` oder `false` zurückgibt und damit zufallssteuert bestimmt, ob Nachrichten ankommen oder nicht.

4.4.4 Beispiel XCS 1 - Ein Einfacher Kommunikationsdienst

Im Rahmen dieses begleitenden Beispiels wird der simple Kommunikationsdienst XCS (*example communication service*) umgesetzt, welcher den Nachrichtenaustausch zwischen Netzwerkknoten unter Verwendung des PDU-Typs StringData simuliert. Um das Beispiel kurz zu halten findet auf der Anwendungsebene neben dem Logging keine weitere Datenverarbeitung statt. Außerdem gibt es lediglich zwei Knoten, wobei der Sender eine Nachricht im String-Format an den Empfänger schickt.

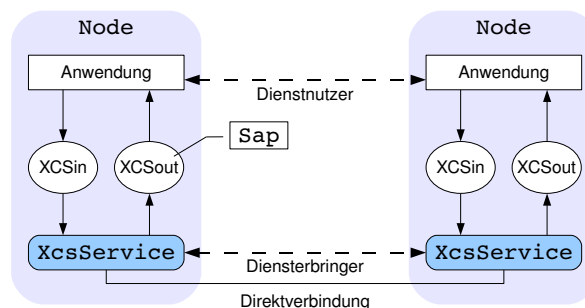


Abbildung 4.7: Knotenmodell des Beispiels auf dem niedrigsten Detailgrad: eine Service-Spezialisierung erbringt den Dienst XCS

Abbildung 4.7 zeigt das Modell der verwendeten Knotenstruktur mit den entsprechenden Klassennamen. Diese erste Version des Dienstes XCS ist mit dem niedrigsten Detailgrad modelliert. Es gibt dementsprechend nur eine Service-Klasse, welche die Übertragung der Nachrichten umsetzt. Die Kommunikation zwischen Anwendungsschicht und Service erfolgt durch Dateneinheiten des Typs StringData.

```

1 #include <odemx/odemx.h>
2 using namespace odemx;
3 using namespace odemx::protocol;
4
5 typedef Service::AddressType Addr;
6 typedef std::string SapName;
7
8 struct StringData: Pdu {
9     typedef std::tr1::shared_ptr< StringData > Ptr;
10     Addr src, dest;
11     SapName destSap;
12     std::string data;

```

```
13
14     StringData( const Addr& src, const Addr& dest, const SapName& dSap,
15                 const std::string& data )
16     :   src( src ), dest( dest ), destSap( dSap ), data( data )
17     {}
18     virtual PduPtr clone() const {
19         return PduPtr( new StringData( src, dest, destSap , data ) );
20     }
21     virtual std::size_t getSize() const {
22         return src.size() + dest.size() + destSap.size() + data.size() ;
23     }
24 };
```

Quelle 4.1: Pdu-Klasse zum Transport von String-Daten

In Quelle 4.1 wird die PDU-Klasse `StringData` definiert, welche neben dem eigentlichen Nachrichten-String noch die Adressen von Sender und Empfänger sowie den Ausgabe-SAP des Empfängers enthält. Da die Klasse die Schnittstelle `Pdu` implementiert, müssen die Methoden `clone` und `getSize` definiert werden. Der Aufruf von `clone` gibt ein neu erzeugtes, `shared_ptr`-verwaltetes `StringData`-Objekt zurück. Die Methode `getSize` berechnet die Größe der Nachricht, indem die Längen der transportierten Strings aufsummiert werden.

```
25 using std::tr1::static_pointer_cast;
26
27 class XcsService: public Service {
28 public:
29     typedef StringData PduType;
30
31     XcsService( base::Simulation& sim, const data::Label& label )
32     :   Service( sim, label ) {
33         addSap( "XCSin" );
34     }
35     virtual void handleSend( const std::string& sap, PduPtr p ) {
36         info << log( "send" );
37         holdFor( p->getSize() );
38         send( static_pointer_cast< PduType >( p )->dest, p );
39     }
40     virtual void handleReceive( PduPtr p ) {
41         info << log( "pass" );
42         pass( static_pointer_cast< PduType >( p )->destSap, p );
43     }
44 };
```

Quelle 4.2: Service-Klasse für die Kommunikation zwischen Netzwerkknoten

Die Dienst-Implementation ist in Quelle 4.2 dargestellt. Die Klasse `XcsService` erbt von der abstrakten Klasse `Service` und definiert als erstes den verwende-

ten PDU-Typ. Bei Kontruktion wird der SAP XCSin hinzugefügt, über den der Dienst Eingaben entgegennimmt. Die Dienst-Funktionalität wird durch die Implementation der rein virtuellen Methoden `handleSend` und `handleReceive` erbracht. Der Aufruf von `handleSend` erfolgt automatisch, sobald an einem Eingabe-SAP Daten vom Dienstnutzer bereitgestellt werden - in diesem Fall also am SAP XCSin. Der Zeitverbrauch beim Versenden von Nachrichten ist durch die Warteanweisung `holdFor` modelliert. Nach dieser Zeit wird die Nachricht mittels `send` an den in der PDU gespeicherten Empfänger weitergeleitet. Die Methode `handleReceive` wird aufgerufen, wenn dem internen Netzwerk-SAP Daten eines Kommunikationspartners übergeben werden. Hier werden die Daten sofort an den in der Nachricht angegebenen SAP weitergereicht. Man beachte, dass Service-Spezialisierungen Prozesse sind und daher auch die Funktionalität der Basisklasse `odemx::data::Producer` nutzen können. So wird im Beispiel der Log-Kanal `info` für die Anzeige der Methodenaufrufe genutzt. Eine weitere Besonderheit ist die Verwendung des Funktionstemplates `static_pointer_cast`, welches zur expliziten Typumwandlung von `shared_ptr`-Objekten verwendet wird.

`std::tr1::static_pointer_cast`

Die zur Typumwandlung bereitgestellten Cast-Operatoren von C++ sind nur auf Zeiger und Referenzen anwendbar. Ein `shared_ptr` ist dagegen ein Objekt. Um dennoch eine Typumwandlung vorzunehmen, muss der intern gekapselte Zeiger per Cast-Operator konvertiert und ein neues `shared_ptr`-Objekt des Ziel-typs angelegt werden. Das Funktionstemplate `static_pointer_cast` übernimmt diese Aufgabe und gibt einen `shared_ptr` zurück, der den gleichen Referen-zenzähler verwendet, wie das Quellobjekt. Die Typumwandlung entspricht dabei `shared_ptr<Derived>(static_cast<Derived*>(sp_base.get()))`.

```
45 class Node: public base::Process {
46 public:
47     Node( const data::Label& label, const Addr& addr, const Addr& recAddr = "" )
48         : Process( label )
49         , service_( getSimulation(), label + " service" )
50         , address_( addr )
51         , receiver_( recAddr )
52         , isSender_( ! recAddr.empty() )
53     {
54         service_.addReceiveSap( "XCSout" );
55         XcsService::registerService( addr, service_ );
56     }
```

```
57     virtual int main() {
58         if( isSender_ ) {
59             Sap* sap = service_.getSap( "XCSin" );
60             StringData::Ptr pdu( new StringData( address_, receiver_, "XCSout", "message" ) );
61             info << log( "sending PDU via XCSin" )
62                 .detail( "to", pdu->dest )
63                 .detail( "data", pdu->data );
64             sap->write( pdu );
65         } else {
66             Sap* sap = service_.getReceiveSap( "XCSout" );
67             while( true ) {
68                 StringData::Ptr pdu = static_pointer_cast< StringData >( sap->read() );
69                 info << log( "received PDU via XCSout" )
70                     .detail( "from", pdu->src )
71                     .detail( "data", pdu->data );
72             }
73         }
74         return 0;
75     }
76 private:
77     XcsService service_;
78     Addr address_, receiver_;
79     bool isSender_;
80 };
```

Quelle 4.3: Prozessklasse zur Modellierung von Netzwerkknoten

Netzwerkknoten werden in diesem Beispiel durch die in Quelle 4.3 gezeigte Prozessklasse `Node` modelliert. Ihre Objekte können als Sender oder Empfänger konfiguriert werden. Dem Konstruktor wird die Adresse übergeben, unter der das `XcsService`-Objekt des Knotens registriert wird. Weiterhin wird der Ausgabe-SAP `XCSout` zu dem Diensterbringer hinzugefügt, damit auf Nachrichten aus dem Netzwerk gewartet werden kann. Wird als Konstruktorparameter eine Empfänger-Adresse angegeben, so agiert das `Node`-Objekt als Sender. Ansonsten übernimmt es die Rolle des Empfängers.

Das Prozessverhalten ist in der `main`-Methode der Klasse `Node` definiert. Als Sender erzeugt der Knoten eine neue `StringData`-Nachricht und übergibt diese an den SAP `XCSin`. Als Empfänger interagiert der Knoten dagegen mit dem SAP `XCSout` und wartet auf ankommende Nachrichten. Der Aufruf von `Sap::read` blockiert den Prozess dabei solange, bis eine Nachricht eintrifft. Sowohl das Senden als auch der Empfang werden über den Log-Kanal `info` aufgezeichnet.

```
81 int main() {  
82     base::Simulation& sim = getDefaultSimulation();  
83     sim.addConsumer( data::channel_id::info,  
84                     data::output::OStreamWriter::create( std::cout ) );  
85  
86     Node sender( "Sender", "Addr0", "Addr1" );  
87     Node receiver( "Receiver", "Addr1" );  
88     sender.activate();  
89     receiver.activate();  
90  
91     sim.run();  
92 }
```

Quelle 4.4: Die main-Funktion des Simulationsprogramms

Die main-Funktion des Simulationsprogramms ist in Quelle 4.4 dargestellt. An dieser Stelle wird demonstriert, dass sich nun auch der Default-Simulationskontext für die Protokollsimulation nutzen lässt. Am Log-Kanal info, der in den Klassen Node und XcsService Verwendung findet, wird ein OStreamWriter registriert, der die Log-Einträge auf der Standardausgabe anzeigt. Danach werden ein Sender- und ein Empfängerknoten erzeugt und aktiviert. Nach dem Start der Simulation durch Simulation::run verschickt der Sender eine Nachricht an den Knoten mit Adresse Addr1. Dessen XcsService empfängt die Nachricht und leitet sie an den Ausgabe-SAP XCSout weiter, so dass der Knoten die Nachricht verarbeiten kann. Die Logging-Ausgaben des Programms sind in Quelle 4.5 zu sehen.

```
1 ODEmX Log  
2 =====  
3 0 info: Sender(Node) sending PDU via XCSin [to=Addr1 | data=message]  
4 0 info: Sender service(XcsService) send  
5 23 info: Receiver service(XcsService) pass  
6 23 info: Receiver(Node) received PDU via XCSout [from=Addr0 | data=message]
```

Quelle 4.5: Ausgabe des ersten Beispielprogramms

4.4.5 Protokollstacks mit Schichten und Instanzen

Die Klasse Entity ist eine weitere Spezialisierung von ServiceProvider, welche als Basisklasse beliebiger Protokollinstanztypen vorgesehen ist. Von dieser Klasse werden ebenfalls die beiden Methoden send und pass bereitgestellt, welche die Weitergabe von PDUs an SAPs der unteren beziehungsweise der oberen Schicht

implementieren. Sollte der angeforderte SAP in der Schicht nicht existieren, so wird eine Warnung ausgegeben. Die rein virtuelle Methode `handleInput` muss in Spezialisierungen dieser Klasse definiert werden.

Schichten werden durch die Klasse `Layer` modelliert, welche die Verwaltung der `ServiceProvider`-Spezialisierungen übernimmt. Neue Dienstbringer können mit der Methode `addServiceProvider` registriert werden. Bei Zerstörung der Schicht werden auch alle registrierten Dienstbringer freigegeben. Die SAPs der einzelnen Dienstbringer werden mit `addSap` registriert, so dass Protokollinstanzen diese mit den Methoden `getSap` und `getSaps` abfragen können. Das Entfernen von SAPs erledigt die Methode `removeSap`. Weiterhin besitzt die Klasse Zeiger auf die obere und die untere Schicht, die beim Hinzufügen von Schichten am Protokollstack gesetzt werden. Protokollinstanzen können dann über die Methoden `getLowerLayer` und `getUpperLayer` auf die angrenzenden Schichten zugreifen und mit deren SAPs interagieren.

Der Protokollstack wird durch die Klasse `Stack` repräsentiert, die wiederum die Verwaltung der mittels `addLayer` registrierten Schichten übernimmt. Im Destruktor wird der Speicher der `Layer`-Objekte wieder freigegeben. Auch die Interaktion mit dem Protokollstack verläuft über SAPs. In diesem Fall wird via `getSap` ein SAP der obersten Schicht angefordert, über den die Nachrichten in den Stack gelangen. Die Verwendung erfolgt analog zu `Service`, womit die beiden Klassen austauschbar sind. Für den Empfang der Nachrichten vom Stack müssen ebenfalls mit der Methode `addReceiveSap` Ausgabe-SAPs hinzugefügt werden, an welche die Instanzen der obersten Schicht dann die empfangenen Daten weitergeben können.

4.4.6 Beispiel XCS 2 - Zweischichtiger Kommunikationsdienst

Dieser zweite Teil des Beispiels befasst sich mit der Umsetzung des Kommunikationsdienstes XCS auf dem mittleren Detailgrad. Die Anwendungsschicht der Klasse `Node` bleibt im Wesentlichen unverändert. Nur der XCS-Dienstbringer wird ersetzt, indem anstelle eines `XcsService`-Objekts nun ein `Stack`-Objekt verwendet wird.

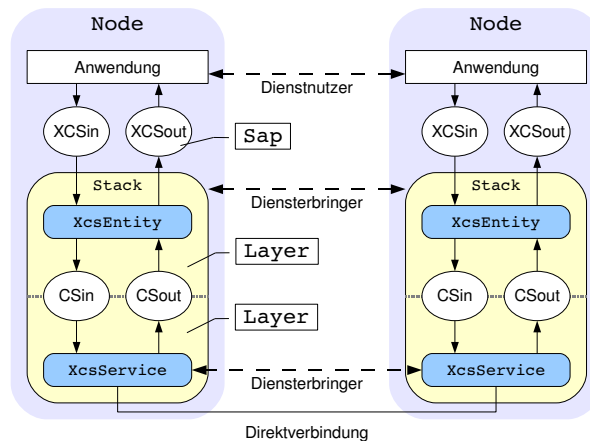


Abbildung 4.8: Knotenmodell des Beispiels auf dem mittleren Detailgrad: der Dienst XCS wird durch einen zweischichtigen Protokollstack erbracht

Abbildung 4.8 zeigt die Knotenstruktur. Der dargestellte Protokollstack ist in zwei Schichten untergliedert. Als Dienstbringer in der oberen Schicht kommt eine Entity-Spezialisierung zum Einsatz, und die untere Schicht wird durch eine Abwandlung der Klasse XcsService modelliert.

```

1 struct TransmissionData: Pdu {
2     Addr& src;
3     Addr& dest;
4     StringData::Ptr payload;
5
6     TransmissionData( StringData::Ptr data )
7     :   src( data->src ), dest( data->dest ), payload( data )
8     {}
9     virtual PduPtr clone() const {
10         return PduPtr( new TransmissionData( payload ) );
11     }
12     virtual std::size_t getSize() const {
13         return payload->getSize();
14     }
15 };
16
17 class XcsService: public Service {
18 public:
19     typedef TransmissionData PduType;
20     XcsService( base::Simulation& sim, const data::Label& label )
21     :   Service( sim, label ) {
22         addSap( "CSin" );
23     }
24     ...

```

```
25     virtual void handleReceive( PduPtr p ) {  
26         ...  
27         pass( "CSout", p );  
28     }  
29 };
```

Quelle 4.6: Die PDU-Klasse `TransmissionData` und Änderungen an der Klasse `XcsService` zur Nutzung in der unteren Schicht des Protokollstacks

Quelle 4.6 zeigt zunächst den PDU-Typ `TransmissionData`, der zur Kommunikation zwischen den `XcsService`-Objekten in der unteren Schicht verwendet wird. Dieser nutzt bei Konstruktion die schon in `StringData` vorhandenen Informationen über Sender und Empfänger und transportiert die unveränderte `StringData`-PDU als Nutzlast. Weiterhin sind die Änderungen an der Klasse `XcsService` dargestellt. Als PDU-Typ wird `TransmissionData` festgelegt, und die SAPs werden umbenannt zu `CSin` und `CSout`, da `XCSin` und `XCSout` auf der obersten Schicht des Stacks verwendet werden.

```
1  class XcsEntity: public Entity {  
2  public:  
3      XcsEntity( base::Simulation& sim, const data::Label& label )  
4      :   Entity( sim, label ) {  
5          addSap( "XCSin" );  
6          addSap( "CSout" );  
7      }  
8      virtual void handleInput( const std::string& sapName, PduPtr p ) {  
9          if( sapName == "XCSin" ) {  
10             info << log( "send" );  
11             StringData::Ptr pdu = static_pointer_cast< StringData >( p );  
12             send( "CSin", PduPtr( new TransmissionData( pdu ) ) );  
13         } else if( sapName == "CSout" ) {  
14             info << log( "pass" );  
15             StringData::Ptr pdu = static_pointer_cast< TransmissionData >( p )->payload;  
16             pass( pdu->destSap, pdu );  
17         }  
18     }  
19 };
```

Quelle 4.7: Die Entity-Spezialisierung der oberen Schicht

Quelle 4.7 zeigt die Klasse `XcsEntity`, welche die Protokollinstanzen der oberen Schicht beschreibt. Bei Konstruktion werden zunächst zwei SAPs erzeugt. Der SAP `XCSin` wird wie im Beispielabschnitt XCS 1 für die Übergabe von Daten der Anwendungsschicht an den Kommunikationsservice verwendet. Der SAP `CSout` dient dagegen dem Empfang von PDUs der unteren Schicht. Die Methode `handleInput`

beschreibt das Verhalten der Protokollinstanz. Bei Eingaben von der Anwendungsschicht wird die erhaltene StringData-PDU in ein TransmissionData-Objekt verpackt, welches über den SAP CSin des XcsService verschickt wird. Aus den TransmissionData-Objekten der unteren Schicht wird wiederum die StringData-PDU extrahiert und an den darin enthaltenen Ausgabe-SAP des Protokollstacks weitergegeben.

Für die Knotenklasse Node ist die Nutzung von Stack oder Service transparent, da die Kommunikation nur durch Interaktion mit den SAPs XCSin und XCSout erfolgt. Die Änderung am Knoten betrifft also lediglich den Austausch des XcsService-Objekts durch einen entsprechend konfigurierten Stack. Dieser Stack wird per `std::auto_ptr` verwaltet und durch die in Quelle 4.8 abgebildete Funktion erzeugt.

```
1  std::auto_ptr< Stack > makeStack( base::Simulation& sim,
2      const data::Label& label, const Addr& address )
3  {
4      Layer* entityLayer = new Layer( sim, label + " entity layer" );
5      entityLayer->addServiceProvider( new XcsEntity( sim, label + " entity" ) );
6      Layer* serviceLayer = new Layer( sim, label + " service layer" );
7      Service* service = new XcsService( sim, label + " service" );
8      serviceLayer->addServiceProvider( service );
9      XcsService::registerService( address, *service );
10
11     std::auto_ptr< Stack > stack( new Stack( sim, label ) );
12     stack->addLayer( entityLayer );
13     stack->addLayer( serviceLayer );
14     return stack;
15 }
```

Quelle 4.8: Funktion zur Erzeugung eines Protokollstacks

In `makeStack` werden zunächst die beiden Schichten erzeugt und die entsprechenden Dienstbringer `XcsEntity` und `XcsService` hinzugefügt. Wie zuvor muss das `XcsService`-Objekt anhand seiner Adresse registriert werden, damit die Direktverbindung zwischen den Knoten hergestellt werden kann. Danach werden die Schichten zu einem Stack hinzugefügt. An dieser Stelle muss die Reihenfolge der Schichten von oben nach unten beachtet werden, weil die Zeiger der `Layer`-Objekte auf die obere und untere Schicht in diesem Schritt initialisiert werden. Die SAP-Zugriffe der Klasse `Node` erfolgen an dem Stack-Objekt genauso, wie es vorher beim `XcsService` der Fall war.

Auch die `main`-Funktion des so erweiterten Beispiels bleibt unverändert. Bei der Ausführung der Simulation verschickt der Sender über den SAP `XCSin` seine Nachricht, während der Empfänger am SAP `XCSout` auf Nachrichtenankunft wartet. Die `StringData`-Dateneinheit passiert zunächst die `XcsEntity` des Senders, wo sie als `TransmmissionData` verpackt und an den `XcsService` weitergegeben wird, der den Transport zum Empfänger-Knoten umsetzt. Der `XcsService` des Empfängers reicht die Nachricht an die `XcsEntity` weiter, welche die Daten extrahiert und an den SAP `XCSout` übergibt. Die Ausgaben des Programms sind in Quelle 4.9 dargestellt.

```
1 ODEmX Log
2 =====
3 0 info: Sender(Node) sending PDU via XCSin [to=Addr1 | data=message]
4 0 info: Sender entity(XcsEntity) send
5 0 info: Sender service(XcsService) send
6 23 info: Receiver service(XcsService) pass
7 23 info: Receiver entity(XcsEntity) pass
8 23 info: Receiver(Node) received PDU via XCSout [from=Addr0 | data=message]
```

Quelle 4.9: Ausgabe des zweiten Beispielprogramms

4.4.7 Netzwerkschnittstellen und Übertragungsmedium

Die Klasse `Device` ist die dritte vom Protokollsimulationsmodul vorgegebene Spezialisierung der Klasse `ServiceProvider`. Sie dient der Modellierung von Netzwerkschnittstellen. Objekte dieser Klasse haben eine Adresse, auf die mit `setAddress` und `getAddress` zugegriffen werden kann. Weiterhin ist ein Zeiger auf das Übertragungsmedium notwendig, da in Version 3.0 von ODEmX Protokollsimulationen mit mehreren Übertragungsmedien möglich sind und die Netzwerkschnittstellen diesen zugeordnet werden müssen.

Von der Struktur ähnelt `Device` der Klasse `Service`, da es hier ebenfalls einen internen Netzwerk-SAP gibt, über den die Nachrichten vom Übertragungsmedium empfangen werden. Analog zu `Service` sind auch die Methoden `receive` und `handleInput` implementiert. Letztere macht dabei die gleiche Unterscheidung zwischen Senden und Empfangen, so dass in Spezialisierungen der Klasse die rein virtuellen Methoden `handleSend` und `handleReceive` zu implementieren sind.

Die Methoden `send` und `pass` sind für das Senden beziehungsweise weiterreichen von PDUs an SAPs anderer Schichten verantwortlich. Während `pass` PDUs einfach

an den gegebenen SAP der oberen Schicht durchreicht, ist `send` eine komplexe Operation, da sie auch die Kollisionserkennung bei der Nachrichtenübertragung beachten muss. Eine **Kollision** tritt ein, wenn zwei miteinander verbundene Knoten gleichzeitig senden. Zur Detektion zählt daher jedes Device-Objekt die gerade aktiven Übertragungen.

Bei Aufruf von `send` erfolgt mit der Methode `busy` als allererstes die Abfrage, ob gerade eine Übertragung stattfindet, also der Zähler nicht null ist. Wenn dem so ist, wird `send` beendet und gibt den Wert `false` zurück. Ansonsten wird der Zähler um eins erhöht und den Nachbarn durch Ruf von `startTransmission` der Beginn einer Sendeperiode mitgeteilt, womit auch deren Zähler inkrementiert wird. Überschreitet dabei der Übertragungszähler einer Netzwerkschnittstelle den Wert eins, so wird eine Kollision festgestellt. Der Sendevorgang ist durch eine von der PDU-Größe abhängige Wartezeit modelliert. Ihre Berechnung muss in Device-Spezialisierungen durch Implementation der virtuellen Methode `computeTransmissionDuration` definiert werden. Ein `odemx::sync::Timer`-Objekt wird verwendet, um den Prozess nach Ablauf der Sendedauer zu reaktivieren.

Das Verhalten bei Kollisionen, die während der abzuwartenden Sendedauer auftreten, kann durch die virtuelle Methode `hasCollisionDetection` bestimmt werden. Diese Methode entscheidet, ob das Device-Objekt über eine Kollisionserkennung verfügt. Wenn dem so ist, wird es bei Feststellung einer Kollision vor Ablauf der Sendedauer durch ein internes `odemx::sync::Memory`-Objekt reaktiviert. In diesem Fall wird die Datenübertragung sofort abgebrochen. Liefert `hasCollisionDetection` jedoch den Wert `false`, so wird immer die gesamte Sendedauer abgewartet. In jedem Fall wird bei Auftreten einer Kollision eine ungültige PDU weitergegeben. Das Ende einer Übertragung wird durch die Methode `endTransmission` bekannt gemacht, die auch die Weiterleitung der PDU-Objekte übernimmt. Erst nach diesem Aufruf wird der Übertragungszähler dekrementiert.

Die Klasse `Device` ist eng verbunden mit der Klasse `Medium`, welche das Übertragungsmedium modelliert. Mit dem erweiterten Protokollsimulationsmodul ist es nun möglich, mehrere Objekte dieser Klasse innerhalb eines Simulationsexperiments zu verwenden, so dass sich auch gemischte Netzwerke simulieren lassen, in denen Knoten über verschiedene Übertragungsmedien kommunizieren. Das `Medium` verwaltet alle per `registerDevice` bekannt gemachten Netzwerkschnittstellen in einer `std::map`, wobei stringbasierte Adressen als Schlüssel verwen-

det werden. Der Zugriff auf die registrierten Objekte erfolgt durch die Methode `getDevice`, welche den unter der angegebenen Adresse abgelegten Device-Zeiger zurückgibt.

Die Topologie des Netzwerks wird anhand einer zweidimensionalen `std::map` beschrieben, wobei die Schlüssel Zeiger auf Device-Objekte sind. Lesezugriff wird über die Methode `getTopology` gewährt. Die darin gespeicherten Links zwischen Netzwerkschnittstellen sind immer unidirektional, so dass für bidirektionale Verbindungen zwei Einträge vorgehalten werden. Jeder Link wird durch ein `LinkInfo`-Objekt beschrieben, welches die Verzögerungszeit und das assoziierte Fehlermodell speichert. Das Einfügen von Links erfolgt durch die Methode `addLink`, welche mindestens eine Quelladresse, eine Zieladresse und die Verbindungsverzögerung als Parameter fordert. Optional kann eine Verbindung auch gleich bidirektional eingetragen oder ein spezielles Fehlermodell festgelegt werden. Wird bei Linkerzeugung kein Fehlermodell angegeben, so wird das Default-Fehlermodell des `Medium`-Objekts verwendet. Dieses kann wie bei der Klasse `Service` mit der Methode `setErrorModel` gesetzt werden. Das Entfernen von Links wird durch die Methode `removeLink` ermöglicht.

Die Umsetzung von Nachrichtenübertragungen basiert auf dem Zusammenspiel der Klassen `Device` und `Medium`. Durch den senderseitigen Aufruf von `signalTransmissionStart` erzeugt das `Medium` für jeden Nachbarn des Senders ein Simulationsereignis des Typs `TransmissionStart`, das nach der angegebenen Linkverzögerung ausgeführt wird und an der benachbarten Netzwerkschnittstelle die Anzahl der aktiven Übertragungen um eins erhöht. Nach Ablauf der Übertragungsdauer für die gegebene Nachricht erfolgt erneut senderseitig der Aufruf von `signalTransmissionEnd`, wodurch das `Medium` wiederum Simulationsereignisse des Typs `TransmissionEnd` erzeugt, welche die zu übertragende Nachricht enthalten und diese nach Ablauf der Linkverzögerung bei den Nachbarn ausliefern. Erst dadurch wird der Zähler der aktiven Übertragungen am `Device`-Objekt wieder um eins verringert. Kollisionen treten nach diesem Modell also immer dann auf, wenn vor einem erwarteten `TransmissionEnd`-Ereignis ein zweites `TransmissionStart`-Ereignis eintritt. Das Fehlermodell kommt ebenfalls in der Methode `signalTransmissionEnd` zur Anwendung, indem für die zu versendende Nachricht `ErrorModel::apply` aufgerufen wird. So kann entweder die Nachricht verfälscht werden, oder `apply` kann `false` zurückgeben, wodurch die Nachricht gar nicht erst beim Empfänger ankommt.

4.4.8 Beispiel XCS 3 - Kommunikation via Übertragungsmedium

Dieser dritte Teil des Beispiels zeigt nun den höchsten Detailgrad in der Modellierung mit dem Protokollsimulationsmodul. Anstelle der Klasse XcsService wird die untere Schicht hier durch eine Device-Klasse modelliert, die ein Medium zur Nachrichtenübertragung verwendet.

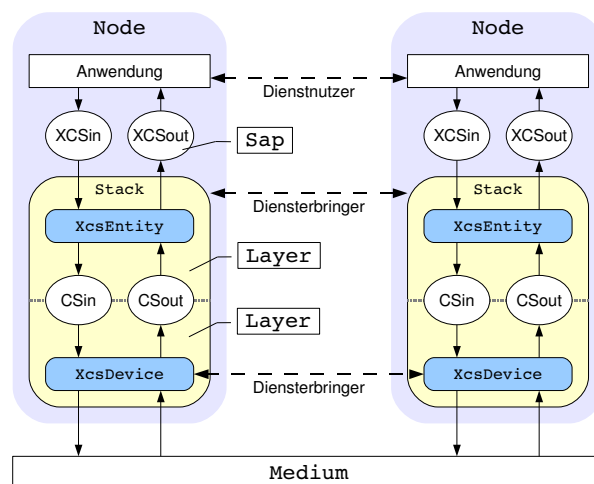


Abbildung 4.9: Knotenmodell des Beispiels auf dem höchsten Detailgrad: der Dienst XCS wird durch einen zweischichtigen Protokollstack und Kommunikation über das Medium erbracht

Abbildung 4.9 zeigt die geänderte Knotenstruktur sowie die Beziehung zum Übertragungsmedium. Die Verbindungen zwischen Knoten wurden in den Beispielabschnitten XCS 1 und XCS 2 durch die zentrale Registrierung aller XcsService-Objekte hergestellt. Stattdessen erfolgt hier bei der Initialisierung des Protokollstacks die Registrierung von Device-Objekten am Medium. Die Links zwischen den Knoten werden später anhand der Adressen registrierter Netzwerkschnittstellen festgelegt.

```

1 class XcsDevice: public Device {
2 public:
3     XcsDevice( base::Simulation& sim, const data::Label& label )
4     :   Device( sim, label ) {
5         addSap( "CSin" );
6     }

```

```
7   virtual void handleSend( const std::string& sap, PduPtr p ) {
8       info << log( "send" );
9       send( p );
10  }
11  virtual void handleReceive( PduPtr p ) {
12      info << log( "pass" );
13      pass( "CSout", p );
14  }
15  virtual base::SimTime computeTransmissionDuration( std::size_t pduSize ) {
16      return static_cast< base::SimTime >( pduSize );
17  }
18  };
```

Quelle 4.10: Device-Spezialisierung, die TransmissionData-PDUs via Übertragungsmedium versendet

Quelle 4.10 zeigt die Klasse `XcsDevice`, die in ihrer Implementation der Klasse `XcsService` ähnelt. Ein Unterschied ist die fehlende `holdFor`-Anweisung in der Methode `handleSend`. Diese Warteperiode muss nicht extra angegeben werden, da die Übertragungsdauer bereits in der Methode `send` durch den internen Aufruf von `computeTransmissionDuration` berücksichtigt wird. Diese rein virtuelle Methode muss allerdings vom Anwender implementiert werden. Wie in den vorangegangenen Beispielabschnitten wird die Größe der Nachricht als Zeitwert interpretiert, um die Warteperiode zu bestimmen.

Die SAP-Konfiguration bleibt in diesem Beispiel unverändert. Daher sind für die Klassen `XcsEntity` und `Node` keinerlei Anpassungen notwendig. Die Initialisierung des Protokollstacks muss hingegen für den Einsatz von `XcsDevice` und `Medium` angepasst werden. Quelle 4.11 zeigt die entsprechende Variante der Funktion `makeStack`.

```
1  Medium medium( getDefaultSimulation(), "TransmissionMedium" );
2
3  std::auto_ptr< Stack > makeStack( base::Simulation& sim,
4      const data::Label& label, const Addr& address )
5  {
6      Layer* entityLayer = new Layer( sim, label + " entity layer" );
7      entityLayer->addServiceProvider( new XcsEntity( sim, label + " entity" ) );
8      Layer* deviceLayer = new Layer( sim, label + " device layer" );
9      Device* device = new XcsDevice( sim, label + " device" );
10     deviceLayer->addServiceProvider( device );
11     medium.registerDevice( address, *device );
12
13     std::auto_ptr< Stack > stack( new Stack( sim, label ) );
```

```
14     stack->addLayer( entityLayer );
15     stack->addLayer( deviceLayer );
16     return stack;
17 }
```

Quelle 4.11: Funktion zur Erzeugung des Protokollstacks, die Device und Medium berücksichtigt

Zunächst wird ein globales Medium-Objekt angelegt, das mit dem Default-Simulationskontext initialisiert wird. In der Funktion `makeStack` werden wie zuvor zwei Layer-Objekte erzeugt, wobei der Dienstbringer der oberen Schicht ein `XcsEntity`-Objekt ist. Anstelle des `XcsService` wird der unteren Schicht nun ein `XcsDevice`-Objekt als Dienstbringer zugeordnet. Analog zur Registrierung eines Service-Objekts wird diese Netzwerkschnittstelle mit ihrer Adresse am Medium registriert, damit anschließend die Topologie anhand von Adressen aufgebaut werden kann.

```
1 int main() {
2     base::Simulation& sim = getDefaultSimulation();
3     sim.addConsumer( data::channel_id::info,
4                     data::output::OStreamWriter::create( std::cout ) );
5
6     Node sender( "Sender", "Addr0", "Addr1" );
7     Node receiver( "Receiver", "Addr1" );
8     sender.activate();
9     receiver.activate();
10    medium.addLink( "Addr0", "Addr1", 0 );
11
12    sim.run();
13 }
```

Quelle 4.12: Angepasste `main`-Funktion, die einen Link zwischen zwei Netzwerkschnittstellen anhand ihrer Adressen hinzufügt

Die in Quelle 4.12 abgebildete `main`-Funktion demonstriert, wie zwei Netzwerkschnittstellen anhand ihrer Adressen verbunden werden. Da das Medium die Topologie verwaltet, wird daran die Methode `addLink` mit den Adressen der Netzwerkschnittstellen beider Knoten aufgerufen. Die hierbei erzeugte Verbindung ist unidirektional und hat eine Verzögerung von null Zeiteinheiten. Als viertes Argument könnte `true` angegeben werden, um einen bidirektionalen Link einzutragen. Als fünftes Argument kann dann noch für jeden einzelnen Link ein `shared_ptr`-verwaltetes `ErrorModel`-Objekt übergeben werden.

Der Ablauf des Programms ist nun folgender: Der Sender verschickt eine `StringData`-Nachricht über den SAP `XCSin`, während der Empfänger am SAP

XCSout lauscht. Die Nachricht passiert die XcsEntity des Senders und wird als TransmmissionData verpackt an die XcsDevice weitergegeben. Der Transport zum Empfänger erfolgt in Koordination mit dem Medium, welches für den Start und das Ende der Nachrichtenübertragung Simulationseignisse erzeugt, von denen das zweite die Nachricht an die XcsDevice des Empfängers weitergibt. Von dort aus gelangt die Nachricht wie gehabt über die XcsEntity und den SAP XCSout zur Anwendungsschicht des Empfängerknotens. Die Ausgabe des Programms ist in Quelle 4.13 dargestellt.

```
1 ODEmX Log
2 =====
3 0 info: Sender(Node) sending PDU via XCSin [to=Addr1 | data=message]
4 0 info: Sender entity(XcsEntity) send
5 0 info: Sender device(XcsDevice) send
6 23 info: Receiver device(XcsDevice) pass
7 23 info: Receiver entity(XcsEntity) pass
8 23 info: Receiver(Node) received PDU via XCSout [from=Addr0 | data=message]
```

Quelle 4.13: Ausgabe des dritten Beispielprogramms

4.4.9 Speicherverwaltung

Strukturtechnisch basiert die Implementation des Moduls auf hierarchischen Klassenbeziehungen, wodurch die Speicherverwaltung erheblich erleichtert wird. Die Besitzverhältnisse von Objekten sind dabei eindeutig vorgegeben und bauen aufeinander auf. So werden SAPs von den Dienstbringern verwaltet. Die Dienstbringer gehören wiederum zu den Schichten, an denen sie registriert sind, und der Protokollstack verwaltet seine Schichten. Lediglich die Protokolldateneinheiten und die Fehlermodelle finden keine Einordnung in dieser Besitzhierarchie. PDUs können von Dienstbringern in allen Schichten erzeugt werden, und es ist manchmal schwer zu überblicken, wann ein Pdu-Objekt nicht mehr benötigt wird und gelöscht werden muss. ErrorModel-Objekte können ebenso von nur einer oder von vielen Komponenten benötigt werden. Daher ist in beiden Fällen die Verwendung des Klassentemplates `shared_ptr` sinnvoll, um die Speicherverwaltung durch Referenzzählung zu organisieren. In der Anwendung des Moduls müssen durch den Nutzer nur die Objekte verwaltet werden, welche von der Anwendungsschicht als Dienstbringer eines Kommunikationsdienstes in Anspruch genommen werden, und gegebenenfalls die Übertragungsmedien.

4.5 Zusammenfassung

Die in Abschnitt 4.3.1 dargelegte Problemanalyse hat eine ganze Reihe von Schwächen des Protokollsimulationsmoduls offenbart. Ein Hauptproblem war dabei der Automatisierungsansatz bei der Erzeugung von Protokollstacks, wodurch dem Anwender Arbeit erspart werden sollte. Der Analyse folgte jedoch die Erkenntnis, dass dadurch die Modellierungsfreiheit hinsichtlich der Struktur des Protokollstacks zu stark eingeschränkt wurde.

Die Entfernung der Automatismen erlaubt nun Protokollsimulationen mit beliebigen Simulationskontexten, insbesondere auch mit dem von der Bibliothek bereitgestellten Default-Kontext. Protokollstacks sind durch die Überarbeitung freier konfigurierbar, da Protokollinstanzen nicht mehr von den Schichten, sondern vom Anwender initialisiert werden. So können jetzt unterschiedlich konfigurierte Stacks innerhalb eines Simulationsexperiments erzeugt werden. Auch die Schichten sind nun variabler und ermöglichen die Verwendung mehrerer Dienstbringer unterschiedlichen Typs, solange diese von einer bestimmten Basisklasse abgeleitet sind.

Anders als zuvor sind Protokollsimulationen ohne den vorgegebenen Simulationskontext nicht mehr auf ein einziges Übertragungsmedium pro Simulationsexperiment beschränkt, wodurch auch gemischte Netzwerke simuliert werden können. Das Zusammenspiel zwischen Netzwerkknoten und Übertragungsmedium wurde um das Konzept der Netzwerkschnittstellen erweitert, so dass einzelne Knoten durchaus mehrere solche Schnittstellen besitzen können und damit Zugriff auf unterschiedliche Übertragungsmedien haben. Netzwerkschnittstellen implementieren zudem ein Konzept zur Kollisionserkennung.

Mit der Einführung einer Klasse für SAPs wurde die Zugriffsschnittstelle der Dienste sauber von der Implementation der Dienstbringer getrennt, so dass letztere nun austauschbar sind, solange sie dieselben SAPs bereitstellen. Auch für die dienstbringenden Komponenten ergibt sich durch die SAP-Klasse eine einfachere Identifikation der von Dienstnutzern zur Interaktion verwendeten Dienstzugangspunkte.

Die mit der Erweiterung eingeführten Abstraktionsstufen erlauben die inkrementelle Entwicklung von Protokollmodellen, indem wie im Beispiel XCS eine schrittweise Verfeinerung der Dienstmodellierung und des Protokollstacks vorgenommen wird, bis der gewünschte Detailgrad erreicht ist. Die beiden für die Übertragung

von Dateneinheiten vorgesehenen Klassen `Service` und `Medium` unterstützen zudem das verallgemeinerte Konzept des Fehlermodells, welches durch eine einheitliche Programmierschnittstelle bereitgestellt wird.

5 Resultate und Ausblick

Im Rahmen der vorliegenden Arbeit wurde die Simulationsbibliothek ODEMx in den Bereichen Datenerfassung und Protokollsimulation grundlegend überarbeitet. Dazu wurden die bekannten Defizite analysiert sowie neue Konzepte und Lösungsansätze entwickelt und implementiert. Als Resultat dieser Diplomarbeit liegt nun Version 3.0 der Simulationsbibliothek ODEMx vor.

Der Bereich der Datenerfassung wurde um einen allgemeinen Logging-Mechanismus erweitert, mit dem auf einfache Weise Simulationsdaten verschiedener Kategorien erzeugt, verteilt und verarbeitet werden können. Anwender brauchen daher nicht mehr ihre eigene Datenerfassung zu implementieren. Auch die Lücke in der Werkzeugkette zwischen der Simulationsbibliothek und dem Experimentmanagement-System konnte geschlossen werden, indem ODEMx mit einer Anbindung an relationale Datenbanksysteme ausgestattet wurde. Dabei kann ODEMx sowohl mit dem eingebetteten SQLite interagieren als auch mit externen Datenbanksystemen, die via ODBC angebunden werden.

Die im Rahmen der Arbeit geschaffene Logging-Bibliothek, welche fortan die Basis der Logging-Komponenten von ODEMx bildet, vereint verschiedene Konzepte moderner Logging-Bibliotheken mit der Grundstruktur des bisherigen Trace-Mechanismus. Daher konnte dieser durch ein verallgemeinertes Logging-Konzept ersetzt werden. Generizität war ein entscheidender Faktor bei der Konzeption der Logging-Komponenten, denn es sollte eine möglichst flexible Grundlage für das Logging in C++-Projekten geschaffen werden. Auch die angestrebte unkomplizierte Verwendung der Bibliothek konnte durch die Einführung von Default-Komponenten erreicht werden. In engem Zusammenhang damit steht die einfache Integration in anderen Projekten. Diese Anforderung konnte durch eine header-only Implementation der Logging-Bibliothek realisiert werden.

Im Bereich der Protokollsimulation gab es eine Vielzahl von Schwachstellen zu beheben. Dafür wurden sowohl das Gesamtkonzept als auch die einzelnen Komponenten einer gründlichen Analyse unterzogen, bei der Lösungsansätze für die

Probleme erarbeitet wurden. Das Resultat ist ein komplett überarbeitetes und konzeptuell erweitertes Protokollsimulationsmodul.

Die wichtigste Änderung betrifft die Variabilität des Protokollstacks. Einzelne Schichten des Stacks können nun beliebig viele Dienstbringer unterschiedlichen Typs enthalten, und der Zugriff auf die angebotenen Dienste erfolgt nun über Objekte einer neu eingeführten SAP-Klasse. Dienstbringer können dabei beliebig viele SAPs bedienen. Mit diesen Konzepten können Anwender nun auch innerhalb eines Simulationsprogramms verschiedene Stackstrukturen verwenden. Aber auch die Abstraktion von Stack und Netzwerkstruktur wird in der neuen Version des Moduls unterstützt.

Zudem findet der Aufbau beliebiger Netzwerkstrukturen Berücksichtigung, indem das erweiterte Modul nun die Nutzung mehrerer Übertragungsmedien in einem Simulationsprogramm zulässt. Jedes davon verwaltet seine eigene Topologie als Verbindungen zwischen Netzwerkschnittstellen. Letztere wurden als neues Konzept eingeführt, um eine bessere Interaktion der Netzwerkknoten mit dem Medium modellieren zu können. In diesem Zusammenhang hat auch die Fehlermodellierung eine Erweiterung erfahren, indem eine spezielle Schnittstelle eingeführt wurde.

5.1 Der kommende C++-Standard (C++0x)

Mit C++0x steht eine umfassende Aktualisierung des C++-Standards kurz vor der Verabschiedung. Es stellt sich also die Frage, welche erwarteten Änderungen und Erweiterungen für die Implementation von ODEMx relevant sind. Dies betrifft vor allem jene Erneuerungen, die eine Leistungssteigerung oder mehr Typsicherheit versprechen.

In Bezug auf die Effizienz sind die neu eingeführten Rvalue-Referenzen (T&&) zu beachten, die unter anderem zur Umsetzung der sogenannten Move-Semantik benötigt werden. Dabei können Daten eines als Argument übergebenen Objekts direkt übernommen werden, anstatt sie zu kopieren. Damit Klassen diese Fähigkeit erhalten, muss ein spezieller Move-Konstruktor erstellt werden, der eine Rvalue-Referenz als Parameter erhält. Es ist möglich, dass dieses Konzept zu einer Leistungssteigerung der Logging-Implementation führt, weil die Methode log

immer eine Kopie eines lokal erstellten Objekts zurückgibt. Sollte der Compiler dabei keine Optimierung (*return value optimization*) vornehmen, so könnte er durch Aufruf des Move-Konstruktors immer noch schnelleren Code erzeugen als durch Kopien. Die Klasse `SimRecord` müsste dafür um den erwähnten Move-Konstruktor erweitert werden.

Mehr Typsicherheit erhalten Zeiger durch das neue Schlüsselwort `nullptr`. Derart initialisierte Zeiger können vom Compiler dann von Integer-Werten unterschieden werden, so dass ungewollte Vergleiche von Zeigern und Integern bereits vom Compiler entdeckt werden. Weiterhin gibt es die Möglichkeit, mit dem neuen Schlüsselwort `static_assert` Bedingungen während des Kompilervorgangs zu testen. So können beispielsweise Template-Parameter auf das Vorhandensein bestimmter Eigenschaften (*type traits*) überprüft werden.

Zur leichteren Wartbarkeit kann der interne Bibliotheks-Code auch optisch verbessert werden, indem neue Konstrukte wie Range-For, einheitliche Initialisierung oder das Delegieren und Vererben von Konstruktoren verwendet werden. Ein weiteres Beispiel sind auch die neuen `default`- und `delete`-Markierungen für Methoden, mit denen explizit gekennzeichnet werden kann ob Copy-Konstruktor oder Zuweisungsoperator vom Compiler generiert oder verboten werden sollen.

Interessant sind auch die Erweiterungen der Standardbibliothek. Eine wichtige Neuerung sind standardisierte Threads. Auch wenn ODEMX seine Simulationsberechnung prinzipiell nur in einem Thread durchführt, lässt sich doch bei den Ausgabekomponenten eine Anwendung für Threads finden. Für die Datenbankanbindung werden Log-Daten zwischengespeichert und bei Erreichen der Pufferobergrenze automatisch eingepflegt. Diese Operation könnte durchaus in einem separaten Thread ablaufen, so dass ODEMX nicht durch die I/O-Operationen ausgebremst wird.

Ebenso werden nun auch die meisten Komponenten der Erweiterung TR1 offiziell Teil des C++-Standards. Darunter befinden sich neben Smart Pointern und Hash-Tabellen unter anderem auch Zufallszahlengeneratoren. Sobald die C++-Compiler den neuen Standard hinreichend unterstützen, sollte ODEMX einmal dahingehend überprüft werden, ob mit den neuen Standard-Komponenten Verbesserungen in der Implementation der Bibliothek zu erreichen sind.

A Konfigurationsmöglichkeiten der Bibliotheken

Logging-Bibliothek

Da die Logging-Bibliothek nur eine Sammlung von Header-Dateien ist, muss sie nicht separat übersetzt werden. Es genügt, sie in ein Verzeichnis zu kopieren, das im Suchpfad des Compilers enthalten ist. Zur vereinfachten Nutzung gibt es eine Header-Datei namens `CppLog.h`, welche die Header-Dateien aller Komponenten der Bibliothek einbindet.

Die Datei `setup.h` dient der Konfiguration der Bibliothekskomponenten, indem die konditionale Kompilierung durch Präprozessor-Direktiven gesteuert wird. In der mit dieser Arbeit vorliegenden Version der Logging-Bibliothek betrifft dies lediglich die Abhängigkeiten der Datenbankzugriffsschicht, welche Komponenten der Bibliothek POCO Data benötigt. Diese Funktionalität ist deaktivierbar, da der Compiler sonst Fehlermeldungen erzeugt, wenn die Komponenten nicht gefunden werden. In Tabelle A.1 sind die verfügbaren Konfigurationsmakros mit ihrer Beschreibung aufgeführt.

ODEMx Version 3.0

In Version 2.2 von ODEMx mussten beim Aufruf von `make` bestimmte Variablen auf der Kommandozeile definiert werden, um die Konfiguration der Bibliothek zu beeinflussen. Dieses Vorgehen hat sich als fehleranfällig erwiesen, weil manchmal die Bibliothek und Simulationsprogramme mit verschiedenen Einstellungen übersetzt wurden. Die Einführung der Datei `odemx/setup.h` löst dieses Problem. Außerdem wird damit die Komponentenkonfiguration von ODEMx an einem zentralen Ort eingerichtet. Tabelle A.2 zeigt die darin bereitgestellten Makros und beschreibt ihren Effekt.

Makro	Beschreibung
CPPLOG_USE_DATABASE	Mit diesem Makro kann die Aktivierung der Datenbankzugriffsschicht kontrolliert werden. Eine Deaktivierung ist notwendig, falls POCO Data nicht verfügbar ist.
CPPLOG_USE_SQLITE	Das Makro aktiviert den Konnektor für SQLite. Es aktiviert außerdem einige spezielle SQL-Anweisungen, welche die Zugriffsschicht intern verwendet, um mit dem RDBMS zu kommunizieren.
CPPLOG_USE_ODBC	Analog zu SQLite kann mit diesem Makro der ODBC-Konnektor aktiviert werden. Dabei ist zu beachten, dass die Verwendung von ODBC auch die Installation der entsprechenden Header-Dateien und einer ODBC-Bibliothek erfordert. Anderenfalls können die ODBC-Komponenten von POCO Data nicht kompiliert werden.
CPPLOG_USE_POSTGRES	Mit diesem Makro wird PostgreSQL-spezifische SQL-Anweisungen aktiviert. Dies ist notwendig, um die Mehrnutzer-Funktionalität dieser RDBMS besser auszunutzen.

Tabelle A.1: Makros zur Konfiguration von ODEMX-Komponenten

Makro	Beschreibung
ODEMX_USE_CONTINUOUS	Dieses Makro aktiviert die Komponente Continuous, welche die Simulation zeitkontinuierlicher Prozesse ermöglicht. Dafür wird der für die Simulationszeit verwendete Datentyp auf double umgestellt, default ist sonst unsigned long long.
ODEMX_USE_TRACE	Mit diesem Makro lässt sich die Erzeugung von Trace-Einträgen aktivieren. Die Deaktivierung führt zu einer deutlichen Leistungssteigerung. Diese Funktionalität sollte nur zur Fehlersuche aktiviert werden.
ODEMX_USE_OBSERVATION	Das Makro aktiviert die Observation-Funktionalität. Zur Leistungssteigerung ist es standardmäßig deaktiviert, da die Funktionalität nur selten Verwendung findet.
ODEMX_USE_ODBC	Dieses Makro aktiviert die Datenbankanbindung für externe Datenbanksysteme. Es verwendet die Makros CPPLOG_USE_ODBC und CPPLOG_USE_POSTGRESQL der Logging-Bibliothek, um ODBC-Verbindungen zu PostgreSQL-Systemen zu ermöglichen. Standardmäßig ist diese Funktionalität deaktiviert, womit automatisch SQLite als Default-RDBMS verwendet wird.
ODEMX_USE_SETJUMP	Mit diesem Makro lässt sich die ursprüngliche Koroutinen-Implementation aktivieren, die auf dem Austausch des Laufzeitstacks basiert. Standardmäßig wird ab Version 3.0 von ODEMX jedoch die auf dem Fiber-API basierende Lösung verwendet, die jeder Koroutine ihren eigenen Laufzeitstack zuordnet.

Tabelle A.2: Makros zur Konfiguration von ODEMX-Komponenten

B XML-Schema-Definitionen

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3
4 <xsd:element name="odemxlog">
5   <xsd:complexType>
6     <xsd:sequence>
7       <xsd:element name="record" type="record_type" minOccurs="0" maxOccurs="unbounded"/>
8     </xsd:sequence>
9   </xsd:complexType>
10 </xsd:element>
11
12 <xsd:complexType name="record_type">
13   <xsd:sequence>
14     <xsd:element name="time" type="xsd:string"/>
15     <xsd:element name="channel" type="xsd:string"/>
16     <xsd:element name="text" type="xsd:string"/>
17     <xsd:element name="scope" type="xsd:string"/>
18     <xsd:element name="senderlabel" type="xsd:string"/>
19     <xsd:element name="sendertype" type="xsd:string"/>
20     <xsd:element name="detail" type="nvp_type" minOccurs="0" maxOccurs="unbounded"/>
21   </xsd:sequence>
22 </xsd:complexType>
23
24 <xsd:complexType name="nvp_type">
25   <xsd:sequence>
26     <xsd:element name="name" type="xsd:string"/>
27     <xsd:element name="value" type="xsd:string"/>
28   </xsd:sequence>
29 </xsd:complexType>
30
31 </xsd:schema>
```

Quelle B.1: XML-Schema-Definition für die Ausgabe der Klasse XmlWriter

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3
4 <xsd:element name="odemxreport">
5   <xsd:complexType>
6     <xsd:sequence>
7       <xsd:element name="table" type="table_type" minOccurs="0" maxOccurs="unbounded"/>
8     </xsd:sequence>
9   </xsd:complexType>
10 </xsd:element>
11
12 <xsd:complexType name="table_type">
13   <xsd:sequence>
14     <xsd:element name="column" type="column_type" minOccurs="0" maxOccurs="unbounded"/>
15     <xsd:element name="line" type="line_type" minOccurs="0" maxOccurs="unbounded"/>
16   </xsd:sequence>
17   <xsd:attribute name="columns" type="xsd:int" use="required"/>
18   <xsd:attribute name="label" type="xsd:string" use="required"/>
19   <xsd:attribute name="lines" type="xsd:int" use="required"/>
20 </xsd:complexType>
21
22 <xsd:complexType name="column_type">
23   <xsd:attribute name="label" type="xsd:string" use="required"/>
24   <xsd:attribute name="number" type="xsd:int" use="required"/>
25   <xsd:attribute name="type" type="xsd:string" use="required"/>
26 </xsd:complexType>
27
28 <xsd:complexType name="line_type">
29   <xsd:sequence>
30     <xsd:element name="cell" type="cell_type" minOccurs="0" maxOccurs="unbounded"/>
31   </xsd:sequence>
32   <xsd:attribute name="number" type="xsd:int" use="required"/>
33 </xsd:complexType>
34
35 <xsd:complexType name="cell_type">
36   <xsd:simpleContent>
37     <xsd:extension base="xsd:string">
38       <xsd:attribute name="col" type="xsd:string" use="required"/>
39     </xsd:extension>
40   </xsd:simpleContent>
41 </xsd:complexType>
42
43 </xsd:schema>
```

Quelle B.2: XML-Schema-Definition für die Ausgabe der Klasse XmlReport

C Simulationsprogramm zum Laufzeittest

```
1 #include <odemx/odemx.h>
2 #include <deque>
3
4 class Philosopher: public odemx::base::Process {
5 public:
6     Philosopher( bool& fork1, bool& fork2, odemx::base::SimTime timePeriod )
7     :   Process( odemx::getDefaultSimulation(), "Philosopher" )
8     ,   fork1( fork1 ), fork2( fork2 ), timePeriod( timePeriod )
9     {}
10 private:
11     virtual int main() {
12         while( true ) {
13             do { // thinking
14                 spendTime();
15             } while( ! takeForks() );
16             // forks acquired, start eating
17             spendTime();
18             returnForks();
19         }
20         return 0;
21     }
22 private:
23     bool& fork1;
24     bool& fork2;
25     int timePeriod;
26 private:
27     bool takeForks() {
28         if( fork1 && fork2 ) {
29             fork1 = fork2 = false;
30             return true;
31         }
32         return false;
33     }
34     void returnForks() { fork1 = fork2 = true; }
35     void spendTime(){ holdFor( timePeriod ); }
36 };
37
38
```

```
39 int main() {  
40     std::deque< Philosopher* > phils;  
41     std::deque< bool > forks;  
42     int seats = 800;  
43     for( int i = 0; i < seats; ++i ) {  
44         forks.push_back( true );  
45     }  
46     for( int i = 0; i < seats - 1; ++i ) {  
47         phils.push_back( new Philosopher( forks[ i ], forks[ i + 1 ], 10 ) );  
48     }  
49     std::for_each( phils.begin(), phils.end(),  
50         std::tr1::bind( &Philosopher::activate, std::tr1::placeholders::_1 ) );  
51  
52     odemx::getDefaultSimulation().runUntil( 10000 );  
53     std::for_each( phils.begin(), phils.end(), odemx::DeletePtr< Philosopher >() );  
54 }
```

Quelle C.1: Implementation des Philosophenproblems in ODEMX

D Ausgaben der Beispielprogramme

Beispiel 1

```
1 Logging via Channel object
2 Logging via set Channel pointer produces output
```

Quelle D.1: Konsolenausgabe von Beispiel 1

Beispiel 2

```
1 INFO (P1) Info logging with Producer and ChannelManager
2 DEBUG (P2) Debug logging with Producer and ChannelManager
```

Quelle D.2: Konsolenausgabe von Beispiel 2

Beispiel 3

```
1 Logging class Record example, file: src/Example_Record.cpp, line: 51, scope: main()
2 Logging class Record example, file: src/Example_Record.cpp, line: 51, scope: main()
```

Quelle D.3: Konsolenausgabe von Beispiel 3

Beispiel 4

```
1 Logging filter example (main) [Info: scope=main()]
```

Quelle D.4: Konsolenausgabe von Beispiel 4

Beispiel 5

```
1 [Info: point=(1 2 3) value=123.45]
2 <record>
3 <point>(1 2 3)</point>
4 <value>123.45</value>
5 </record>
```

Quelle D.5: Konsolenausgabe von Beispiel 5

D Ausgaben der Beispielpprogramme

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <cpplog>
3   <record>
4     <point>(1 2 3)</point>
5     <value>123.45</value>
6   </record>
7 </cpplog>
```

Quelle D.6: Inhalt der Datei Example_0.xml des Beispiels 5

Beispiel 6

```
1 CREATE TABLE example_table (id INTEGER,text VARCHAR,PRIMARY KEY(id));
2 INSERT INTO "example_table" VALUES(1,'first_example_string');
3 INSERT INTO "example_table" VALUES(2,'second_example_string');
```

Quelle D.7: SQL-Anweisungen zur Archivierung der Daten in Beispiel 6

Beispiel 7

```
1 CREATE TABLE log_record(
2   id INTEGER, text VARCHAR, info_1 VARCHAR, info_2 INTEGER, info_3 REAL, PRIMARY KEY(id)
3 );
4 INSERT INTO "log_record" VALUES(1,'Logging example: SQLite database access','string detail',2,3.0);
```

Quelle D.8: SQL-Anweisungen zur Archivierung der Daten in Beispiel 7

Beispiel 8

```
1 ODEMX Log
2 =====
3 0 info: Stop Watch(StopWatch) started
4 10 info: Stop Watch(StopWatch) stopped [duration=10]
5 ODEMX ERROR: Stopwatch::stop(): stop watch is not running [Time: 10] [Object: Stop Watch (StopWatch)]
6 10 error: Stop Watch(StopWatch) Stopwatch::stop(): stop watch is not running
```

Quelle D.9: Konsolenausgabe von Beispiel 8

Beispiel 9

```
1 ODEMX Log
2 =====
3 0 info: Stop Watch(DebugStopWatch) started
4 0 info: Stop Watch(DebugStopWatch) stopped [duration=0]
```

Quelle D.10: Konsolenausgabe von Beispiel 9

Beispiel 10

```
1 ODEmx Log
2 =====
3 2009-12-30 / 10:00:00.000 debug: Stop Watch(DebugStopWatch) construction [Class Scope: DebugStopWatch]
4 2009-12-30 / 10:00:00.000 debug: Stop Watch(DebugStopWatch) started [Class Scope: DebugStopWatch]
5 2009-12-30 / 10:00:00.000 info: Stop Watch(DebugStopWatch) started
6 2009-12-30 / 10:00:00.042 debug: Stop Watch(DebugStopWatch) stopped [Class Scope: DebugStopWatch]
7 2009-12-30 / 10:00:00.042 info: Stop Watch(DebugStopWatch) stopped [duration=42]
8 2009-12-30 / 10:00:00.042 debug: Stop Watch(DebugStopWatch) destruction [Class Scope: DebugStopWatch]
```

Quelle D.11: Konsolenausgabe von Beispiel 10

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <odemxlog>
3   <record>
4     <time>2009-12-30T10:00:00.000+01:00</time>
5     <channel>info</channel>
6     <text>started</text>
7     <scope/>
8     <senderlabel>Stop Watch</senderlabel>
9     <sendertype>DebugStopWatch</sendertype>
10  </record>
11  <record>
12    <time>2009-12-30T10:00:00.042+01:00</time>
13    <channel>info</channel>
14    <text>stopped</text>
15    <scope/>
16    <senderlabel>Stop Watch</senderlabel>
17    <sendertype>DebugStopWatch</sendertype>
18    <detail>
19      <name>duration</name>
20      <value>42</value>
21    </detail>
22  </record>
23 </odemxlog>
```

Quelle D.12: Inhalt der Datei Example_Consumer.xml des Beispiels 10

Beispiel 11

```
1 CREATE TABLE odemx_simulation_run(
2   id INTEGER NOT NULL, date TIMESTAMP NOT NULL, label VARCHAR NOT NULL,
3   description VARCHAR NOT NULL, uuid CHAR(40) NOT NULL, PRIMARY KEY(id)
4 );
5 CREATE TABLE odemx_sim_record(
6   id INTEGER NOT NULL, text VARCHAR, channel VARCHAR NOT NULL,
7   class_scope VARCHAR, sender_label VARCHAR NOT NULL, sender_type VARCHAR NOT NULL,
8   sim_time VARCHAR NOT NULL, sim_id INTEGER NOT NULL, PRIMARY KEY(id)
9 );
10 CREATE TABLE odemx_sim_record_detail(
11   id INTEGER NOT NULL, name VARCHAR NOT NULL, value VARCHAR NOT NULL,
12   record_id INTEGER NOT NULL, PRIMARY KEY(id)
```

```
13 );
14 INSERT INTO "odemx_simulation_run" VALUES(
15     1, '2010-04-25 22:02:56', 'DefaultSimulation', 'no description',
16     '72bb53ad-0955-4dce-b9e4-f629f0b7cf77'
17 );
18 INSERT INTO "odemx_sim_record" VALUES(
19     1, 'started', 'info', NULL, 'Stop Watch', 'StopWatch', '2009-04-04 / 16:00:00', 1
20 );
21 INSERT INTO "odemx_sim_record" VALUES(
22     2, 'stopped', 'info', NULL, 'Stop Watch', 'StopWatch', '2009-04-04 / 16:00:10', 1
23 );
24 INSERT INTO "odemx_sim_record_detail" VALUES( 1, 'duration', '10', 2 );
```

Quelle D.13: SQL-Anweisungen zur Archivierung der Daten in Beispiel 11

Beispiel 12

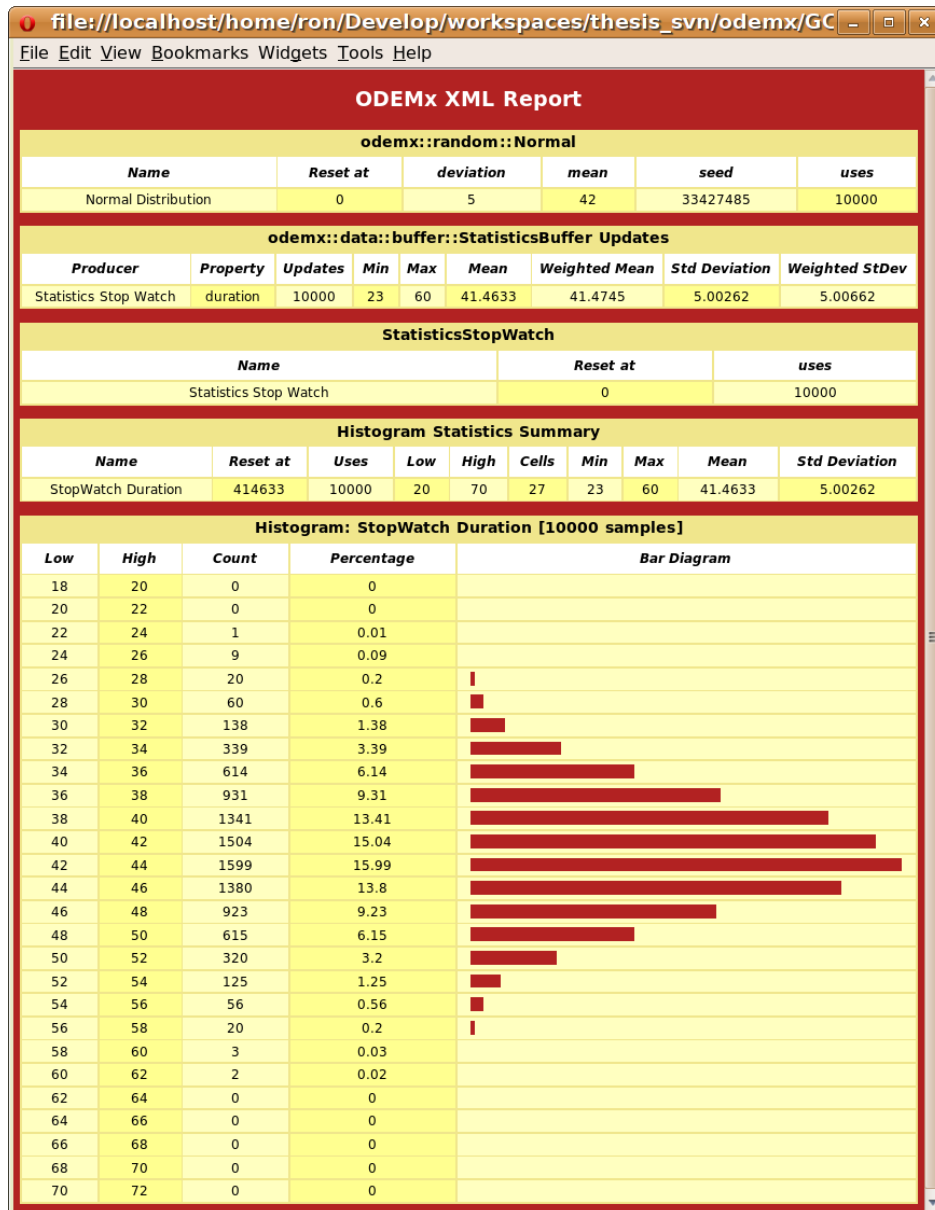


Abbildung D.1: XML-Report von Beispiel 12 dargestellt im Browser

Literaturverzeichnis

- [Apache] *Apache log4cxx*
<http://logging.apache.org/log4cxx/index.html>.
(Abgerufen am 10. November 2009)
- [Becker06] Pete Becker: *The C++ Standard Library Extensions A Tutorial and Reference*. Addison-Wesley, 2006
- [Boost] *Boost.Log*
<http://boost-log.sourceforge.net/libs/log/doc/html/index.html>.
(Abgerufen am 10. November 2009)
- [Chen76] Peter Pin-Shan Chen: *The Entity-Relationship Model: Towards a Unified View of Data*. ACM Transactions on Database Systems Vol.1, No. 1, 1976
- [Codd70] Edgar Frank Codd: *A Relational Model for Large Shared Data Banks*. CACM, 13:6, Juni 1970
- [DTL] *Database Template Library*
<http://dtemplatelib.sourceforge.net/>
(Abgerufen am 09. April 2010)
- [Elma00] Ramez Elmasri, Shamkant B. Navathe: *Fundamentals of Database Systems, Third Edition*. Addison-Wesley, 2000
- [Eves06] Ingmar Eveslage: *Reimplementation einer Stahlwerkssimulation auf Basis der Simulationsbibliothek ODEMx*. Humboldt-Universität zu Berlin, 2006
- [Fischer08] Joachim Fischer, Frank Kühnlenz, Klaus Ahrens: *Model-based Development of Self-organizing Earthquake Early Warning Systems*. Genf, 2008
- [Fischer96] Joachim Fischer, Klaus Ahrens: *Objektorientierte Prozeßsimulation in C++*. Addison-Wesley, 1996
- [Gerst03] Ralf Gerstenberger: *ODEMx Neue Lösungen für die Realisierung von C++-Bibliotheken zur Prozesssimulation*. Humboldt-Universität zu Berlin, 2003

- [Gülcü03] Ceki Gülcü: *The Complete Log4j Manual*. QOS.ch, 2003
- [Gupta05] Samudra Gupta: *Pro Apache Log4j*. Apress, 2005
- [Härder83] Theo Härder, Andreas Reuter: *Principles of Transaction-Oriented Database Recovery in Computing Surveys*, Vol.15, No.4. ACM, 1983
- [Hender06] Thomas R. Henderson, Sumit Roy, Sally Floyd, George F. Riley: *ns-3 Project Goals*. WNS2'06, Pisa, 2006
- [Issari08] Teerawat Issariyakul, Ekram Hossain: *Introduction to Network Simulator NS2*. Springer, 2008
- [Jeruch00] Michel C. Jeruchim, Philip Balaban, K.Sam Shanmugan: *Simulation of Communication Systems (2nd Edition)*. Springer, 2000
- [Kluth07] Ronald Kluth: *Ereignisorientierte Computersimulation mit ODEMX*. Humboldt-Universität zu Berlin, 2007
- [König03] Hartmut König: *Protocol Engineering*. B.G. Teubner Verlag, 2003
- [LibODBC++] *libODBC++*
<http://sourceforge.net/projects/libodbcxx/>
(Abgerufen am 09. April 2010)
- [OPNET] OPNET Technologies, Inc.: *OPNET Modeler*
http://www.opnet.com/solutions/network_rd/modeler.html
(Abgerufen am 12. Januar 2010)
- [OTL] *Oracle, ODBC and DB2-CLI Template Library*
<http://otl.sourceforge.net/>
(Abgerufen am 09. April 2010)
- [POCO] *POCO C++ Libraries*
<http://www.pocoproject.org>.
(Abgerufen am 09. April 2010)
- [QualNet] Scalable Network Technologies, Inc.: *QualNet Developer*
<http://www.scalable-networks.com/products/qualnet/>
(Abgerufen am 12. Januar 2010)
- [SAFER] *SAFER - Seismic eArly warning For EuRope*
<http://www.saferproject.net>.
(Abgerufen am 11. November 2009)
- [Schub04] Matthias Schubert: *Datenbanken*. B.G. Teubner Verlag, 2004

- [SOCl] *SOCl - The C++ Database Access Library*
<http://soci.sourceforge.net/>
(Abgerufen am 09. April 2010)
- [SQLite] *SQLite*
<http://www.sqlite.org>.
(Abgerufen am 27. November 2009)
- [SQLiteAnw] *Anwendungsfälle von SQLite als eingebettete Datenbank*
<http://www.sqlite.org/famous.html>
(Abgerufen am 09. April 2010)
- [Step88] Alexander A. Stepanov, David R. Musser: *Generic Programming*. ISSAC, 1988
- [Tanen02] Andrew S. Tanenbaum: *Computer Networks (4th Edition)*. Prentice Hall, 2002
- [Tranter04] William H. Tranter, K. Sam Shanmugan, Theodore S. Rappaport, Kurt L. Kosbar: *Principles of Communication Systems Simulation with Wireless Applications*. Prentice Hall, 2004
- [UTPP] *UnitTest++*
<http://unittest-cpp.sourceforge.net/>
(Abgerufen am 12. Januar 2010)
- [Vande02] David Vandevoorde, Nicolai M. Josuttis: *C++ Templates: The Complete Guide*. Addison-Wesley, 2002
- [Varga08] András Varga, Rudolf Hornig: *SIMUTools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems workshops: An overview of the OMNeT++ simulation environment*. ICST Brüssel, 2008
- [Zeigler00] Bernard Zeigler, Tag Gon Kim, Herbert Praehofer: *Theory of Modeling and Simulation, Second Edition*. Academic Press, 2000

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Berlin, den 5. Mai 2010

.....

Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den 5. Mai 2010

.....