Regelbasierte Replikationsstrategie für heterogene, autonome Informationssysteme

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Heinrich Niemann

Kiel
2009

1. Gutachter: Prof. Dr. Wilhelm Hasselbring

2. Gutachter: Prof. Dr. Stefan Conrad

Datum der mündlichen Prüfung: 1.7.2009

Zusammenfassung

Die Replikation von Daten stellt in heterogenen, autonomen Informationssystemen hohe Ansprüche an eine geeignete Replikationsstrategie. So müssen Schreib- und Lesezugriffe auf die Replikate derart koordiniert werden, dass ein optimaler Kompromiss hinsichtlich der konkurrierenden Replikationsziele Verfügbarkeit, Performance und Konsistenz erreicht wird. Dieses Abwägen hinsichtlich der Replikationsziele wird dadurch erschwert, dass die beteiligten Informationssysteme ihre Systemzustände ändern und dass auf diese Veränderungen reagiert werden muss. Für diese Anwendungsbereiche wurden adaptive Replikationsstrategien entwickelt, die sich zur Laufzeit den veränderten Systemzuständen anpassen.

In dieser Dissertation wird die regelbasierte Replikationsstrategie RegRess sowie die Regelsprache RRML vorgestellt, die die Formulierung von Replikationsregeln für RegRess ermöglicht. Bei RegRess erfolgt die Koordination für Schreib- und Lesezugriffe auf Basis dieser Regeln, indem vor jedem Zugriff eine Inferenz der Regeln durchgeführt wird, wodurch die von dem Zugriff betroffenen Replikate ermittelt werden. Durch diese Vorgehensweise wird unterschiedlichstes Konsistenzverhalten von RegRess realisiert, insbesondere werden temporäre Inkonsistenzen toleriert. Eine Regelmenge mit für den Anwendungsfall spezifizierten Regeln bildet die Konfiguration von RegRess. Weil in den Regeln Systemzustände berücksichtigt werden können, kann zur Laufzeit das Verhalten angepasst werden. Somit handelt es sich bei RegRess um eine konfigurierbare, adaptive Replikationsstrategie.

Mit der Regelsprache RRML können so genannte Reaktionsregeln formuliert werden. Bei einer Regel der RRML wird auf Zugriffe auf Replikationseinheiten, die Teilmengen aller logischen Objekte bilden, reagiert, indem die Bedingung der Regel geprüft wird. Die Bedingung einer Replikationsregel beinhaltet neben Gültigkeitszeiträumen vor allem fachliche und technische Konsistenzbedingungen, die z.B. eine Reaktion auf zeitlichen Verzug der Aktualisierungen oder Nicht-Verfügbarkeit eines Rechners erlauben. Wenn die Bedingung einer Replikationsregel erfüllt ist, dann wird im Aktionsteil der Regel die Zugriffsart auf die Replikate festgelegt. Weil die Replikationsregeln widersprüchliche Aktionen auslösen können, beinhaltet die RRML eine Widerspruchsbehandlung.

Zur Realisierung der Replikationsstrategie RegRess dient der Replikationsmanager KARMA, der neben den Protokollen für die Schreib- und Lesezugriffe einen Regelinterpreter für die Replikationsregeln der RRML beinhaltet. Für den KARMA wird eine Softwarearchitektur konzipiert, wobei eine Spezifikation der einzelnen Komponenten des KARMA vorgenommen wird. Ein wichtiger Aspekt bei den Zugriffen auf die Replikate ist die transaktionale Anbindung der beteiligten Systeme. Daher werden Transaktionskonzepte spezifiziert, die bei der Umsetzung der Protokolle benötigt werden.

Der Replikationsmanager KARMA ist mittels Plugin-Mechanismus in den Simulator F4SR integriert, der im Rahmen dieser Dissertation entstanden ist. Mit dem F4SR können Replikationsstrategien oder unterschiedliche Konfigurationen einer Replikationsstrategie verglichen werden. So kann beispielsweise das Verhalten hinsichtlich der Replikationsziele Verfügbarkeit, Performance und Konsistenz für verschiedene Regelmengen untersucht werden. Der F4SR bietet für die Analyse verschiedene Diagramme, die die Ergebnisse eines Simulationslaufs illustrieren.

Inhaltsverzeichnis

1. Einleitung 1
1.1. Motivation . . . 2
1.2. Problemstellung und Zielsetzung . . . 3
1.3. Aufbau der Arbeit . . . 5

I. Grundlagen und verwandte Arbeiten 7

2. Grundlegende Konzepte 9
2.1. Verteilte Systeme . . . 9
2.1.1. Definition, Transparenz und Zeitstempel . . . 9
2.1.2. Kommunikationsmodelle, Kommunikationsfehler und Middleware . . . 11
2.1.3. Klassifizierung . . . 12
2.2. Transaktionen . . . 13
2.2.1. Definition, ACID-Eigenschaften und elementare Operationen . . . 13
2.2.2. Commit-Protokolle und XA-Protokoll . . . 15
2.2.3. Synchronisation von Nebenläufigkeit und Serialisierbarkeit . . . 17
2.2.4. Abgeschwächte Serialisierbarkeitskriterien . . . 18
2.2.5. Weiterführende Transaktionskonzepte: Sagas und Queued Transactions . . . 20
2.3. Allgemeine Konsistenzbegriffe und Korrektheit . . . 21
2.3.1. Konsistenz, Korrektheit und Integrität . . . 22
2.3.2. Integritäts- bzw. Konsistenzbedingungen . . . 23
2.3.3. Daten- und Transaktionskonsistenz, abgeschwächte Konsistenz . . . 24
2.3.4. Konsistenzmodelle verteilter, gemeinsamer Speicher . . . 25

3. Datenreplikation und verwandte Arbeiten 27
3.1. Ziele der Replikation . . . 27
3.2. Grundlagen der Replikation . . . 29
3.2.1. Begriffe und Definitionen . . . 29
3.2.2. Konsistenz in replizierten Datenbanken . . . 33
3.2.3. Abgeschwächte Konsistenzbegriffe in replizierten Datenbanken . . . 35
3.2.4. Klassifizierungen . . . 40
3.3. Traditionelle Replikationsstrategien . . . 42
3.3.1. Synchrone Replikationsstrategien . . . 43
3.3.2. Asynchrone Replikationsstrategien . . . 46
3.4. Verwandte Arbeiten . . . 49
3.4.1. Replikation und Datenintegration . . . 49
3.4.2. Adaptive Replikationsstrategien . . . 51

II. Regelbasierte Replikationsstrategie 55

4. Regelbasierte Replikationsstrategie RegRess 57
4.1. Annahmen und verwendete Notationen . . . 58
4.1.1. Annahmen . . . 58
4.1.2. Begriffe und Notationen . . . 60
4.2. Regelbasierte Koordination der Zugriffe . . . 62
4.2.1. Konfigurierbare, adaptive Replikationsstrategie mittels Regeln . . . 63
4.2.2. Koordination von Schreibzugriffen . . . 67
4.2.3. Koordination von Lesezugriffen . . . 75
4.2.4. Inferenz bei Schreib- und Lesezugriffen . . . 78
4.2.5. Korrektheitskriterium Letztendliche Konsistenz . . . 81
4.3. Konsistenz von RegRess . . . 84
4.3.1. Fachliche Konsistenzbedingungen . . . 85
4.3.2. Technische Konsistenzbedingungen . . . 88
4.3.3. Konsistenzgrad von RegRess . . . 90
4.4. Trade-Off der Replikationsziele . . . 104
4.4.1. Rationales Entscheiden bei Replikation . . . 104
4.4.2. Verfügbarkeit, Performance und Konsistenz bei RegRess . . . 105
4.4.3. Multiattributive Wertfunktion . . . 109

5. Regelsprache RRML 111
5.1. Zielsetzung, Regeldarstellung und Notation . . . 112
5.2. Anforderungen an die RRML . . . 114
5.3. Konzeptioneller Entwurf der RRML . . . 115
5.3.1. Formale Spezifikation der Regeln . . . 115
5.3.2. Regeln für die synchrone Aktualisierung . . . 118
5.3.3. Regeln für die asynchrone Aktualisierung . . . 123
5.3.4. Regeln für Lesezugriffe . . . 128
5.3.5. Behandlung von widersprüchlichen Regeln . . . 132
5.3.6. Bedingungen der Regeln . . . 136
5.3.7. Verifikation der Anforderungen . . . 140
5.4. XML-Repräsentation der RRML . . . 142
5.4.1. XML-basierte RRML-Syntax . . . 143
5.4.2. Mapping in der RRML . . . 146
5.4.3. RRML-Schema . . . 147
5.5. Analyse und Bewertung der RRML . . . 147
5.5.1. Terminierung . . . 148
5.5.2. Konfluenz . . . 149
5.5.3. Bewertung . . . 150

III. Software-Entwurf und Evaluation 151

6. Replikationsmanager KARMA 153
6.1. Softwarearchitektur KARMA . . . 153
6.2. Komponentenspezifikation KARMA . . . 157
6.3. Transaktionsverarbeitung des KARMA . . . 167
6.3.1. Transaktionskonzepte . . . 167
6.3.2. Regelbasierter, dynamischer Transaktionsmanager . . . 173

7. Prototypische Implementierung und Evaluation 177
7.1. Integration heterogener Informationssysteme mittels der Java EE 178
7.2. Implementierte Prototypen durch studentische Arbeiten 180
7.3. Simulation der Replikationsstrategie RegRess 183
7.3.1. Framework für die Simulation von Replikationsstrategien 183
7.3.2. Implementierung des Prototypen KARMA 187
7.3.3. Das KARMA-Plugin für das Simulationsframework F4SR 197
7.3.4. Evaluation der Simulationsexperimente 200

IV. Zusammenfassung, Fazit und Ausblick 211

8. Zusammenfassung und Fazit 213

9. Ausblick 217

Anhang 221

A. Regelsprachen und regelbasierte Systeme 223
A.1. Regeln 223
A.2. Regelsprachen 224
A.3. Regelbasierte Systeme 225

B. Tabellarische Aufstellung der Anforderungen an das regelbasierte System 227

C. UML-Modell des KARMA-Prototypen 229

Definitionen 235
Nummerische Sortierung 235
Alphabetische Sortierung 237

Abbildungsverzeichnis 239

Verzeichnis der Listings 240

Literaturverzeichnis 242

1. Einleitung

Die Replikation von Daten, d.h. das mehrfache Speichern von Datenkopien auf verschiedenen Rechnern, war, ist und bleibt ein relevantes Forschungsthema im Bereich von Datenbanken und Informationssystemen. Der Grund für die Datenreplikation (im Folgenden einfach Replikation) liegt darin, dass die Verfügbarkeit und/oder Performance des Gesamtsystems erhöht werden soll bzw. zwischen bereits bestehenden autonomen Anwendungssystemen Daten abgeglichen werden sollen. Mit Replikation soll erreicht werden, dass alle Kopien eines Objekts den gleichen Wert haben und bei Änderung einer Kopie automatisch die Änderung auch an den anderen Kopien durchgeführt wird. Durch das mehrfache Speichern identischer Kopien eines Objekts kann dann auf eine beliebige Kopie des Objekts zugegriffen werden, um z.B. den Wert des Objekts zu lesen. Der Vorteil dieser erhöhten Verfügbarkeit wird dadurch erkauft, dass eine Replikationsstrategie umgesetzt werden muss, die die Schreib- und Lesezugriffe koordiniert. Durch eine geeignete Koordination sollen vor allem Konsistenzeigenschaften gewährleistet werden.

In einem homogenen Systemumfeld, in dem z.B. einheitliche Datenbankmanagementsysteme (DBMS, siehe z.B. [HR01, EN02]) eingesetzt werden, kann die Replikation im Allgemeinen vom DBMS selbst übernommen werden. Es muss lediglich bestimmt werden, welche Daten repliziert werden sollen. Demgegenüber müssen in einer heterogenen Systemlandschaft zusätzlich Fragen wie die Interoperabilität der Systeme, lokale Schemata oder Semantik der Daten geklärt werden. Häufig wird zusätzlich Autonomie der betroffenen Systeme gefordert, d.h. durch die Koordination der Replikationsstrategie darf die Eigenständigkeit der Systeme nicht bzw. nicht zu stark eingeschränkt werden. Als Beispiel heterogener Systeme dienen Krankenhausinformationssysteme (KIS), die in der Praxis immer aus vielen unterschiedlichen Anwendungssystemen bestehen. Zwischen den meisten dieser Systeme müssen Patientendaten ausgetauscht, sprich repliziert werden.

Die wachsende Komplexität der verteilten Systeme und die gestiegenen Anforderungen an die Datenhaltung erfordern häufig eine hohe Flexibilität der Replikationsstrategie. Daher werden vermehrt Replikationsstrategien entwickelt, die sich zur Laufzeit den Zustandsänderungen des zugrunde liegenden, verteilten Systems anpassen. Auch die in dieser Dissertation entwickelte Replikationsstrategie RegRess (**Reg**elbasierte **Re**plikation**ss**trategie) reiht sich in dieser Kategorie ein. Bei RegRess erfolgt die Konfiguration durch die Spezifikation von Replikationsregeln, die in einer eigens entwickelten Sprache formuliert werden. Eine Adaption zur Laufzeit ergibt sich dadurch, dass die Koordination der Schreib- und Lesezugriffe auf Basis einer Inferenz der Regeln bestimmt wird. Dabei erfolgt die Schlußfolgerung der Regeln bei jedem Zugriff, um aktuelle Kennzahlen des Systems zu berücksichtigen. Somit handelt es sich bei RegRess um eine konfigurierbare, adaptive Replikationsstrategie.

Nachfolgend wird in Abschnitt 1.1 die Motivation für diese Dissertation vorgestellt, die im Wesentlichen auf der Integration von klinischen Informationssystemen beruht. Der Abschnitt 1.2 beschreibt die Problemstellung und die Ziele, die mit dieser Arbeit erreicht werden sollen. Neben der Konzeption der regelbasierten Replikationsstrategie RegRess ist die Entwicklung der Regelsprache RRML (**R**eplication **R**ule **M**arkup **L**anguage) sowie eine Softwarearchitektur für einen Replikationsmanager zu nennen, der die regelbasierte Replikationsstrategie RegRess inklusive eines geeigneten Regelinterpreters implementiert. Abschließend wird in Abschnitt 1.3 der Aufbau dieser Dissertation erläutert.

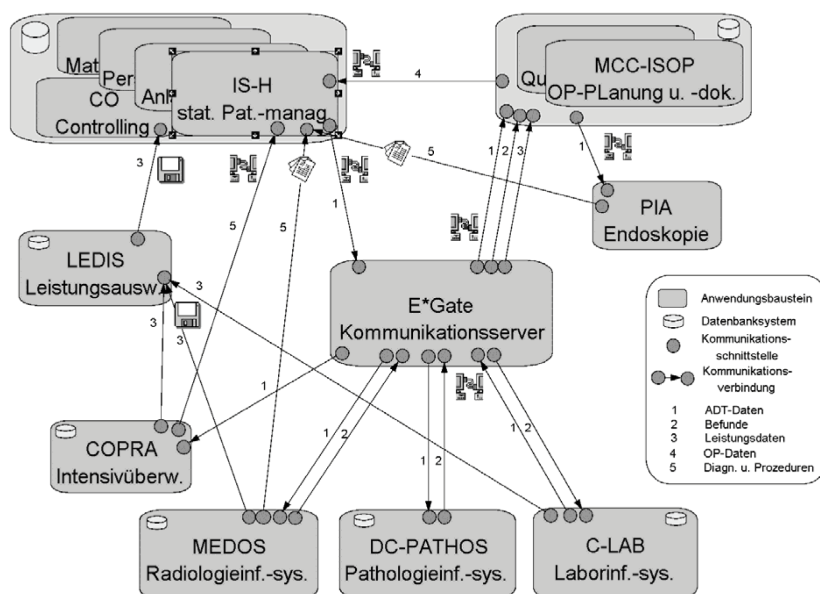

Abbildung 1.1.: Ausschnitt aus der elektronischen ADT- und Leistungsdatenkommunikation am Universitätsklinikum Leipzig [NHW+02]

1.1. Motivation

Krankenhausinformationssysteme (KIS) wie das des Universitätsklinikums Leipzig (UKL) bestehen oft aus vielen unterschiedlichen Anwendungssystemen, die auf Produkten verschiedener Hersteller basieren. Die verschiedenen Anwendungssysteme speichern gemeinsame Patientendaten. Für den Datenabgleich dieser Patientendaten werden im UKL unterschiedliche Techniken eingesetzt, wobei hauptsächlich, wie auch häufig in anderen Kliniken, ein nachrichtenorientierter Kommunikationsserver [LPH99] für den Datenaustausch verwendet wird. In Abbildung 1.1 ist ein Ausschnitt der Systemlandschaft des UKL mit Kommunikationswegen abgebildet. Weil die Anwendungssysteme durch die Kopplung nicht bzw. nicht zu stark in ihrer Autonomie eingeschränkt werden dürfen, um nicht die Patientenversorgung zu gefährden, handelt es sich um heterogene, autonome Informationssysteme.

Im Kooperationsprojekt EKKIS (Enge Kopplung klinischer Informationssysteme [NHW+02]) wurden in Zusammenarbeit des UKL mit dem OFFIS (Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und Systeme, http://www.offis.de) und der Meierhofer AG (http://www.meierhofer.de) Kopplungsstrategien für KIS-Komponenten entwickelt. Insbesondere wurde untersucht, wie die Anwendungssysteme des KIS enger gekoppelt werden können als über die eher lose Kopplung mittels eines nachrichtenorientierten Kommunikationsservers. Es sollte also neben der asynchronen Kommunikation über den Kommunikationsserver auch eine synchrone Kommunikation (siehe Abschnitt 2.1.2) möglich sein. Eine derartige Kombination von synchroner und asynchroner Kommunikation zum Zwecke des Datenabgleichs ist in Abbildung 3.4 auf Seite 50 dargestellt.

Das theoretische Konzept für den Datenabgleich bilden Replikationsstrategien, die ausführlich im Kapitel 3 vorgestellt werden. Einerseits sorgen Replikationsstrategien für die Konsistenz der Replikate (Kopien), worunter hier zunächst vereinfacht verstanden werden soll, dass alle Replikate eines logischen Objekts den gleichen Wert speichern, andererseits sollen, wie oben erwähnt, die Verfügbarkeit und die Performance gesteigert werden. Diese Ziele der Replikation sind gegenläufig bzw. konkurrierend, d.h. wenn z.B. die Konsistenz erhöht wird, dann müssen möglichst viele Replikate synchron geändert werden, was im Allgemeinen zu Lasten der Verfügbarkeit oder Performance geschieht.

Demnach soll hier vereinfacht unter Wahrung der Autonomie verstanden werden, dass die lokalen Informationssysteme in ihrer Verfügbarkeit und Performance nicht bzw. nicht zu stark durch die Replikationsstrategie eingeschränkt werden. Wie oben erwähnt, gilt dies besonders für klinische Informationssysteme. Auf der anderen Seite müssen Patientendaten natürlich möglichst konsistent sein, um die Behandlung der Patienten auf Basis korrekter Daten durchführen zu können. Um diesen Konflikt hinsichtlich Verfügbarkeit, Performance und Konsistenz zu lösen, wird eine flexible Replikationsstrategie benötigt, die z.B. hohen Wert auf Konsistenz legt, aber bei Nicht-Verfügbarkeit einzelner Systeme ein lokales Weiterarbeiten ermöglicht.

Ein weiterer Aspekt ist es, dass bei Patientendaten zwischen wichtigen und weniger wichtigen Daten unterschieden werden kann. Aus Sicht der medizinischen Versorgung sind z.B. Diagnosedaten eines Patienten weitaus wichtiger als Adressdaten. Durch diesen Sachverhalt, der durchaus auf andere Anwendungsdomänen übertragbar ist, folgt, dass ein unterschiedliches Verhalten der Replikationsstrategie im Hinblick der zu replizierenden Daten selbst gewünscht ist. Somit ist eine Replikationsstrategie erforderlich, die die Koordination der Schreib- und Lesezugriffe auf Basis des aktuellen Systemzustands und der zu replizierenden Daten vornimmt.

Während in der Forschung zunächst Replikationsstrategien entwickelt wurden, bei denen die Art der Koordination von Schreib- und Lesezugriffen fest vorgegeben war, werden in der jüngeren Vergangenheit hauptsächlich adaptive Replikationsstrategien vorgestellt, die ihr Verhalten zur Laufzeit anpassen. Häufig werden dabei Erweiterungen und/oder Kombinationen von traditionellen Replikationsstrategien vorgenommen (siehe verwandte Arbeiten in Abschnitt 3.4). Grundsätzlich wird diese Idee bei der in dieser Dissertation vorgestellten Replikationsstrategie RegRess aufgegriffen, wobei eine möglichst große Flexibilität dadurch erreicht wird, dass bei jedem Zugriff die Menge der betroffenen Replikate ermittelt wird.

Ob nun die Gewichtung bei einer Änderungspropagierung auf Konsistenz oder auf Autonomie (Verfügbarkeit/Performance) liegt, soll nicht fest vorgegeben werden, sondern durch die Fähigkeiten und Eigenschaften der beteiligten Systeme zu einem bestimmten Zeitpunkt ermittelt und festgelegt werden. Der Grundgedanke der Replikationsstrategie RegRess ist es, die betroffenen Replikate bei einer Änderung möglichst konsistent zu aktualisieren. Die Replikate, die jedoch die Autonomie spezieller Systeme gefährden bzw. zu stark einschränken, werden zeitversetzt aktualisiert, d.h. hier werden Inkonsistenzen temporär in Kauf genommen.

Damit möglichst situationsabhängig die Menge der betroffenen Replikate bei einem Zugriff bestimmt werden kann, war die Idee, die Bestimmung auf Basis von Regeln durchzuführen. In den Regeln sollen Kennzahlen der beteiligten Informationssysteme wie z.B. Konsistenz spezieller Daten, Performance der beteiligten Informationssysteme, etc. berücksichtigt werden, um eine möglichst optimale Koordination der Zugriffe zu erreichen. Daher handelt es sich bei RegRess um eine regelbasierte Replikationsstrategie. Die Verwendung von Regeln bietet den Vorteil, dass die Anwendungslogik von der *„Replikationslogik“*, also der Art der Koordination von Zugriffen, getrennt ist. Der Nachteil liegt darin, dass bei jedem Zugriff eine Inferenz auf den Regeln durchgeführt werden muss.

1.2. Problemstellung und Zielsetzung

Replikation in einer heterogenen, autonomen Systemlandschaft bedeutet, dass neben einer geeigneten Replikationsstrategie auch technische Aspekte wie z.B. die Kopplung der beteiligten Informationssysteme beachtet werden muss. In diesem Zusammenhang wird von Integration der Informationssysteme gesprochen. Zur systematischen Behandlung dieser Thematik wurden verschiedene Modelle geschaffen. In [Has00] wird die Integration z.B. an Hand eines 3-Ebenen Modells diskutiert, das in Abbildung 1.2 dargestellt ist. Die Organisationseinheiten werden in dem Modell durch drei Architekturebenen beschrieben: Geschäftsarchitektur, Anwendungsarchitektur und technologische Architektur.

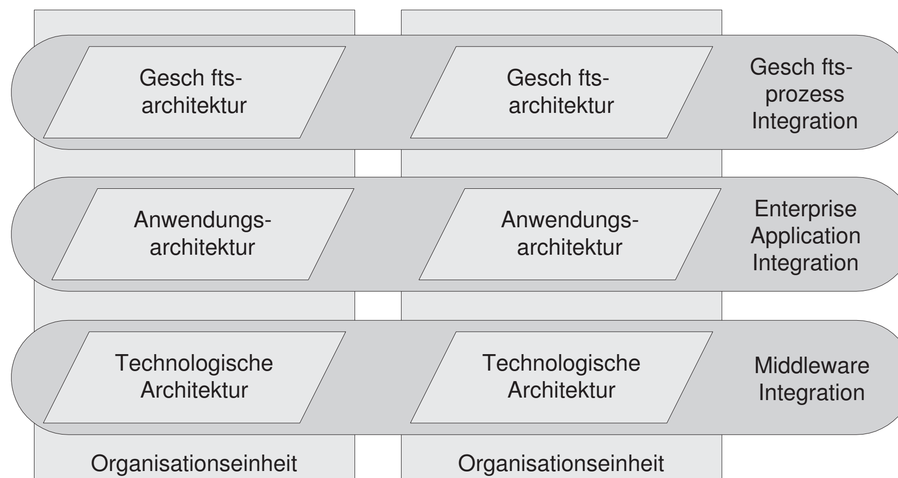

Abbildung 1.2.: Horizontale Integration zur Unterstützung der Geschäftsprozesse [Has00]

Eine möglichst effektive Unterstützung der übergreifenden Geschäftsprozesse erfordert nach diesem Modell eine horizontale Integration auf den verschiedenen Ebenen. Auf der obersten Ebene erfolgt die Geschäftsprozessintegration durch eine Geschäftsprozessanalyse. Die Integration auf der Anwendungsebene ist auch als EAI (Enterprise Application Integration, [Lin99]) bekannt. Auf der technologischen Ebene setzt Middleware [BN96] dabei die Integration technisch um, z.B. durch CORBA Object Request Broker [COR02]. Hinsichtlich dieser Ebenen können idealerweise für die Replikation folgende Arbeitsbereiche identifiziert werden:

1.	Geschäftsprozessintegration	→	Identifikation der betroffenen Daten
2.	EAI	→	Schemaintegration
3.	EAI	→	Replikationsstrategie
4.	Middleware	→	Transaktionskonzept
5.	Middleware	→	Technische Anbindung der Systeme

In den ersten beiden Arbeitsbereichen, die in dieser Dissertation nicht behandelt werden, werden eher semantische Aspekte geklärt, zum Thema Geschäftsprozesse siehe z.B. [Sch98] bzw. für Schemaintegration z.B. [Con02]. Der Fokus in der vorliegenden Arbeit wird auf die eher technologischen Arbeitsbereiche 3. bis 5. gelegt, wobei die Replikationsstrategie den Schwerpunkt bildet. Wie in Abschnitt 1.1 dargestellt, soll durch die Replikationsstrategie die Replikation von Daten für den Anwendungsbereich heterogener, autonomer Informationssysteme ermöglicht werden, d.h. bei weitestgehender Wahrung der Autonomie sollen Änderungen möglichst konsistent durchgeführt werden.

Zwar wurde im Kontext dieser Dissertation ein Transaktionsmanager für so genannte High-Level-Services entwickelt (siehe Abschnitt 6.3.2) und in verschiedenen Prototypen die Anbindung von Informationssystemen mittels der Java Platform Enterprise Edition (siehe Abschnitt 7.2) untersucht, trotzdem wird in dieser Dissertation hauptsächlich die Entwicklung der Replikationsstrategie RegRess betrachtet. Das Transaktionskonzept und die technische Anbindung wird nur insoweit thematisiert, wie es für RegRess benötigt wird. Dabei wird davon ausgegangen, dass die beteiligten Informationssysteme bekannt und „stationär“ sind, d.h. es werden keine mobilen Systeme oder Peer-To-Peer-Systeme betrachtet.

Damit ist das Hauptziel dieser Dissertation die Entwicklung der regelbasierten Replikationsstrategie RegRess. Damit einher geht die Entwicklung einer Regelsprache, die es erlaubt, Replikationsregeln zu formulieren, sowie die Implementierung eines Replikationsmanagers, der RegRess umsetzt. Im einzelnen sind folgende Ziele zu nennen:

1. Konzeption der regelbasierten Replikationsstrategie RegRess: Die Protokolle von RegRess (**Reg**elbasierte **Re**plikation**ss**trategie) für die Koordination von Schreib- und Lesezugriffen werden spezifiziert und das Konsistenzverhalten von RegRess wird festgelegt. Dazu erfolgt die Definition des Konsistenzbegriffes *„Konsistenzgrad"*.
2. Entwicklung der Regelsprache RRML: Die Syntax und die Semantik der Regelsprache RRML (**R**eplication **R**ule **M**arkup **L**anguage), die die Formulierung von Replikationsregeln erlaubt, wird vereinbart. Für eine Regelmenge wird eine XML-Repräsentation angegeben, um die Regeln persistieren und austauschen zu können.
3. Entwurf des Replikationsmanagers KARMA: Es wird eine Softwarearchitektur für den KARMA (**K**onfigurierbarer, **a**daptiver **R**eplikations**ma**nager) entworfen, der die Replikationsstrategie RegRess implementiert. Hierbei wird eine Spezifikation der einzelnen Komponenten und der Transaktionsverarbeitung vorgenommen.
4. Simulation der Replikationsstrategie RegRess: Zur Evaluation wird die Replikationsstrategie RegRess bzw. der Replikationsmanager KARMA in ein Simulationsframework eingebunden. Hierdurch ist die Analyse von Kennzahlen möglich, die das Verhalten von RegRess bei Simulationsläufen beschreibt.

Angemerkt sei, dass das Simulationsframework F4SR (siehe Abschnitt 7.3.1) im Rahmen dieser Dissertation durch eine studentische Arbeit entwickelt wurde, um Replikationsstrategien mittels Simulation zu analysieren. Der F4SR bietet einen Plugin-Mechanismus, der das Einbinden von Replikationsstrategien erlaubt.

1.3. Aufbau der Arbeit

Diese Arbeit gliedert sich in vier Teile plus dem Anhang. Nachfolgend werden diese Teile mit den darin enthaltenen Kapiteln kurz vorgestellt:

1. Teil: Grundlagen und verwandte Arbeiten
Im ersten Teil werden die theoretischen Grundlagen vorgestellt, die zum Verständnis dieser Dissertation nötig sind. Neben einer allgemeinen Einführung, die sowohl für replizierte als auch für nicht-replizierte Datenbanken relevant ist, folgt der Schwerpunkt mit dem Thema Datenreplikation. Weil hier u.a. Replikationsstrategien vorgestellt werden, wird hier auf verwandte Arbeiten eingegangen. Der erste Teil besteht aus den folgenden Kapiteln:

2. Kapitel: Grundlegende Konzepte
In den grundlegenden Konzepten werden zunächst einige Begriffe erläutert, die allgemein für Datenbanken gelten. Anschließend werden Transaktionen betrachtet, wobei insbesondere die für diese Arbeit benötigten, so genannten weiterführenden Transaktionskonzepte diskutiert werden. Abschließend werden allgemeine Konsistenzbegriffe vorgestellt.

3. Kapitel: Datenreplikation und verwandte Arbeiten Dieses Kapitel bildet den Hauptteil der Grundlagen. Es beinhaltet die benötigten Begriffe und Definitionen, um die in dieser Arbeit verwendete Terminologie festzulegen. Dabei werden neben allgemeinen Definitionen vor allem verschiedene Definitionen für Konsistenz in replizierten Datenbanken aus der Literatur erläutert. Des Weiteren werden verschiedene Replikationsstrategien vorgestellt, wobei die adaptiven Replikationsstrategien, die ihr Verhalten zur Laufzeit dem Systemzustand anpassen, im Wesentlichen die verwandten Arbeiten bilden.

2. Teil: Regelbasierte Replikationsstrategie
Das in dieser Arbeit entwickelte Konzept einer regelbasierten Replikationsstrategie wird im zweiten Teil dargestellt. Neben den Protokollen für die Koordination von Schreib- und Lesezugriffen sowie der Spezifikation des Konsistenzverhaltens wird in den folgenden Kapiteln die Regelsprache RRML dargeboten:

4. Kapitel: Regelbasierte Replikationsstrategie RegRess
Zunächst werden Annahmen getroffen, die ohne Beschränkung der Allgemeinheit die Konzeption der regelbasierten Replikationsstrategie RegRess erleichtert. Die Koordination von Schreib- und Lesezugriffen wird durch Protokolle für die Verarbeitung der Zugriffe definiert. Darauf folgt die Festlegung des Konsistenzverhaltens von RegRess durch eine Anforderungsanalyse für fachliche und technische Konsistenzbedingungen. Abschließend wird gezeigt, wie ein Trade-Off hinsichtlich der Replikationsziele Verfügbarkeit, Performance und Konsistenz erreicht werden kann, wodurch die Auswahl geeigneter Regeln unterstützt wird.

5. Kapitel: Regelsprache RRML
Der Schwerpunkt dieses Kapitels ist der konzeptionelle Entwurf der Regelsprache RRML, d.h. die Syntax und die Semantik der Replikationsregeln wird spezifiziert. Dabei wird ggf. das benötigte Verhalten eines Regelinterpreters festgelegt. Abschließend wird kurz die XML-Repräsentation der Regeln gezeigt und eine Analyse der RRML vorgenommen.

3. Teil: Software-Entwurf und Evaluation
Der dritte Teil zeigt zunächst einen exemplarischen Software-Entwurf eines Replikationsmanagers, der die regelbasierte Replikationsstrategie implementiert, sowie verschiedene experimentelle Prototypen, die im Kontext dieser Dissertation entstanden sind. Diese Themen sind auf folgende Kapitel verteilt:

6. Kapitel: Replikationsmanager KARMA
Neben der Softwarearchitektur des Replikationsmanagers KARMA werden dessen Komponenten beschrieben und die Funktionsweise aufgezeigt. Weiterhin wird das benötigte Transaktionskonzept erläutert.

7. Kapitel: Prototypische Implementierung und Evaluation
In diesem Kapitel wird neben den Prototypen, die durch studentische Arbeiten erstellt wurden, der Prototyp KARMA vorgestellt. Des Weiteren wird das KARMA-Plugin für den Simulator F4SR erklärt, womit der F4SR die Replikationsstrategie RegRess simulieren kann. Die Evaluation erfolgt abschließend auf Basis von Simulationsexperimenten mit dem F4SR.

4. Teil: Zusammenfassung und Ausblick
Die Arbeit schließt mit dem vierten Teil, der folgende Kapitel beinhaltet:

8. Kapitel: Zusammenfassung
Dieses Kapitel fasst die Ergebnisse der Dissertation zusammen.

9. Kapitel: Ausblick
Im Ausblick werden mögliche Erweiterungen und zukünftige Arbeiten beschrieben.

Anhang
Der Anhang enthält neben den Verzeichnissen einen kurzen Einstieg zu regelbasierten Systemen. Weiterhin werden die Anforderungen an das Konsistenzverhalten von RegRess tabellarisch aufgelistet sowie das UML-Modell des KARMA-Prototypen demonstriert.

Teil I.

Grundlagen und verwandte Arbeiten

2. Grundlegende Konzepte

Bevor näher auf Replikation, d.h. beispielsweise auf Definitionen und Replikationsstrategien, eingegangen wird, folgen in diesem Kapitel einige grundlegende Konzepte, die im Wesentlichen die Basis für Anwendungen mit replizierten Daten darstellen. So bedeutet die Speicherung von Daten an verschiedenen Lokalisationen, dass mehrere Rechner involviert sind. Diese Rechner müssen miteinander interagieren, um ihre Daten abzugleichen. Daher ist ein Verständnis darüber, wie Rechner verteilt kooperieren und kommunizieren, unerlässlich.

Ein wichtiger Punkt bei verteilten Daten ist es, dass eine ordnungsgemäße Verarbeitung gewährleistet wird. Insbesondere im Mehrbenutzerbetrieb ist eine Koordination von gleichzeitig durchgeführten Zugriffen auf die Daten von großer Bedeutung, um unerwünschte Nebeneffekte bzw. Fehler zu vermeiden. Für einzelne Operationen bzw. Prozesse ist ein widerspruchsfreier Zugriff, d.h. ein konsistenter Zugriff, auf die Datenobjekte elementar.

Daher wird zunächst in Abschnitt 2.1 auf verteilte Systeme eingegangen, um anschließend im Abschnitt 2.2 eine ordnungsgemäße Verarbeitung mittels Transaktionen zu beschreiben. Zum Abschluss wird in Abschnitt 2.3 diskutiert, was unter Konsistenz und Korrektheit bei verteilten Systemen bzw. verteilten Daten zu verstehen ist.

2.1. Verteilte Systeme

Verteilte Systeme bieten gegenüber zentralisierten Systemen bzw. zentralen Rechnern verschiedene Vorteile, z.B. Wirtschaftlichkeit, Geschwindigkeit, inhärente Verteilung, Zuverlässigkeit und Skalierbarkeit [Web98]. Die Nachteile liegen z.B. im hohen Bedienungs- und Wartungsaufwand, in Sicherheitsproblemen und in der Komplexität des Kommunikationssystems. Häufig treten Probleme auf, weil Heterogenitäten überwunden werden müssen, insbesondere dann, wenn einzelne, „isolierte" Systeme gekoppelt werden, um Ressourcen gemeinsam zu nutzen. In diesem Zusammenhang wird auch von Integration von Informationssystemen gesprochen [Has00], was z.B. im Anwendungsbereich Krankenhausinformationssysteme eine große Herausforderung darstellt [GGH00].

Für tiefergehende Informationen zu verteilten Systemen sei auf die einschlägige Literatur verwiesen, z.B. [CDK00, TvS02]. An dieser Stelle werden nur die Punkte detaillierter erläutert, die im Hinblick auf die vorliegende Arbeit relevant sind:

- Definition, Transparenz und Zeitstempel (Abschnitt 2.1.1)
- Kommunikationsmodelle, Kommunikationsfehler und Middleware (Abschnitt 2.1.2)
- Klassifizierung (Abschnitt 2.1.3)

2.1.1. Definition, Transparenz und Zeitstempel

In [TvS02] wird ein verteiltes System wie folgt definiert: *„Ein verteiltes System ist eine Menge unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen."* Hierbei handelt es sich um eine mitunter schwer zu erfüllende Anforderung an das verteilte System, weil die einzelnen Rechner sich wie eine vollständige Einheit darstellen müssen. Dafür müssen im Grunde die wesentlichen Transparenzeigenschaften (siehe unten) vollständig erfüllt werden, was sich letztlich schwer realisieren lässt. Eine abgeschwächte Definition ist in [CDK00] angegeben: *„Ein verteiltes System ist ein System, in welchem Hardware- und Software-Komponenten, die auf im Netzwerk arbeitenden Computern lokalisiert sind, kommunizieren und*

ihre Aktivitäten nur durch Austausch von Nachrichten koordinieren.“ Es wird also darauf abgestellt, dass die beteiligten Rechner in irgendeiner Form interagieren, ohne jedoch eine zu strenge Forderung hinsichtlich der Art der Integration zu stellen. Diesem Verständnis von verteilten Systemen wird sich in dieser Arbeit angeschlossen, weil Replikationsstrategien für Systeme entwickelt werden, die so weit wie möglich ihre Eigenständigkeit bewahren sollen.

Zu dem allgemeinen Forschungsbereich „Verteilte Systeme“ [TR85, Web98, CDK00] haben sich spezielle Forschungsbereiche entwickelt, wie z.B. „Verteilte Betriebssysteme“ [Gal99], „Verteilter, gemeinsamer Speicher“ [PTM97] oder „Verteilte Datenbanken“ [Ozs99] bzw. „Verteilte Datenbankmanagementsysteme (VDBMS)“. Speziell der zuletzt genannte Forschungsbereich befasst sich u.a. mit Datenreplikation. In [Rah94] werden Mehrrechner-Datenbanksysteme in *„shared-everything“*, *„shared-disk“* und *„shared-nothing“* klassifiziert, wobei die Externspeicherzuordnung (gemeinsam oder partitioniert), die räumliche Verteilung (lokal oder ortsverteilt) und die Rechnerkopplung (eng, nahe oder lose) berücksichtigt werden. Nach [Rah94] sind verteilte Datenbanksysteme integrierte, geographisch verteilte Mehrrechner-Datenbanksysteme vom Typ *„shared-nothing“*, wobei *„shared-nothing“* wie folgt charakterisiert wird:

„Die Datenbank-Verarbeitung erfolgt durch mehrere, im Allgemeinen lose gekoppelte Rechner, auf denen jeweils ein DBMS abläuft. Die Externspeicher sind unter den Rechnern partitioniert, so dass jedes DBMS nur auf Daten der lokalen Partition direkt zugreifen kann. Die Rechner können lokal oder geographisch verteilt angeordnet sein. Multiprozessoren sind als Rechnerknoten möglich, d.h. die lokalen DBMS können vom Typ „shared-everything“ sein.“

Nach [TvS02] ist ein wichtiges Ziel von verteilten Systemen, die Verteilung der Ressourcen auf mehrere Rechner zu verbergen, wobei unterschiedliche Formen der Transparenz betrachtet werden können: Zugriff, Position, Migration, Relokation, Replikation, Nebenläufigkeit, Fehler und Persistenz. Für einen Benutzer soll z.B. transparent sein, wo sich eine Ressource befindet (Positionstransparenz bzw. Lokalisationstransparenz). Je vollständiger diese Transparenzeigenschaften von dem verteilten System erfüllt werden, um so mehr erscheint es nach außen als ein zentrales System.

Für verteilte Datenbanken wurden in [Dat03] 12 Regeln angegeben, deren Erfüllung die Verteilung der Datenbank vor dem Benutzer versteckt: Lokale Autonomie, keine Abhängigkeit von einer zentralen Stelle, höhere Verfügbarkeit und Zuverlässigkeit, Ortstransparenz, Fragmentierungsunabhängigkeit, Replikationstransparenz, verteilte Anfragebearbeitung, verteiltes Transaktionsmanagement, Hardware-, Betriebssystem-, Netzwerk- und Datenbankmanagementsystem-Unabhängigkeit. Diese 12 Regeln lassen sich jedoch nicht immer vollständig erfüllen, weil sie zumindest teilweise in Widerspruch zueinander stehen (siehe Abschnitt 3.1).

In verteilten Systemen spielt Zeit eine bedeutende Rolle, um Ereignissen, wie z.B. eine Datenänderung, eine exakte (Uhr-)Zeit, einen so genannten *„Zeitstempel“*, zuordnen zu können oder um Ereignisse bzw. Prozesse zeitlich sortieren zu können und damit eine Synchronisation zu ermöglichen. Während in einem zentralen System auf eine einzige Uhr Bezug genommen werden kann, muss in einem verteilten System für alle Rechner eine gemeinsame Zeitbasis ermittelt werden. Somit müssen in verteilten Systemen die Uhren der einzelnen Rechner selbst synchronisiert werden [SWL90, FL04], um zeitabhängige Aufgaben zu bewältigen. Beispiele für die Verwendung von synchronisierten Uhren sind in [Lis91] zu finden, u.a. die Realisierung der Fehlersemantik At-Most-Once (siehe Abschnitt 2.1.2). Auch bei Replikationsstrategien (siehe Abschnitt 3.3) werden Zeitstempel verwendet, um ein aktuelles Replikat zu bestimmen.

Eine exakte Synchronisation mehrerer Rechneruhren auf eine absolute physikalische Realzeit ist nicht möglich [CDK00], sondern je nach Aktualisierungsintervall kann bestenfalls garantiert werden, dass die Rechneruhren sich innerhalb eines bestimmten Bereichs bewegen. Die Synchronisation von Uhren kann unterteilt werden in externe Synchronisation, bei der auf eine externe Referenzzeit synchronisiert wird, und in interne Synchronisation, bei der ausschließlich die internen Uhren der Rechner verwendet werden. Eine physikalische Referenzzeit ist UTC (Universal Coordinated Time, siehe z.B. [TvS02]), die per Kurzwelle bzw. per Satellit ausgestrahlt wird.

Beispiele für externe Synchronisation, die UTC verwenden, sind die Methode von Cristians [Cri89] und das Network Time Protocol (NTP, [Mil03]). Der Berkeley Algorithmus [GZ89] ist beispielsweise der internen Synchronisation zuzuordnen.

Wenn nur die zeitliche Sortierung von Ereignissen benötigt wird, dann ist keine physikalische Realzeit erforderlich. In diesem Fall kann eine so genannte logische Uhr verwendet werden, mittels derer eine logische Reihenfolge von Ereignissen bestimmt werden kann. Logische Uhren werden im Allgemeinen als Zähler realisiert, die bei bestimmten Ereignissen inkrementiert werden. Die Synchronisation von logischen Uhren wird in [Lam78] durch eine „passiert-vor"-Relation realisiert, wobei entweder Ereignisse innerhalb eines Prozesses oder Sendeereignis und Empfangsereignis unterschiedlicher Prozesse in Beziehung zueinander gesetzt werden. Eine Erweiterung hierzu sind *„Vektor-Zeituhren"* (auch *„Versionsvektoren"*, [PPR83, Mat89]), über die weitere Kausalitätsbeziehungen erkennbar sind.

2.1.2. Kommunikationsmodelle, Kommunikationsfehler und Middleware

Damit die Rechner eines verteilten Systems untereinander Nachrichten austauschen können, müssen sie durch ein Rechnernetzwerk (oder einfach Netzwerk) miteinander verbunden sein. Die Vernetzung der einzelnen Rechner, Netzwerk-Topologien, Protokolle und Technologien sind beispielsweise in [Tan96] beschrieben. An dieser Stelle wird nicht detailliert auf die Netzwerktechnik eingegangen, sondern es sollen verschiedene Kommunikationsmodelle erläutert werden. In [Web98] werden die unterschiedlichen Formen anhand folgernder Kriterien klassifiziert: Adressierung (direkt oder indirekt), Blockierung (synchron oder asynchron), Pufferung (ungepuffert oder gepuffert) und Kommunikationsform (meldungsorientiert oder auftragsorientiert).

Im Hinblick auf die Klassifizierung von Replikationsstrategien (siehe Abschnitt 3.2.4) bzw. auf die zur Datenreplikation gehörende Transaktionsverarbeitung (siehe Abschnitt 2.2) ist die Blockierung von besonderem Interesse: Bei blockierender Kommunikation, sprich synchroner Kommunikation, ist der Sender einer Nachricht (Anforderung, request) solange blockiert, bis eine Antwort (reply) vom Empfänger gegeben wird. Bei nicht-blockierender Kommunikation, sprich asynchroner Kommunikation, kann der Sender dann weiter arbeiten, wenn die Nachricht vom Transportsystem entgegengenommen ist. Der Sender weiß nicht, wann die Nachricht zugestellt wird und wann ggf. die Antwort abgeholt werden kann. Durch eine synchrone Kommunikation ist ohne weitere Hilfe eine Synchronisation von Prozessen möglich. Dieser Vorteil ist jedoch mit dem Nachteil verbunden, dass möglicherweise ein Prozess auf Dauer blockiert wird, falls ein Teilnehmer an der Kommunikation ausfällt. Bei der asynchronen Kommunikation erfolgt eine zeitliche Entkopplung, wodurch Parallelarbeit ermöglicht wird bzw. ein Ausfall eines Teilnehmers überbrückt werden kann. Auf Synchronisation wird im Allgemeinen bei der asynchronen Kommunikation verzichtet. Weiterhin wird in diesem Fall eine persistente Pufferung benötigt, um eine zuverlässige Nachrichtenzustellung zu gewährleisten.

Eine wichtige Eigenschaft von verteilten Systemen ist die Zuverlässigkeit, mit der die einzelnen Rechner untereinander kommunizieren. In [HT94] werden Kommunikationsfehler klassifiziert, wobei insbesondere eine Unterscheidung zwischen Fehlern von Prozessen (Dienst-, Rechnerausfall) und von Kommunikationskanälen (Nachrichtenverlust) vorgenommen wird. Falls Fehler auf Kommunikationskanälen vorliegen, kann es zu Netzwerkpartitionierung (oder Netzpartitionierung) kommen, die nach [Rah94] wie folgt definiert ist: *„Bei Netzwerkpartitionierung entstehen auf Grund von Fehlern im Kommunikationssystem disjunkte Teilnetze oder Partitionen, so dass Rechner verschiedener Partitionen nicht mehr miteinander kommunizieren können."* Im Allgemeinen kann jedoch nicht festgestellt werden, ob der Grund eines Kommunikationsfehlers eine Netzpartitionierung oder ein Rechnerausfall ist [Dav84].

Zur Beschreibung des Verhaltens im Fehlerfall wurden die so genannten Fehlersemantiken für Kommunikationsmodelle entwickelt [Spe82, TR85]. Es werden vier verschiedene Klassen unterschieden:

1. Maybe: Ein Auftrag wird entweder gar nicht oder höchstens einmal durchgeführt, d.h. Fehler werden nicht behandelt.
2. At-Least-Once: Ein Auftrag wird mindestens einmal durchgeführt. Bei Nachrichtenverlusten wird der Auftrag bis zum Erfolg wiederholt. Bei einem Rechnerausfall wird ggf. gewartet, bis der Rechner wieder zur Verfügung steht oder es wird mit einem anderen Rechner verbunden. Da Auftragsduplikate nicht erkannt werden, kann ein Auftrag mehrfach ausgeführt werden.
3. At-Most-Once: Ein Auftrag wird höchstens einmal ausgeführt. Bei Nachrichtenverlusten wird der Auftrag wiederholt, wobei Auftragsduplikate vom Empfänger erkannt werden. Bei Rechnerausfall wird keine Antwort erwartet, d.h. es wird ein Fehler an den Sender gemeldet, der für eine Behandlung des nicht ausgeführten Auftrags sorgen muss.
4. Exactly-Once: Ein Auftrag wird genau einmal durchgeführt, d.h. der Absturz und Wiederanlauf von Rechnern bzw. Diensten wird eingeschlossen. Nach [Web98] kann diese Fehlersemantik durch persistente Datenhaltung und verteilte Transaktionsmechanismen auf ein At-Most-Once-Protokoll implementiert werden, ist aber nach [TvS02] im allgemeinen Fall nicht realisierbar.

Die meisten RPC-Implementierungen (remote procedure call, entfernter Prozeduraufruf [BN84]) bzw. hierauf basierende Erweiterungen verwenden die Fehlersemantik At-Most-Once. Bei der Datenreplikation muss für die erfolgreiche Propagierung einer Änderung, also einer Aktualisierung aller Kopien, die Fehlersemantik Exactly-Once implementiert werden [FG01], d.h. es wird eine geeignete Transaktionsverarbeitung benötigt (siehe Abschnitt 2.2).

Ein Dienst bzw. eine Menge von Diensten, die eine Kommunikation in einem verteilten System erlaubt, wird als Middleware bezeichnet [BN96], wobei je nach Definition die Dienste bestimmte Eigenschaften aufweisen müssen. Der entfernte Prozeduraufruf bzw. darauf basierende Erweiterungen wie z.B. der entfernte Objektmethodenaufruf (remote method invocation, [Gro01]) ist Middleware, die normalerweise für eine synchrone Kommunikation verwendet wird. Für die asynchrone Kommunikation werden im Allgemeinen Nachrichtendienste genutzt, man spricht von Message-oriented Middleware (MOM, [BCSS99]), wobei häufig Warteschlangen (queues) für die Pufferung genutzt werden. Nach [Men05] haben sowohl RPC als auch MOM je nach Anwendungsszenario Performancevorteile. Ein Server, der neben der Grundfunktionalität eines Nachrichtendienstes weitere Funktionen bereitstellt, um z.B. Heterogenitäten wie unterschiedliche Schemata zu überwinden, wird Kommunikationsserver genannt. Kommunikationsserver werden z.B. in klinischen Informationssystemen eingesetzt [LPH99], um zwischen den einzelnen Systemen insbesondere Patientendaten auszutauschen.

2.1.3. Klassifizierung

Verteilte Systeme können nach den verschiedensten Kategorien klassifiziert werden, z.B. Verteilungsmodell (welche Ressourcen werden wie verteilt, siehe z.B. [Neu94]), Heterogenität [Con97], Kommunikationsmodell (siehe Abschnitt 2.1.2), Netzwerktopologien [Mul93], Sicherheitsmodell [And01, Bör03], Hardware- bzw. Software-Architektur [VR01, DGH03] etc. Häufig werden für bestimmte Gesichtspunkte nur ausgewählte Dimensionen betrachtet. So sind für die Sicht auf die physikalische Verbindung der einzelnen Rechner in erster Linie die Netzwerktopologie und die Netzwerkprotokolle von Interesse.

Hinsichtlich der Datenreplikation sollen an dieser Stelle zwei Aspekte betrachtet werden: Einerseits die Heterogenität der beteiligten Systeme und andererseits die *„Kommunikationsfähigkeit“* bestimmter Typen von Systemen. Verteilte Systeme können in homogene und heterogene Systeme unterteilt werden: Homogene Systeme sind gleichartig aufgebaute Systeme, während sich heterogene Systeme beispielsweise in der Hardware-Plattform, dem Betriebssystem, dem Datenbankmanagementsystem und/oder den Datenschemata unterscheiden können. In homogenen Systemen kann die Umsetzung einer Replikationsstrategie im Allgemeinen vom Datenbankmanagementsystem selbst übernommen werden [HLUA02], während in heterogenen Systemen die

Implementierung einer Replikationsstrategie höhere Ansprüche stellt, weil die Heterogenitäten überwunden werden müssen. Beispielsweise ist bei vorliegen unterschiedlicher Datenschemata eine Schemaintegration erforderlich [Con02].

Mit *„Kommunikationsfähigkeit“* sind hier Charakteristika spezieller Rechner bzw. Systeme gemeint, nämlich von mobilen Systemen [PS97] und von Peer-To-Peer-Systemen (P2P-Systemen, [Sch01a, SW04a]). Die Besonderheit bei mobilen Systemen liegt darin, dass diese Systeme häufig keine Verbindung zu anderen Rechnern haben, obwohl im eigentlichen Sinn kein Kommunikationsfehler vorliegt. Auf Grund der Tatsache, dass mobile Systeme des Öfteren ihre Lokalisation wechseln, besteht mitunter keine physikalische Verbindung zu anderen Kommunikationspartnern. Bei P2P-Systemen ist insbesondere der Verzeichnisdienst (Look Up Service, [BKK+03]) von Bedeutung, um das gegenseitige Finden der durch die Peers (gleichberechtigte Rechner) bereitgestellten Daten zu ermöglichen. Bei P2P-Systemen ist es auf Grund der Suche von Kommunikationspartnern über den Verzeichnisdienst nicht von vornherein festgelegt, wer die Kommunikationspartner sind bzw. ob ein ehemaliger Kommunikationspartner noch erreicht wird. Sowohl bei mobilen Systemen als auch bei P2P-Systemen werden im Allgemeinen wegen ihrer besonderen Charakteristika spezielle Replikationsstrategien benötigt (siehe Abschnitt 3.3.2).

Im Titel dieser Arbeit „Regelbasierte Replikationsstrategien für heterogene, autonome Informationssysteme“ wird durch Informationssysteme ausgedrückt, dass es sich um Systeme handelt, die insbesondere Daten verwalten. Im Wesentlichen handelt es sich um verteilte Mehrrechner-Datenbanksysteme vom Typ „shared-nothing“ (siehe Abschnitt 2.1.1). Diese Systeme können heterogen sein, wobei auch mobile Systeme eingeschlossen sind. P2P-Systeme werden in dieser Arbeit nicht betrachtet. Durch das Adjektiv „autonom“ wird zusätzlich darauf hingewiesen, dass die einzelnen Rechner bzw. Informationssysteme durch eine Replikationsstrategie nicht zu stark eingeschränkt werden dürfen, d.h. die lokalen Anwendungen sollen so wenig wie möglich durch die Replikationsstrategie beeinflusst werden.

2.2. Transaktionen

Sowohl in zentralen als auch in verteilten Datenbanken oder Anwendungen ist eine ordnungsgemäße Verarbeitung unentbehrlich. Dies beinhaltet, dass ein einzelner Benutzer sich auf seine durchgeführten Operationen verlassen kann und dass auch im Mehrbenutzerbetrieb keine unakzeptablen Nebeneffekte oder Fehler auftreten. Zur Vermeidung dieser Mängel wird auf die so genannte Transaktionsverarbeitung gesetzt, die im Folgenden kurz dargestellt wird.

Die Bedeutung der Transaktionsverarbeitung zeigt sich in der Vergangenheit darin, dass eine Vielzahl Veröffentlichungen zu diesem Thema publiziert wurden. Umfassende Überblicke und Zusammenfassungen sind z. B. in [BGMS92, GR93, BN97, WV01] zu finden. Folgende, für diese Arbeit relevante Punkte sollen genauer untersucht werden:

- Definition, ACID-Eigenschaften und elementare Operationen (Abschnitt 2.2.1)
- Commit-Protokolle und XA-Protokoll (Abschnitt 2.2.2)
- Synchronisation von Nebenläufigkeit und Serialisierbarkeit (Abschnitt 2.2.3)
- Abgeschwächte Serialisierbarkeitskriterien (Abschnitt 2.2.4)
- Weiterführende Transaktionskonzepte: Sagas und Queued Transactions (Abschnitt 2.2.5)

2.2.1. Definition, ACID-Eigenschaften und elementare Operationen

In der Literatur sind verschiedene Definitionen vom Begriff Transaktion zu finden, die häufig je nach Anwendungsgebiet variieren, aber im Allgemeinen sinngemäß gleiche Aussagen beinhalten. Eine allgemeine Definition aus einem Lexikon [Klu01] lautet: *„Allgemein der kleinste, unteilbare und daher an einem Stück ununterbrochen abzuarbeitende Prozess einer Anwendung. Transaktionen werden daher vollständig oder gar nicht abgearbeitet.“* Im Datenbankbereich hat sich eine Referenzdefinition etabliert, die in [SH99] wie folgt wiedergegeben ist: *„Eine Transaktion ist*

eine Folge von Operationen (Aktionen), die die Datenbank von einem konsistenten Zustand in einen konsistenten, eventuell veränderten, Zustand überführt, wobei das ACID-Prinzip eingehalten werden muss."

Ein wichtiges Merkmal einer Transaktion ist also, dass das ACID-Prinzip bzw. die so genannten ACID-Eigenschaften (**A**tomicity, **C**onsistency, **I**solation, **D**urability, nach [HR83], basierend auf [Gra78, Gra81]) eingehalten werden:

Atomicity (Atomarität): Eine Transaktion wird ganz oder gar nicht ausgeführt, d.h. alle Operationen einer Transaktion werden als logische Einheit gesehen. Insbesondere bei Abbruch der Transaktion werden keine Zwischenergebnisse gespeichert, sondern der Zustand zu Beginn der Transaktion wird wieder hergestellt.

Consistency (Konsistenz): Eine Transaktion, die ihr normales Ende erreicht, bewahrt die Konsistenz der Datenbank, d.h. insbesondere, dass nach Ende einer Transaktion sämtliche physischen und logischen Integritätsbedingungen erfüllt sind (siehe Abschnitt 2.3).

Isolation (Isolation): Durch die Transaktion bzw. durch das Transaktionssystem wird ein „logischer Einbenutzerbetrieb" geboten, d.h. parallele Datenbankzugriffe durch nebenläufige Prozesse bleiben unsichtbar. Hiermit eingeschlossen ist, dass keine unerwünschten Nebenwirkungen auftreten, auch dann nicht, wenn eine Transaktion zurückgesetzt werden muss.

Durability (Dauerhaftigkeit): Nach dem erfolgreichen Ende einer Transaktion sind die Ergebnisse dieser Transaktion dauerhaft, d.h. die Änderungen überstehen Fehler wie Rechnerausfall, Speicherfehler, Kommunikationsfehler etc.

Die Einhaltung der ACID-Eigenschaften wird auch als *„das Transaktionskonzept"* [Rah93] bezeichnet. Eine *„ACID-Transaktionen"* ist eine Transaktion, die die ACID-Eigenschaften einhält. Da neben diesen so genannten *„flachen Transaktionen"* weiterführende Transaktionsmodelle bzw. -konzepte insbesondere für den verteilten Fall benötigt wurden und werden, sind unter anderem die *„geschachtelten Transaktionen"* entwickelt worden (siehe Abschnitt 2.2.5), wobei mitunter die ACID-Eigenschaften nur in abgeschwächter Form umgesetzt werden. Beispielsweise ist bei *„langlebigen Transaktionen"*, also Transaktionen mit langer Ausführungszeit, ein Zurücksetzen im Allgemeinen sehr aufwändig, so dass die Eigenschaft Atomarität nur auf eine Teilfolge der Operationen (*„Teiltransaktion"*) angewandt wird.

Die Gewährleistung der ACID-Eigenschaften, auch in abgeschwächter Form, wird vom so genannten *„Transaktionssystem"* übernommen. Die Eigenschaften Atomarität und Dauerhaftigkeit werden dabei von Freigabe- bzw. Commit-Protokollen (siehe Abschnitt 2.2.2) und die Eigenschaft Isolation wird im Mehrbenutzerbetrieb durch Synchronisation erreicht (siehe Abschnitt 2.2.3). Die Überwachung der Konsistenzeigenschaft ist schwer überprüfbar. Falls so genannte Konsistenz- oder Integritätsbedingungen (siehe Abschnitt 2.3.2) vorliegen, kann deren Einhaltung überprüft werden. Andernfalls muss das Transaktionssystem davon ausgehen, dass erfolgreiche Transaktionen die Datenbank wiederum in einen konsistenten Zustand überführen (siehe Abschnitt 2.3).

Damit ein Anwendungsprogramm transaktionsorientiert arbeiten kann, muss es dem Transaktionssystem zumindest den Anfang und das Ende einer Transaktion mitteilen. Da aus Sicht des Anwendungsprogramms eine Transaktion eine Einheit darstellt, die entweder komplett wirksam wird oder gar nicht (ein Abbruch der Transaktion kann auf Grund eines Systemfehlers oder auf Grund der Anwendungslogik erfolgen), muss es für das Ende einer Transaktion zwei Operationen geben: Eine zum erfolgreichen Beenden einer Transaktion und eine zum Zurücksetzen der Transaktion. Weiterhin sollen innerhalb der Transaktionen Operationen möglich sein, wobei an dieser Stelle die elementaren Operationen Lesen und Schreiben eines Objekts eingeführt werden. Somit ergeben sich fünf elementare Operationen:

1. BOT: Begin of Transaction, Beginn einer Transaktion.
2. Commit: Erfolgreiches Ende einer Transaktion, d.h. die Änderungen werden gültig.

3. Abort bzw. Rollback: Abbruch einer Transaktion, d.h. Änderungen werden zurückgesetzt.
4. r_i(A): Lesen (read) eines Objektes A von der Transaktion i.
5. w_i(A): Schreiben (write) eines Objektes A von der Transaktion i.

In dieser Dissertation werden Transaktionen im Datenbank-Kontext untersucht, so dass mit diesen elementaren Operationen aus Sicht des Anwendungsprogramms die notwendige Funktionalität gegeben ist. Beispielsweise können die Änderungsanweisungen der Datenbanksprache SQL (Structured Query Language [For99]), die den Standard bei relationalen Datenbanken bildet [Cod90], wie folgt durch write-Operationen abgebildet werden: „insert“ durch erstmaliges Schreiben, „update“ durch erneutes Schreiben und „delete“ durch Schreiben von Nullwerten.

In [Dad96] werden lokale und globale Transaktionen, die auch verteilte Transaktionen genannt werden, sinngemäß wie folgt definiert: *„Transaktionen, die zu ihrer Ausführung nur auf die lokal vorhandenen Daten zugreifen müssen und keine Abstimmung mit Transaktionen an anderen Rechnern erfordern, werden als lokale Transaktionen (local transactions) bezeichnet. Im Gegensatz hierzu stehen Transaktionen, die rechnerübergreifend ausgeführt werden müssen. Diese bezeichnet man als globale Transaktionen (global transactions) oder auch als verteilte Transaktionen (distributed transactions).“*

Verteilte Transaktionen werden an irgendeinem Rechner gestartet und breiten sich von dort auf andere Rechner aus. Somit teilt sich die globale Transaktion in mehrere *„Teiltransaktionen“* oder *„Subtransaktionen“*, wobei eine Teiltransaktion auf einem Rechner als lokale Transaktion auf diesem Rechner gesehen werden kann. Die initiierende (Teil-)Transaktion wird als *„Primärtransaktion“* (primary transaction, master transaction) bezeichnet. Eine Teiltransaktion kann ggf. weitere Teiltransaktionen erzeugen, so dass sich ein hierarchischer *„Transaktionsbaum“* ergeben kann, an dessen Wurzel die Primärtransaktion steht. Bei verteilten Transaktionen handelt es sich nach [Rah94] nicht um so genannte geschachtelte Transaktionen (nested transactions [Mos85], siehe Abschnitt 2.2.5), weil geschachtelte Transaktionen z.B. isolierte Zurücksetzung von Teiltransaktionen erlauben.

Die Beendigung einer Transaktion durch die Operation „commit“ bzw. „abort“ wird, wie bereits erwähnt, von einem Transaktionssystem bzw. Transaktionsmanager übernommen. Bei einer lokalen Transaktion wird die Einhaltung der ACID-Eigenschaften vom Transaktionssystem überwacht. Bei einer verteilten Transaktion müssen zusätzlich die Teiltransaktionen koordiniert werden, d.h. es muss sichergestellt werden, dass alle Teiltransaktionen entweder erfolgreich abgeschlossen oder zurückgesetzt werden. Im Allgemeinen wird angenommen, dass an jedem Rechner ein lokaler Transaktionsmanager existiert. Bei einer verteilten Transaktion wird ein Transaktionsmanager zum Koordinator, der sich mit den anderen beteiligten Transaktionsmanagern über das Ende der verteilten Transaktion koordiniert. Der Koordinator kann der Transaktionsmanager sein, auf dem die Primärtransaktion gestartet wurde, oder ein gesondertes Transaktionssystem. Für die Koordination wurden so genannte *„Commit-Protokolle“* entwickelt.

2.2.2. Commit-Protokolle und XA-Protokoll

Commit-Protokolle dienen im Wesentlichen dazu, um von den ACID-Eigenschaften die Atomarität und die Dauerhaftigkeit durch geeignete Logging- und Recovery-Maßnahmen einzuhalten. Bei lokalen Transaktionen wird dazu zunächst ein so genannter *„Commit-Satz“* in die Log-Datei geschrieben. Wenn der Commit-Satz z.B. wegen eines Rechnerausfalls nicht vollständig geschrieben werden konnte, dann wird bei Wiederanlauf des Rechners der jüngste transaktionskonsistente Datenbankzustand durch Zurücksetzen (Rollback, Abort) der Transaktionen hergestellt (Undo-Recovery). Andernfalls ist das erfolgreiche Ende der Transaktion garantiert, weil durch das Schreiben in die Log-Datei die Wiederholbarkeit der Transaktion garantiert ist (Redo-Recovery). Nach dem Schreiben des Commit-Satzes werden die Änderungen der erfolgreichen Transaktion anderen Transaktionen sichtbar gemacht, z.B. durch Freigabe von Sperren (siehe Abschnitt 2.2.3).

Bei verteilten Transaktionen sind nach [Lam94] zusätzlich folgende Anforderungen an das Commit-Protokoll zu stellen: Wenn ein Rechner bzw. Transaktionsmanager für einen Abbruch stimmt, dann muss der Koordinator ein *„globales Abort“* ausrufen. Wenn alle Rechner bzw. Transaktionsmanager für ein erfolgreiches Transaktionsende (Commit) stimmen, dann muss der Koordinator die verteilte Transaktion mit Commit beenden. Die einfache Erweiterung der Commit-Behandlung für den lokalen Fall auf mehrere Rechner reicht zur Erfüllung der Anforderungen nicht aus. Würde z.B. ein Koordinator die lokalen Transaktionsmanager zum Commit ihrer lokalen Transaktionen aufrufen und anschließend vor dem Schreiben des lokalen Commit-Satzes ein beteiligter Rechner ausfallen, dann würde dieser Rechner bei Wiederanlauf ein Zurücksetzen (Abort) durchführen, während andere Rechner ihre Teiltransaktionen erfolgreich beendet haben. Daher sind im verteilten Fall erweiterte Protokolle nötig.

Das bekannteste Commit-Protokoll für verteilte Transaktionen ist das 2-Phasen-Commit-Protokoll (2PC-Protokoll [Gra78]). Das 2PC-Protokoll basiert auf dem Grundgedanken, dass alle an einer verteilten Transaktion T beteiligten Rechner bzw. Transaktionsmanager darüber abstimmen, ob T global „committed“ oder „aborted“ wird. Hierzu dienen die folgenden beiden Phasen:

Phase 1 *„Prepare to Commit“*: Der Koordinator fordert die Teilnehmer auf, das Commit von T vorzubereiten, und erwartet das Abstimmungsergebnis: *„Ready-to-Commit“* oder *„Abort“*.

Phase 2 *„Commit / Abort“*: Falls der Koordinator von allen Knoten Zustimmung erhält, dann meldet der Koordinator *„Commit“*. Andernfalls, wenn nur ein Knoten seine Zustimmung verweigert, wird *„Abort“* gemeldet.

Eine wichtige Voraussetzung für das Funktionieren des 2PC-Protokolls ist es, dass jeder Rechner, der Ready-to-Commit gemeldet hat, das folgende Commit oder Abort des Koordinators dann auch tatsächlich ausführen kann. Insbesondere darf ein Rechner seine Zustimmung nicht wieder zurücknehmen bzw. die lokale Transaktion nach seiner Ready-to-Commit-Meldung aus eigener Entscheidung abbrechen oder zurücksetzen. Daher muss ein Rechner, der bei Wiederanlauf nach einem Absturz eine Teiltransaktion im Ready-to-Commit-Zustand hat, sich beim Koordinator melden und dort die endgültige Entscheidung erfragen. Hier offenbart sich auch die Schwäche des 2PC-Protokolls, weil bei einem dauerhaften Ausfall des Koordinators beteiligte Rechner ggf. blockiert bleiben. Abhilfe gegen derartige Blockierungen sind unter bestimmten Einschränkungen durch das 3-Phasen-Commit-Protokoll [BHG87] möglich.

Die Open Group (http://www.opengroup.org/, vormals die X/Open-Organisation), ein Konsortium verschiedener Hersteller von IT-Technologien, entwickelt unter anderem auch Standards für verteilte Transaktionsverarbeitung. Unter X/Open DTP (Distributed Transaction Processing [Ope03]) wurden Schnittstellen und Kommunikationsprotokolle für eine herstellerübergreifende Transaktionsverarbeitung spezifiziert, die auf dem 2PC-Protokoll basieren und zu einem De-

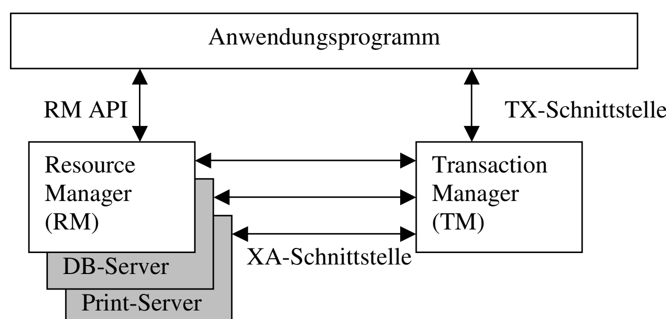

Abbildung 2.1.: X/Open-Modell eines zentralisierten Transaktionssystems (nach [Lam94])

facto-Standard geworden sind. In Abbildung 2.1 ist das X/Open-Modell für den zentralen Fall dargestellt. Ein Resource-Manager (RM), der im obigen Sinne ein Transaktionsmanager ohne Koordinationsfunktionalität ist, übernimmt die lokale Transaktionsverarbeitung, wobei die lokalen Anwendungen nicht nur Datenbanken, sondern beliebige Ressourcen sein können, wie z.B. ein Drucker, ein Dateisystem oder ein Mail-Server. Ein Transaction-Manager (TM) übernimmt die Rolle des Koordinators. Für die Resource-Manager wurde die so genannte XA-Schnittstelle und für die Transaction-Manager die so genannte TX-Schnittstelle spezifiziert.

Bei einer Transaktionsausführung übermittelt eine Anwendung über die TX-Schnittstelle dem TM den Beginn, das Ende oder den Abbruch einer verteilten Transaktion. Nach dem Beginn der Transaktion, für die der TM eine eindeutige Transaktionsnummer vergibt, kann die Anwendung über die RM API (Application Programming Interface, Anwendungsschnittstelle) Operationen auf die Ressource durchführen, z.B. SQL-Statements für Resource-Manager einer relationalen Datenbank. Wenn die verteilte Transaktion erfolgreich beendet werden soll, muss die Anwendung lediglich dem TM ein „Commit“ senden. Der TM führt dann über die XA-Schnittstelle ein 2PC-Protokoll mit den Resource-Managern durch.

Auch für verteilte Transaktionssysteme hat die Open Group unter X/Open DTP ein Modell entwickelt (siehe z.B. [GR93]). In diesem Fall gibt es mehrere Transaction-Manager, die jeweils auf „ihre“ Resourcen-Manager zugreifen, also analog zur Abbildung 2.1. Zusätzlich existiert zu jedem Transaction-Manager ein Communication-Manager (CM), der die Kommunikation zu anderen Transaction-Manager über deren CM herstellt. Neben X/Open DTP gibt es weitere Standardisierungsansätze, wie beispielsweise Remote Database Access (RDA) etc. (siehe z.B. [GR93, Rah94, Lam94]).

In Abschnitt 1.1 wurde motiviert, dass in dieser Arbeit die Replikation von Daten insbesondere im Kontext von Systemintegration betrachtet wird. Wenn Altanwendungen integriert werden müssen, ist die Fähigkeit der Altanwendungen, an einer verteilten Transaktionsverarbeitung teilnehmen zu können, von zentraler Bedeutung. Weil so genannte Integrations- bzw. Applikationsserver (siehe z.B. [Gor00]) im Allgemeinen Transaktionsdienste gemäß der X/Open-DTP-Spezifikation bereitstellen, können bei Verwendung dieser Transaktionsdienste Altanwendungen genau dann an verteilten Transaktionen teilnehmen, wenn sie die XA-Schnittstelle bzw. das XA-Protokoll unterstützen.

2.2.3. Synchronisation von Nebenläufigkeit und Serialisierbarkeit

Durch die Eigenschaft Isolation der ACID-Eigenschaften (siehe Abschnitt 2.2.1) soll im Mehrbenutzerbetrieb ein logischer Einbenutzerbetrieb erreicht werden. Die nebenläufige Verarbeitung (überlappende bzw. konkurrierende Verarbeitung) von Transaktionen soll den Benutzern verborgen bleiben. Wenn alle Transaktionen seriell, also nacheinander, ausgeführt werden, dann ist ohne Synchronisationsmaßnahmen ein logischer Einbenutzerbetrieb erreicht. Eine serielle Ausführung von Transaktionen geht aber zu Lasten des Gesamtdurchsatzes, d.h. es ist mit nicht akzeptablen Leistungseinbußen zu rechnen, weil z.B. bei Transaktionsunterbrechungen auf Grund von E/A-Vorgängen die Prozessoren nicht ausgelastet sind.

Demgegenüber birgt die nebenläufige Verarbeitung ohne jegliche Kontrolle die Gefahr der so genannten *„Nebenläufigkeitsanomalien“*: Verlorengegangene Änderungen (lost update), Lesen schmutziger Änderungen (dirty read), inkonsistente Analyse (non-repeatable read) und Phantome [Reu87, HR01]. Zur Vermeidung dieser Nebenläufigkeitsanomalien werden Synchronisationsverfahren verwendet, deren Ziel es ist, das so genannte Korrektheitskriterium *„Serialisierbarkeit“* [EGLT76, Pap86, BHG87] zu erfüllen. Serialisierbarkeit bzw. serialisierbare Ausführung bedeutet, dass die nebenläufige Ausführung von Transaktionen äquivalent zu irgendeiner seriellen Ausführung der Transaktionen ist. Hiermit ist nach [Rah94] gemeint: *„Äquivalent bedeutet in diesem Zusammenhang, dass für jede der Transaktionen dieselbe Ausgabe wie in der seriellen Abarbeitungsreihenfolge abgeleitet wird und dass der gleiche Datenbank-Endzustand erzeugt wird.“*

In [Dad96] werden *„korrekte Transaktionsausführungen“* anhand des Kriteriums Serialisierbarkeit definiert, wobei zwischen dem lokalen und dem verteilten Fall unterschieden wird. Es wird angemerkt, dass es im verteilten Fall nicht hinlangt, wenn die lokalen (Teil-)Transaktionsausführungen serialisierbar sind, sondern dass eine Äquivalenz zu einer seriellen Ausführung auch für die verteilte Transaktion gelten muss. Eine Prüfung der Serialisierbarkeit ist durch so genannte Transaktionsabhängigkeitsgraphen möglich, wobei das Ergebnis im Allgemeinen leider erst nachträglich ermittelt wird [Pei87]. Daher werden zur Synchronisation eher Verfahren eingesetzt, für die nachgewiesen wurde, dass sie die Serialisierbarkeit gewährleisten. Nach [Rah94] lassen sich die Mehrzahl der vorgeschlagenen Verfahren entweder den Sperrverfahren, den optimistischen Protokollen sowie den Zeitmarkenverfahren zuordnen.

Beim Sperrverfahren belegen Transaktionen während ihrer Verarbeitung die benötigten Objekte mit Sperren, wobei häufig zwischen Lesesperren (verträglich zu weiteren Lesesperren) und Schreibsperren (exklusive Sperren) unterschieden wird. Das so genannte *„Zwei-Phasen-Sperrprotokoll“* (2-Phase-Locking, 2PL-Protokoll [Gra78]) garantiert Serialisierbarkeit dadurch, dass in der Sperrphase benötigte Objekte vor dem Zugriff gesperrt werden und in der Freigabephase die Sperren freigegeben werden, ohne nach einer Freigabe weitere Sperren anzufordern. Falls andere Transaktionen auch auf gesperrte Objekte zugreifen möchten, müssen sie warten, bis diese Objekte wieder freigegeben werden. Dadurch kann es zu Verklemmungen von Transaktionen kommen, zu so genannten *„Deadlocks“*, d.h. dass Transaktionen zyklisch aufeinander warten. Diese Deadlocks müssen vermieden oder erkannt und behandelt werden (siehe z.B. [CES71, Elm86, Lev03]).

Die optimistischen Protokolle bzw. Synchronisationsverfahren (optimistic concurrency control [KR81]) gehen von der Annahme aus, dass Zugriffskonflikte selten und die Transaktionen eher kurz sind. Deshalb ist eine Wiederholung der Transaktion im Konfliktfall billiger als eine Blockierung und eine eventuelle Analyse auf Verklemmungen. Beim optimistischen Protokoll durchläuft eine Transaktion drei Phasen: In der Lesephase werden benötigte Objekte ohne Setzen irgendwelcher Sperren gelesen. Änderungsoperationen werden auf Kopien im privaten Adressbereich ausgeführt. In der Validationsphase, die exklusiv durchlaufen wird, wird geprüft, ob möglicherweise inkonsistente Daten gelesen wurden. Hierzu wird geprüft, ob andere Transaktionen mittlerweile Objekte geändert haben. Wenn die Validationsphase erfolgreich überstanden ist, dann werden in der Schreibphase alle geänderten Objekte gespeichert.

Die Zeitmarkenverfahren verwenden Zeitstempel (siehe Abschnitt 2.1.1), die bei Beginn einer Transaktion gesetzt werden (Begin-of-Transaction-Zeitmarke, BOT-Zeitmarke). Ein Vertreter der Zeitmarkenverfahren ist beispielsweise das „Basic Timestamp Ordering“ [BG81]. In der Serialisierungsreihenfolge ist die Position einer Transaktion durch die BOT-Zeitmarke festgelegt. Falls zwei Transaktionen miteinander in Konflikt geraten, so „gewinnt“ je nach angewendetem Verfahren entweder stets die „ältere“ oder stets die „jüngere“ und die andere Transaktion wird abgebrochen oder erneut aufgesetzt.

2.2.4. Abgeschwächte Serialisierbarkeitskriterien

Das in Abschnitt 2.2.3 vorgestellte Korrektheitskriterium Serialisierbarkeit wird auch *„Konfliktserialisierbarkeit“* genannt (siehe z.B. [Pap86]). Es gewährleistet eine korrekte, nebenläufige Verarbeitung von Transaktionen. Die Folge ist jedoch, dass Transaktionen unter Umständen abgebrochen werden müssen. Daher versuchte eine Reihe von Forschungsarbeiten, die Transaktionsverwaltung durch Unterstützung schwächerer Korrektheitskriterien als der Konfliktserialisierbarkeit zu erleichtern, d.h. durch Erweiterungen und/oder Nutzen weiterer Informationen die Anzahl der Transaktionsabbrüche im Vergleich zur Konfliktserialisierbarkeit zu reduzieren. Diese Arbeiten werden nach [RMB+93] in drei Kategorien klassifiziert:

1. Nutzen der Semantik von Datenbankoperationen: Bei diesen Arbeiten (siehe z.B. [Kor83, Wei88, Wei91, BR92]) werden im Allgemeinen Transaktionen als eine Folge von höher stufi-

gen Operationen (high-level operations) aufgefasst. Anstatt der einfachen Folge von Schreib- und Leseoperationen wird Kommutativität dieser höher stufigen Operationen genutzt, um Konflikte zu erkennen, die einen Abbruch von Transaktionen erfordern.
2. Nutzen der Semantik von Transaktionen: Allgemein fallen unter dieser Kategorie diejenigen Arbeiten, die Erweiterungen bzw. Abschwächungen der Serialisierbarkeit von Transaktionen ausschließlich auf Basis elementarer Transaktionsoperationen vornehmen (siehe z.B. [GM83, FÖ89]). Hierunter fallen beispielsweise auch die so genannten geschachtelten Transaktionen (siehe Abschnitt 2.2.5) oder abgeschwächte Serialisierbarkeitskriterien (siehe unten).
3. Nutzen von Integritätsbedingungen: Bei diesen Arbeiten werden die Integritätsbedingungen der beteiligten Datenbanken genutzt. Nebenläufige Transaktionen werden zugelassen, wenn nach bestimmten Regeln die Integritätsbedingungen nicht verletzt werden. Der bekannteste Vertreter dieser Kategorie ist die prädikatenweise Serialisierbarkeit (predicatewise serializability [KS88, KKB88]).

Für die vorliegende Arbeit ist die zweite Kategorie von besonderem Interesse und davon speziell die abgeschwächten Serialisierbarkeitskriterien. Hier werden nebenläufige Transaktionen auch dann noch zugelassen, obwohl im Sinne der Konfliktserialisierbarkeit Konflikte aufgetreten sind, die jedoch aus Anwendungssicht toleriert werden können. Folgende Serialisierbarkeitskriterien werden detaillierter betrachtet:

- Quasi-Serialisierbarkeit [DE89]
- Mehrversionen-Serialisierbarkeit [BG83a]
- Epsilon-Serialisierbarkeit [PL90, PL92]
- Konsistenzstufen [GLPT76], Isolationslevel [DD97] und Konsistenzanforderungen [GMW82]

Bei der Quasi-Serialisierbarkeit müssen die lokalen Transaktionsausführungen serialisierbar und die globalen Transaktionsausführungen müssen äquivalent zu einer quasi-seriellen Ausführung sein. Nach [DE89] bedeutet eine quasi-serielle Ausführung: *„Wenn eine totale Ordnung auf den globalen Transaktionen existiert, so dass, wenn G_i vor G_j in dieser Ordnung steht, in jeder lokalen Abarbeitungsreihenfolge gilt: wenn Operationen von G_i und G_j in der Abarbeitungsreihenfolge enthalten sind, dann stehen alle solche Operationen von G_i vor allen solchen Operationen von G_j."* Nach [Rah94] sind globale Transaktionsausführungen, die auf Grund von lokalen Transaktionen ohne Wechselwirkung auf die globalen (Teil-)Transaktionen nicht-serialisierbar sind, häufig quasi-serialisierbar und können damit zugelassen werden. Eine weitere Abschwächung zur Quasi-Serialisierbarkeit stellt die zweistufige Serialisierbarkeit (Two-Level Serializability [MRKS98]) dar, bei der die globalen und die lokalen Transaktionsausführungsfolgen getrennt betrachtet werden.

Die Mehrversionen-Serialisierbarkeit zeichnet sich dadurch aus, dass für geänderte Objekte zumindest zeitweise mehrere Versionen geführt werden. Während Schreiboperationen immer auf das aktuelle Objekt zugreifen und bei der Änderung eine neue Version erzeugen, wird bei Lesezugriffen entschieden, welche Version des Objekts gelesen wird. Dabei soll sichergestellt werden, dass nur solche Versionen gelesen werden, die eine konsistente Sicht auf die Datenbank gewährleisten. Dafür wird im Allgemeinen der Stand herangezogen, der zu Beginn der Transaktion gültig war, wie z.B. bei der *„Snapshot Isolation"* [BBG$^+$95]. Für die Synchronisation der Schreibzugriffe kann nach [Rah94] jedes Synchronisationsverfahren verwendet werden. Transaktionen, die nur lesend zugreifen, müssen nicht synchronisiert werden, weil über die Versionen eine konsistente Sicht gewährleistet ist. Der Nachteil besteht darin, dass möglicherweise veraltete Daten gelesen werden, insbesondere bei langlebigen Transaktionen.

Die Epsilon-Serialisierbarkeit verwendet so genannte Epsilon-Transaktionen (ET) anstatt der ACID-Transaktionen. Transaktionen werden unterteilt in Query-ET und Update-ET. Query-ET führen nur Lesezugriffe auf Daten durch. Demgegenüber enthalten Update-ET mindestens einen Schreibzugriff. Während Update-ET nach wie vor untereinander serialisiert werden, kann für eine Query-ET der tolerierbare Grad der Inkonsistenz spezifiziert werden. Dazu wird ein

Abstandsmaß zum aktuellen Objekt festgelegt, z.B. Anzahl erfolgter Aktualisierungen oder eine Wertdifferenz. Innerhalb eines Epsilon-Bereichs wird dann das Lesen „veralteter“ Daten toleriert. Ähnlich wird bei dem Korrektheitskriterium *„bounded ignorance“* [KB91] verfahren, bei dem eine bestimmte Zahl ausgelassener Schreibtransaktionen toleriert wird.

Durch das 2-Phasen-Sperrprotokoll (siehe Abschnitt 2.2.3) wird Konfliktserialisierbarkeit garantiert, aber die Lese- und Schreibsperren müssen „lange“ gesetzt bleiben, d.h. bis zum Transaktionsende. In [GLPT76] werden vier Konsistenzstufen mittels „kurzer“ Sperren definiert: Je nach Konsistenzstufe sind Schreib- bzw. Lesesperren möglich, die schon vor Transaktionsende freigegeben werden. Dadurch können mehr Transaktionen parallel ausgeführt werden, aber es werden Nebenläufigkeitsanomalien in Kauf genommen (siehe Abschnitt 2.2.3). Im SQL92-Standard (siehe z.B. [DD97]) werden ebenfalls vier Konsistenzstufen (so genannte Isolationslevel) definiert, bei denen in Abhängigkeit der Stufe jeweils die Nebenläufigkeitsanomalien „Dirty Reads“, „Nonrepeatable Reads“ oder „Phantome“ toleriert werden. In [GMW82] werden abgestufte Konsistenzanforderungen für Lese-Transaktionen definiert, d.h. für Transaktionen, die nur Lesezugriffe durchführen, kann zwischen starker, abgeschwächter und keiner Konsistenzanforderung (strong, weak and no consistency requirements) unterschieden werden.

Wenn die Quasi-Serialisierbarkeit als Korrektheitskriterium für die Synchronisation von nebenläufigen Transaktionen eingesetzt wird, müssen im Vergleich zur Konfliktserialisierbarkeit im Allgemeinen weniger Transaktionen abgebrochen werden. Dennoch wird die Konsistenz gewahrt. Bei der Mehrversionen-Serialisierbarkeit wird ebenfalls eine konsistente Sicht auf die Datenbank gewährleistet, jedoch sind bei Lesezugriffen die Daten mitunter veraltet. Bei der Epsilon-Serialisierbarkeit werden bewusst Inkonsistenzen bei Lesezugriffen in Kauf genommen, d.h. es werden möglicherweise sowohl veraltete als auch aktuelle Daten gelesen. Bei Verwendung von Konsistenzstufen ist in der schwächsten Form sogar das Lesen ungültiger Daten möglich. Insbesondere die Mehrversionen- und die Epsilon-Serialisierbarkeit spielen bei der Replikation eine Rolle, weil sich der Korrektheitsbegriff auf die replizierten Daten übertragen lässt (siehe Abschnitt 3.2.2).

2.2.5. Weiterführende Transaktionskonzepte: Sagas und Queued Transactions

Als Erweiterung zu dem Transaktionskonzept, das auf den ACID-Eigenschaften (siehe Abschnitt 2.2.1) basiert, wurden Konzepte entwickelt, in denen innerhalb einer Transaktion weitere Transaktionen gestartet werden können, d.h. eine Vater-Transaktion kann Kind-Transaktionen starten, die auch wiederum weitere Kind-Transaktionen starten können. Die Kind-Transaktionen werden auch als Teil- oder Subtransaktionen bezeichnet, wobei es sich hier im Gegensatz zu den Subtransaktionen einer verteilten Transaktion, die eine Strukturierung hinsichtlich der ausführenden Rechner darstellt (siehe Abschnitt 2.2.1), um eine Strukturierung von „Anwendungsfunktionen“ handelt. Mit diesen Konzepten werden somit geschachtelte Transaktionen (nested transactions [Mos85]) möglich. Durch diese weiterführenden Transaktionskonzepte sollen vor allem lange Blockierungen vermieden werden, die durch zu langes Sperren von Objekten entstehen können.

Je nachdem, ob die Vater-Transaktion erst nach dem Ende der Kind-Transaktion committed oder auch vor dem Ende der Kind-Transaktion bereits committed hat, kann zwischen geschlossen- und offen-geschachtelten Transaktionen unterschieden werden [HR01]. Geschlossen-geschachtelte Transaktionen gewährleisten weiterhin die ACID-Eigenschaften, bieten aber isolierte Zurücksetzbarkeit der Subtransaktionen und damit intern differenzierte Rücksetzmöglichkeiten. Demgegenüber garantieren die offen-geschachtelten Transaktionen nicht mehr die Isolation und haben abgeschwächte Konsistenzbegriffe, d.h. die ACID-Eigenschaften werden nicht vollständig erfüllt.

Im Folgenden sollen zwei Transaktionskonzepte der offen-geschachtelten Transaktionen detaillierter betrachtet werden, die für die vorliegende Arbeit von Interesse sind (siehe Abschnitt 6.3):

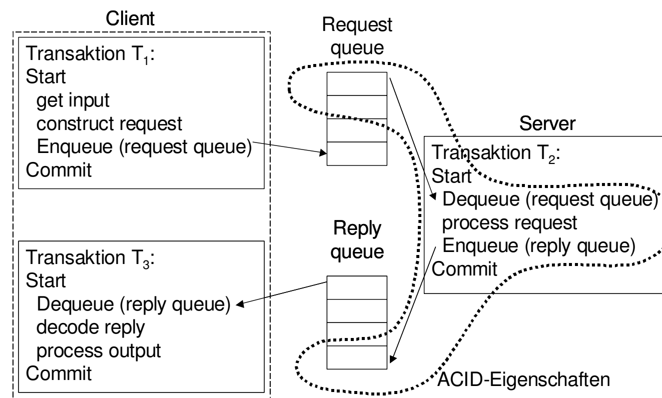

Abbildung 2.2.: Queued Transaction Processing Modell (nach [BN97])

- Sagas [GMS87] (auch *„verkettete Transaktion“*)
- Queued Transactions [BHM90, BN97]

Eine Saga-Transaktion besteht aus einer Folge von einzelnen (Teil-)Transaktionen T_1,...,T_n. Bei Abbruch können die bis dahin abgeschlossenen Einzeltransaktionen T_i ($i \in 1,..,n$) nicht mehr zurückgesetzt werden, da sie bereits erfolgreich beendet wurden, d.h. sie müssen jeweils durch geeignete Kompensationstransaktionen [KLS90] in ihrer Wirkung rückgängig gemacht werden. Somit muss zu jeder Transaktion T_i eine geeignete Kompensationstransaktion C_i existieren. In [GMS87] wird diese Form als Backward Recovery bezeichnet. Alternativ kann ab einer geeigneten Stelle („Saga Save Point“) ein Forward Recovery durchgeführt werden, d.h. es erfolgt ein Zurücksetzen bis zum Saga Save Point und ein neues Aufsetzen der nicht ausgeführten Teiltransaktionen. Für verkettete Transaktionen treffen isolierte Zurücksetzbarkeit und Serialisierbarkeit im Sinne der ACID-Eigenschaften nicht mehr zu. Sie sind für Anwendungen geeignet, die auf diese Eigenschaften nicht unbedingt angewiesen sind und diese abgeschwächte Form der Konsistenz tolerieren.

Das Queued Transaction Processing Modell wird in Abbildung 2.2 anhand des Client/Server-Modells erläutert. Eine Transaktion zerfällt in drei Teiltransaktionen, wobei die Kommunikation über zwei Warteschlangen, der *„request queue“* und der *„reply queue“*, erfolgt:

T_1 Ein Client stellt eine Anfrage in die request queue.
T_2 Der Server holt die Anfrage aus der request queue, bearbeitet sie und stellt eine Antwort in die replay queue.
T_3 Der Client holt die Antwort aus der reply queue und bearbeitet sie.

Die drei Teiltransaktionen unterliegen jeweils den ACID-Eigenschaften, beispielsweise muss die Teiltransaktion T_2, d.h. das Holen der Anfrage, die Bearbeitung und das Bereitstellen der Antwort, atomar verarbeitet werden (in Abbildung 2.2 durch die gepunktete Linie angedeutet). Durch die Entkopplung über die Warteschlangen wird unter anderem eine größere Unabhängigkeit der Systeme erreicht, die durch einen höheren Kommunikationsaufwand erkauft wird.

2.3. Allgemeine Konsistenzbegriffe und Korrektheit

Um bewerten zu können, ob die Daten bzw. die Zugriffe auf die Daten insbesondere im Mehrbenutzerbetrieb widerspruchsfrei sind, wurden Konsistenzbegriffe und Korrektheitskriterien eingeführt. Bevor im Abschnitt 3.2.2 auf die Konsistenz und Korrektheit von Replikaten eingegangen wird, werden die Begriffe in diesem Abschnitt im Allgemeinen diskutiert. Speziell im Datenbankbereich wird auch der Begriff Integrität verwendet, mit dem Aussagen zu einer zulässigen Verarbeitung von Operationen und Transaktionen gemacht werden.

Obwohl die Begriffe in der Literatur unterschiedlich definiert werden, kann vereinfacht gesagt werden, dass mittels der Begriffe Konsistenz, Korrektheit und Integrität die Menge der zulässigen Transaktionen festgelegt wird, d.h. beispielsweise, dass Transaktionen, die die Konsistenz verletzen, abgebrochen werden. Wenn z.B. eine „starke“ Definition von Konsistenz angewandt wird, dann werden mitunter mehr Transaktionen abgebrochen als nötig. Daher wurden auch so genannte „abgeschwächte“ Konsistenz- bzw. Korrektheitsbegriffe definiert, die zwar eine größere Anzahl nebenläufiger Transaktionen zulassen, aber dennoch den Anforderungen spezieller Anwendungsbereiche genügen. Während Überblicke z.B. in [EGLT76, TGGL82, Pap86] niedergelegt sind, sollen die für diese Arbeit wichtigen Aspekte im Folgenden diskutiert werden:

- Konsistenz, Korrektheit und Integrität (Abschnitt 2.3.1)
- Integritäts- bzw. Konsistenzbedingungen (Abschnitt 2.3.2)
- Daten- und Transaktionskonsistenz, abgeschwächte Konsistenz (Abschnitt 2.3.3)
- Konsistenzmodelle verteilter, gemeinsamer Speicher (Abschnitt Abschnitt 2.3.4)

2.3.1. Konsistenz, Korrektheit und Integrität

In der Literatur werden die Begriffe Konsistenz, Korrektheit und Integrität nicht einheitlich verwendet. Integrität wird z.B. in [TvS02] hinsichtlich der Zuverlässigkeit verteilter Systeme wie folgt definiert: *„Integrität ist die Eigenschaft, dass Änderungen an einem System nur von autorisierten Benutzern vorgenommen werden können.“* Im Datenbankbereich bezieht sich Integrität auf den Zustand der Daten, z.B. nach [Dat03]: *„Vertraulichkeit (Security) meint Schutz der Daten gegen nicht autorisierte Benutzer. Integrität meint Schutz der Daten gegen autorisierte Benutzer.“* Selbst in der Datenbank-Literatur werden die Begriffe Integrität, Konsistenz und Korrektheit unterschiedlich definiert: Einerseits werden Abgrenzungen zwischen den Begriffen vorgenommen, andererseits werden sie mehr oder weniger synonym verwendet.

In [Sch90] wird Integrität und Konsistenz wie folgt unterschieden: *„Integrität bezeichnet die Übereinstimmung der Datenbank mit der realen Welt, was nicht vom Datenbanksystem überprüft werden kann. Konsistenz bedeutet das Erfüllen von Konsistenzbedingungen. Konsistenzbedingungen sind explizit spezifizierte Regeln zur genaueren Modellierung des abzubildenden Ausschnitts der realen Welt. Diese Regeln können vom Datenbanksystem überprüft und sichergestellt werden.“* In ähnlicher Weise werden die Begriffe häufig gegeneinander abgegrenzt, d.h. durch Einhalten spezieller Bedingungen wird Konsistenz erreicht, während bei Integrität bzw. Korrektheit ein Vergleich zur realen Welt hergestellt wird. So auch in [Dat03]: *„Integrität bedeutet, sicherzustellen, dass die Dinge, die Benutzer tun, korrekt sind.“* und *„Ein System kann nicht Wirklichkeit erzwingen, nur Konsistenz.“* Hier wird also Integrität und Korrektheit in ähnlicher Weise verwendet, wobei Korrektheit als Übereinstimmung mit der realen Welt definiert wird.

Andere Autoren beziehen auch den Begriff Konsistenz auf die reale Welt, so dass die Begriffe Integrität, Konsistenz und Korrektheit synonym verwendet werden können. So auch nach [SS83a]: *„Synonym zum Begriff der Integrität der Daten verwenden wir den Begriff Konsistenz. Mit Konsistenz hat man den Blick auf die Widerspruchsfreiheit (auch gegenüber der Realwelt) und die Vollständigkeit der Daten im Auge, weniger die technischen Aspekte, die zu ihrer Erhaltung zu berücksichtigen sind. Letzten Endes ist mit beiden Begriffen die Korrektheit der Daten in Bezug auf die Realwelt gemeint.“* Eine ähnliche Auffassung ist im Glossar von [GR93] zu finden, wo konsistent mit korrekt erklärt wird.

Einigkeit besteht darüber, dass nicht überprüft werden kann, ob die Daten einer Datenbank mit der realen Welt übereinstimmen. Es kann beispielsweise bei Adressdaten geprüft werden, ob eine zulässige Postleitzahl eingegeben wurde, aber es lässt sich nicht prüfen, ob eine zulässige Adresse dem wirklichen Wohnsitz einer Person entspricht oder ob diese Daten mittlerweile nicht mehr aktuell sind, weil die Person umgezogen ist. Es kann lediglich untersucht werden, ob die zuletzt eingegebenen Daten speziellen Bedingungen genügen. Diese Bedingungen werden *„Integritätsbedingungen“* oder auch *„Konsistenzbedingungen“* genannt (siehe Abschnitt 2.3.2). Auch

die Einbeziehung so genannter Gültigkeitszeiten, mit denen Zustandsänderungen von Objekten der realen Welt in temporalen Datenbanken (siehe z.B. [Sno99]) berücksichtigt werden, schafft keine Abhilfe, weil auch diese Zeiten erst bei Eingabe in die Datenbank auf Integritäts- bzw. Konsistenzbedingungen geprüft werden können, selbst bei automatischer Eingabe beispielsweise über Sensoren. In [Ram93] wird demzufolge ein Datenobjekt als korrekt definiert, wenn es *„logisch konsistent“* ist, d.h. Integritätsbedingungen werden eingehalten, und wenn es *„zeitlich konsistent“* ist, d.h. spezielle zeitliche Bedingungen werden erfüllt.

In dieser Arbeit werden die Begriffe Integrität, Konsistenz und Korrektheit analog zu [SS83a] (siehe oben) synonym verwendet. Da, wie bereits angemerkt, nicht überprüft werden kann, ob die Daten mit der realen Welt übereinstimmen, wird das Augenmerk im Allgemeinen auf die zuletzt eingebrachten Änderungen gelegt, also nach [Ram93] auf die *„logische Konsistenz“*. In [Rah94] wird Konsistenz als zweite der ACID-Eigenschaften (siehe Abschnitt 2.2.1) wie folgt definiert: *„Die Transaktion ist die Einheit der Datenbank-Konsistenz. Dies bedeutet, dass bei Beginn und nach Ende einer Transaktion sämtliche physischen und logischen Integritätsbedingungen [SW85, Reu87] erfüllt sind.“* Änderungen sind demnach genau dann konsistent und korrekt, wenn sämtliche Integritätsbedingungen erfüllt sind.

2.3.2. Integritäts- bzw. Konsistenzbedingungen

Wie auch in [GR93], wo im Glossar Integritäts- und Konsistenzbedingungen gleich gesetzt werden und Konsistenzbedingungen das Einhalten von Vor-, Nach- und Transformationsbedingungen bei der Transaktionsverarbeitung meint, soll in dieser Arbeit kein Unterschied hinsichtlich dieser Begriffe erfolgen. In der deutschsprachigen Datenbank-Literatur wird im Allgemeinen von Integritätsbedingungen anstatt von Konsistenzbedingungen gesprochen. So wird der Begriff Integritätsbedingung z.B. in [HS00] wie folgt definiert: *„Allgemein wird als Integritätsbedingung eine Bedingung für die „Zuverlässigkeit“ oder „Korrektheit“ bezeichnet. In Bezug auf Datenbanken kann diese Bedingung einzelne Datenbankzustände, Zustandsübergänge oder auch langfristige Datenbankentwicklungen betreffen.“*

Integritätsbedingungen können nach verschiedenen Dimensionen klassifiziert werden, z.B. in operationale und semantische Integritätsbedingungen [SS83a]. Die operationalen Integritätsbedingungen sind Bedingungen, die nicht vom Datenbankprogrammierer beeinflusst werden können, sondern vom Datenbankmanagementsystem und dem zugrunde liegenden Betriebssystem. Hierunter fallen die so genannten physischen Integritätsbedingungen, die die Vollständigkeit der Zugriffspfade und der physischen Speicherstrukturen beinhalten, und die Ablaufintegritätsbedingungen, die die Korrektheit der ablaufenden Programme im Mehrbenutzerbetrieb betreffen. Letzteres wird im Allgemeinen durch ein Transaktionssystem (siehe Abschnitt 2.2) gewährleistet.

Semantische Integritätsbedingungen sind Bedingungen, die vom Datenbankprogrammierer beeinflusst werden können. Insbesondere mittels der semantischen Integritätsbedingungen ist es möglich, Bedingungen der realen Welt in das Datenbankmanagementsystem zu übernehmen und so die Benutzer vor Fehleingaben zu schützen. Nach [HR01] lassen sich semantische Integritätsbedingungen hinsichtlich folgender Kategorien klassifizieren:

- Modellinhärente versus modellunabhängige Integritätsbedingungen: Modellinhärente Bedingungen sind abhängig vom jeweiligen Datenmodell, z.B. für relationale Datenmodelle Primärschlüsseleigenschaften, referentielle Integrität (für Fremdschlüssel) oder Wertebereiche für Attribute.
- Reichweite der Bedingungen: Unterscheidung nach Attributwert-, Satz- und Satztyp-Bedingungen sowie satztypübergreifende Bedingungen.
- Statische und dynamische Bedingungen: Während die statischen Bedingungen die Zustände der Datenbank beschränken, z.B. Gehalt kleiner als Grenzwert, legen die dynamischen Bedingungen zulässige Zustandsübergänge fest, z.B. Gehalt darf nicht kleiner werden.
- Zeitpunkt der Überprüfbarkeit: Unverzögerte versus verzögerte Integritätsbedingungen.

Ein Datenbankmanagementsystem prüft bei einer Änderung durch die so genannte *„Integritätskontrolle“*, ob alle Integritätsbedingungen eingehalten werden. Wenn das der Fall ist, dann kann die Änderung gültig gemacht werden und der neue Zustand wird als korrekt und konsistent eingestuft.

2.3.3. Daten- und Transaktionskonsistenz, abgeschwächte Konsistenz

Eine Datenbank ist konsistent, wenn alle Konsistenz- bzw. Integritätsbedingungen eingehalten werden (siehe Abschnitt 2.3.2). Damit ist die Datenbank entweder konsistent oder sie ist inkonsistent. Da es Anwendungsbereiche gibt, in denen nicht alle Integritätsbedingungen zu jedem Zeitpunkt vollständig erfüllt werden können, wurden in der Literatur so genannte *„abgeschwächte Konsistenzbegriffe“* oder, hier synonym, *„abgeschwächte Korrektheitskriterien“* definiert. Abgeschwächte Konsistenz spielt häufig bei Verwendung von Replikaten eine Rolle (siehe Abschnitt 3.2.2). An dieser Stelle wird zunächst kurz auf abgeschwächte Konsistenz im Allgemeinen eingegangen.

In [RC96] wird eine Taxonomie für Korrektheitskriterien in Datenbank-Anwendungen vorgestellt, wobei zuerst eine grobe Klassifizierung nach Konsistenz von Datenbank-Zuständen und Korrektheit von Transaktionen vorgenommen wird. Da in dieser Arbeit die Begriffe synonym verwendet werden, wird dementsprechend von Daten- und Transaktionskonsistenz gesprochen. In [Len97] erfolgt eine Unterteilung der Datenkonsistenz in physische und logische Datenkonsistenz. Diesen Konsistenzbegriffen können folgende Integritätsbedingungen zugeordnet werden (vergleiche Abschnitt 2.3.2):

- physische Datenkonsistenz: physische Integritätsbedingungen
- logische Datenkonsistenz: semantische Integritätsbedingungen
- Transaktionskonsistenz: Ablaufintegritätsbedingungen

Eine Abschwächung der Konsistenzbegriffe kann nun daraus resultieren, dass die zugehörigen Integritätsbedingungen nicht vollständig bzw. abgeschwächt erfüllt werden und/oder nur zu bestimmten Zeitpunkten gelten [WQ87, SR90]. Eine Abschwächung der physischen Datenkonsistenz wird im Allgemeinen nicht betrachtet, d.h. es wird davon ausgegangen, dass ein Rechner entweder die Daten korrekt speichert und korrekt kommuniziert oder aber ausfällt. Im Forschungsgebiet „Verlässliche Systeme“ [ALRL04] werden auch Fehler betrachtet, die auf Grund fehlerhafter Speicherung oder Kommunikation entstehen, ob durch Störung oder böswilligen Eingriff. Diese so genannten *„Byzantinischen Fehler“* (siehe z.B. [MR98, CL99]), in dessen Zusammenhang auch von *„interaktiver Konsistenz“* gesprochen wird [LSP82], werden in dieser Arbeit nicht behandelt.

Die logische Datenkonsistenz kann dadurch abgeschwächt sein, dass die semantischen Integritätsbedingungen nicht für die komplette Datenbank gelten, ob zentral oder verteilt, sondern beispielsweise nur für Mengen von Objekten bzw. für einzelne Objekte. Insbesondere in verteilten Datenbanken kann zwischen lokaler, logischer Datenkonsistenz und globaler, logischer Datenkonsistenz unterschieden werden (siehe z.B. [Len97]). Dabei ist es möglich, dass jede lokale Datenbank die logische Datenkonsistenz erfüllt, aber global gesehen Inkonsistenzen auftreten können. Insbesondere bei ressourcenschwachen Systemen wie z.B. bei mobilen Systemen, ist eine derartige Unterscheidung sinnvoll. In [PB99] werden so genannte *„logische Cluster“* für Daten bzw. Integritätsbedingungen eingeführt, wobei die Cluster z.B. auf Basis der physikalischen Lokalisation der Daten gebildet werden können. Hierüber werden dann abgeschwächte Konsistenzbegriffe für mobile Datenbanken definiert (siehe auch Abschnitt 3.2.2).

Da die Ablaufintegritätsbedingungen wesentlich die nebenläufige Verarbeitung von Transaktionen im Mehrbenutzerbetrieb betrifft, ergibt sich eine abgeschwächte Transaktionskonsistenz durch abgeschwächte Serialisierbarkeitskriterien. Wie in Abschnitt 2.2.4 dargestellt, gewährleistet die Konfliktserialisierbarkeit eine korrekte, nebenläufige Verarbeitung, während z.B. die Mehrversionen-Serialisierbarkeit und die Epsilon-Serialisierbarkeit den Zugriff auf „veraltete“

Daten erlauben. Bei der Epsilon-Serialisierbarkeit kann ein Abstandsmaß vom aktuellen Objekt zum veralteten Objekt festgelegt werden, dass demnach als *„Konsistenzmaß“* betrachtet werden kann. Unter *„Konsistenzmaß“* soll hier ein Maß verstanden werden, mit dem die abgeschwächte Konsistenz bzw. das Verhältnis konsistenter Daten zu inkonsistenter Daten oder ähnliches nummerisch ausgedrückt wird. Auch die Serialisierbarkeitskriterien Konsistenzstufen bzw. Isolationslevel können als Konsistenzmaß aufgefasst werden (siehe Abschnitt 2.2.4).

2.3.4. Konsistenzmodelle verteilter, gemeinsamer Speicher

Im Forschungsbereich „verteilter, gemeinsamer Speicher“ (distributed shared memory, DSM) wurden so genannte *„Konsistenzmodelle“* entwickelt, die nach [Web98] wie folgt definiert sind: *„Unter einem Konsistenzmodell versteht man eine Vereinbarung zwischen der übergeordneten, benutzenden Software und dem verteilten Speicher.“* Im Wesentlichen geht es darum, dass Prozesse, die Lese- und Schreibzugriffe auf einen gemeinsamen Speicher durchführen, bestimmte Garantien vom Speichersystem gewährleistet bekommen. Zwar wird in [TvS02] allgemein vom Datenspeicher gesprochen, womit Speicher, Datenbanken und Dateisysteme gemeint sind, aber häufig wird auf den Kontext Speicher bezogen, so dass auch von Speicher-Konsistenzmodellen (memory consistency models) gesprochen wird. Überblicke sind z.B. in [GLL+90, Mos93, AG96] zu finden.

Die *„strikte Konsistenz“* oder auch strenge Konsistenz, der „stärkste“ Konsistenzbegriff, bedeutet, dass jeder Lesezugriff auf die Speicherstelle x den Wert der global zeitlich letzten Schreiboperation auf x liefert. In verteilten Systemen ist strenge Konsistenz im Allgemeinen aber nicht realisierbar, weil hierfür eine absolute, zeitliche Ordnung der Zugriffe benötigt wird und eine Synchronisation aller Rechner auf eine absolute, physikalische Realzeit nicht möglich ist (siehe Abschnitt 2.1.1). Beispiele für abgeschwächte Konsistenzmodelle, aufgeführt in absteigender Reihenfolge, ist die *„sequentielle Konsistenz“* [Lam79], die *„kausale Konsistenz“* [HA90], die *„PRAM-Konsistenz“* [LS88], die *„schwache Konsistenz“* [DSB88] oder die *„Release-Konsistenz“* [GLL+90]. Bei den beiden letztgenannten Konsistenzmodellen werden explizite Synchronisationsoperationen verwendet. Die Abschwächung ergibt sich im Allgemeinen durch die Art und Weise, wie die Zugriffe paralleler Prozesse auf die Speicher synchronisiert werden.

Nach [TvS02] ist die sequentielle Konsistenz mit der Serialisierbarkeit von Transaktionen vergleichbar (siehe Abschnitt 2.2.4), wobei der Unterschied in der Granularität liegt: *„Die sequentielle Konsistenz ist für Lese- und Schreiboperationen definiert, während die Serialisierbarkeit für Transaktionen ist, die solche Operationen in sich aufnehmen.“* In dieser Arbeit werden Replikate betrachtet, die in Datenbanken lokalisiert sind, auf denen mit Transaktionen zugegriffen wird. Daher sind weniger die Speicher-Konsistenzmodelle von Interesse, sondern eher die Serialisierbarkeit von Transaktionen bzw. die Konsistenzbegriffe aus dem Forschungsgebiet Datenbanken (siehe Abschnitt 2.3.3).

3. Datenreplikation und verwandte Arbeiten

Die Replikation von Daten, d.h. das mehrfache, redundante Speichern von Kopien an verschiedenen Lokalisationen, ist ein wichtiges Thema in verteilten Systemen. Der Grund für die Datenreplikation (im Folgenden einfach Replikation; Begriffsdefinitionen folgen im Abschnitt 3.2.1) liegt darin, dass die Verfügbarkeit und/oder Performance des Gesamtsystems erhöht werden soll bzw. zwischen bereits bestehenden autonomen Anwendungssystemen Daten abgeglichen werden müssen. Mit Replikation soll erreicht werden, dass alle Kopien eines Objekts den gleichen Wert haben und bei Änderung einer Kopie automatisch die Änderung auch an den anderen Kopien durchgeführt wird. Durch das mehrfache Speichern identischer Kopien eines Objekts kann dann auf eine beliebige Kopie zugegriffen werden, um z.B. den Wert des Objekts zu lesen. Der Vorteil dieser erhöhten Verfügbarkeit wird dadurch erkauft, dass eine Strategie umgesetzt werden muss, die bei Änderung einer Kopie die anderen Kopien automatisch aktualisiert, d.h. die Änderung muss propagiert werden.

Der weitere Aufbau dieses Kapitels gliedert sich wie folgt: Zunächst werden in Abschnitt 3.1 die Ziele der Replikation und die damit einhergehenden Konflikte erläutert. Anschließend werden in Abschnitt 3.2 die Grundlagen von Datenreplikation aufgeführt, insbesondere werden Begriffe definiert, Konsistenz und Korrektheit erläutert sowie Klassifizierungen von Replikationsstrategien aufgezeigt. Der Abschnitt 3.3 beschäftigt sich mit den traditionellen Replikationsstrategien, wobei über eine geeignete Klassifizierung eine kurze Vorstellung verschiedener Strategien vorgenommen wird. Im Abschnitt 3.4 wird abschließend auf verwandte Arbeiten mit den Themen adaptive Replikationsstrategien und Datenintegration eingegangen.

3.1. Ziele der Replikation

Replikation wird im Allgemeinen in verteilten Systemen durchgeführt (siehe Abschnitt 2.1). In [Dat03] werden 12 Regeln angegeben, die für ein verteiltes Datenbanksystem gelten sollten. Diese Regeln lassen sich jedoch nicht immer vollständig erfüllen, weil sie zumindest teilweise in Widerspruch zueinander stehen. Aus Sicht der Replikation seien zwei relevante Regeln aufgeführt, die in [Rah94] wie folgt wiedergegeben sind:

Lokale Autonomie: Jeder Rechner sollte ein Maximum an Kontrolle über die bei ihm gespeicherten Daten haben. Insbesondere sollte der Zugriff auf diese Daten nicht von anderen Rechnern abhängen.

Replikationstransparenz: Die replizierte Speicherung von Teilen der Datenbank sollte für den Benutzer unsichtbar bleiben; die Wartung der Redundanz obliegt ausschließlich der Datenbank-Software.

Einerseits soll also ein lokaler Rechner möglichst autonom die bei ihm gespeicherten Daten kontrollieren, andererseits bedarf es einer rechnerübergreifenden Koordination, um die bei der Replikation auftretende Redundanz zu warten. Diese Anforderungen stehen sich somit konträr gegenüber. In der Literatur wurden weitere, sich ähnelnde Anforderungen und Ziele aufgeführt, wobei die Ziele im Allgemeinen ein gewisses Konfliktpotential besitzen. In [Len97] wird festgestellt, dass *„Konsistenz und Effizienz gegenläufige Zielsetzungen sind“*. Als Hauptbestandteile von Effizienz werden kurze Antwortzeiten und hohe Verfügbarkeit und Zuverlässigkeit gesehen. Neben der Beschreibung und Klassifizierung verschiedener Replikationsstrategien werden in [BD96] drei in Konflikt zueinander stehende Ziele bei der Replikation angegeben:

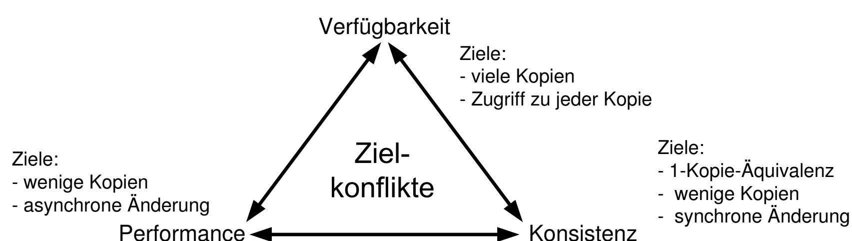

Abbildung 3.1.: Zielkonflikte der Datenreplikation (nach [Has97])

1. Erhaltung der Datenkonsistenz
2. Erhöhung der Verfügbarkeit und des effizienten Zugriffs
3. Minimierung des Aufwands einer Änderungsoperation

An Stelle des dritten Punktes wird in [Has97] Performance als ein Ziel genannt, wobei die Autoren die Anforderung „Aufwand“ und „Performance“ im Wesentlichen gleich verstehen. In den beiden letzt genannten Papieren werden die drei Ziele in einem Konfliktdreieck dargestellt, wie es z.B. in Abbildung 3.1 zu sehen ist. Die Kernaussage dabei lautet, dass die Verbesserung eines dieser Ziele im Allgemeinen eine Verschlechterung der anderen Ziele nach sich zieht. Beispielsweise ist eine Verbesserung der Konsistenz häufig nur zu Lasten der Verfügbarkeit und der Performance zu erreichen oder eine Erhöhung der Performance ist nur dann möglich, wenn Abstriche bei der Verfügbarkeit und der Konsistenz hingenommen werden.

Weiterhin zeigt Abbildung 3.1, dass für jedes der drei Ziele Konsistenz, Performance und Verfügbarkeit einerseits die Anzahl an Kopien und andererseits die Art des Zugriffs eine gewichtige Rolle spielen. Wenn z.B. eine hohe Konsistenz gefordert wird, dann sollten möglichst wenig Kopien vorhanden sein und diese sollten bei einer Änderung möglichst synchron aktualisiert werden. Es sei angemerkt, dass die Konflikte der Replikation in Abbildung 3.1 vereinfacht dargestellt sind, trotzdem ein erstes Verständnis der Problematik erkennen lassen. Bei einer detaillierteren Darstellung sollte z.B. darauf eingegangen werden, ob es sich bei dem Zugriff auf die Kopien um Lese- oder um Schreibzugriffe handelt bzw. wie die Abhängigkeiten sind. Beispielsweise wird bei Lesezugriffen die Verfügbarkeit erhöht, wenn viele Kopien vorhanden sind und eine beliebige Kopie gelesen werden darf. Allerdings kann eine hohe Anzahl an Kopien die Verfügbarkeit bei Schreibzugriffen stark einschränken, insbesondere dann, wenn Änderungen synchronisiert durchgeführt werden müssen.

Es lässt sich somit festhalten, dass bei der Replikation die Anzahl der Kopien und die Art des Zugriffs wichtige Faktoren sind, wobei diese Faktoren voneinander abhängen. Mit „Art des Zugriffs“ ist die Koordination von Lese- und Schreibzugriffen gemeint, d.h. die Replikationsstrategie (siehe auch Definition 10 in Abschnitt 3.2.1). Während bei der Anzahl der Kopien im Allgemeinen wenig Spielraum gegeben ist, z.B. durch die vorhandenen Systeme vorgegeben, können verschiedene Replikationsstrategien zum Einsatz kommen. Je nach Zielsetzung, genauer gesagt, je nach Priorisierung der Ziele Konsistenz, Performance und Verfügbarkeit in Abhängigkeit des Anwendungsbereichs sind unterschiedliche Replikationsstrategien denkbar. Daher wurden in der Vergangenheit bereits die unterschiedlichsten Replikationsstrategien für verschiedene Bereiche und Zielsetzungen vorgestellt. Übersichten sind z.B. in [DGMD85, BD96, Len97] zu finden.

Die „traditionellen“ Replikationsstrategien sind dabei eher von einem starren Gesamtsystem ausgegangen, d.h. sie wurden für ein bestimmtes Anwendungsszenario bzw. für eine bestimmte Zielsetzung entwickelt, ohne aber flexibel auf dynamische Zustandsänderungen des verteilten Systems einzugehen. „Adaptive“ Replikationsstrategien versuchen diesen Gedanken dahingehend aufzugreifen, dass die Koordination der Zugriffe dem sich dynamisch ändernden Gesamtsystem angepasst wird. Eine Anpassung kann z.B. durch Ausfall von Rechnern begründet sein, aber auch durch Überlastung eines Rechners, was zu geringerer Performance führen würde. Allgemein

gesagt kann eine Adaption auf Grund des sich ändernden Anwendungsverhaltens oder auf Grund des sich ändernden technischen Umfelds erforderlich sein. Durch die Adaption werden dann im Allgemeinen die Ziele, genauer gesagt, die Gewichtung der Ziele, verändert.

3.2. Grundlagen der Replikation

In diesem Abschnitt werden einige grundlegende Anmerkungen zu Replikation ausgeführt, wobei insbesondere eine geeignete Terminologie festgelegt wird. Zunächst werden die wichtigsten Fachbegriffe in Abschnitt 3.2.1 definiert. Anschließend wird in Abschnitt 3.2.2 auf Konsistenz und Korrektheit in replizierten Datenbanken eingegangen. Da abgeschwächte Konsistenz einen hohen Stellenwert in dieser Arbeit hat, folgt in Abschnitt 3.2.3 eine ausführliche Betrachtung von abgeschwächten Konsistenzbegriffen aus der Literatur. Abschließend werden in Abschnitt 3.2.4 Klassifizierungsmöglichkeiten diskutiert.

3.2.1. Begriffe und Definitionen

In dieser Arbeit wird, wie bereits erwähnt, ausschließlich die Replikation von Daten untersucht. Daher werden einleitend die Abstraktionsebenen von Datenbanksystemen (3-Schichten-Modell, siehe z.B. [EN02]) betrachtet. Eine Schicht in diesem Modell wird als Sicht oder Schema bezeichnet:

- Externe Sichten: Spezielle Sichten für Benutzergruppen.
- Konzeptuelle Sicht: Logische Daten und Strukturen als Abbild der realen Welt.
- Interne Sicht: Implementierung der Datenbank.

Bei Datenbanksystemen wird also insbesondere eine Unterscheidung zwischen logischer und physischer Ebene vorgenommen. Auf logischer Ebene werden die Daten der realen Welt möglichst in integrierter einheitlicher Darstellung abgebildet. Auf der physischen Ebene geht es u.a. um die Speicherung der Daten. Diese Unterscheidung ist daher auch für einzelne Datenobjekte als Teil des Gesamtbestandes der Daten sinnvoll. Somit folgen zunächst Definitionen für logische und physische Datenobjekte:

Definition 1 (nach [Len97]) ***Logisches Datenobjekt****: Ein logisches Datenobjekt beschreibt eine Menge von Aussagen zur Darstellung von Sachverhalten, die sich auf denselben Gegenstand der realen Welt beziehen. Die Menge der Aussagen, die zu einem Zeitpunkt durch das logische Objekt beschrieben werden, ist der momentane Wert des logischen Objekts. Der Wertebereich eines logischen Objekts ist die Menge der zulässigen Werte. Der momentane Wert ist immer ein Element des Wertebereichs.*

Definition 2 (nach [Len97]) ***Physisches Datenobjekt****: Ein physisches Datenobjekt ist die physische Ausprägung eines logisches Datenobjekts, d.h. die Speicherung des logischen Datenobjekts auf einem Speichermedium eines Rechners. Ein physisches Datenobjekt hat stets denselben Wertebereich wie das zugehörige logische Objekt.*

Häufig wird Replikation dahingehend erklärt, dass es sich um das mehrfache, redundante Speichern von Kopien an verschiedenen Lokalisationen handelt (siehe z.B. [BG84]). Replikation geht also mit „gewollter" Redundanz einher. Angemerkt sei, dass Redundanz nicht nur durch Replikation entsteht, sondern auch durch „abgeleitete" Daten. Mit abgeleiteten Daten sind Daten gemeint, die sich beispielsweise aus anderen Daten errechnen lassen, aber trotzdem abgespeichert werden. Mittels der Definitionen 1 und 2 werden nun die Begriffe Replikation und Replikat definiert:

Definition 3 ***Replikation****: Replikation liegt vor, wenn ein logisches Datenobjekt repliziert ist. Ein logisches Datenobjekt ist genau dann repliziert, wenn mehrere physische Datenobjekte zum logischen Datenobjekt existieren.*

Definition 4 ***Replikat**: Wenn ein logisches Datenobjekt repliziert ist, dann wird jedes physische Datenobjekt als Replikat bezeichnet.*

Ein Replikat wird oft auch einfach „Kopie“ des logischen Datenobjekts genannt. Nach [Len97] ist der Wert eines Replikats gültig, wenn er identisch ist mit dem momentanen Wert des zugehörigen logischen Datenobjekts. Das hat zur Folge, dass es bei einer Änderung des logischen Datenobjekts, also der realen Welt, erst dann wieder gültige Replikate gibt, wenn die Änderung physisch eingebracht wurde. Dieses Problem ist vergleichbar mit den Begriffsdefinitionen *„Konsistenz“* bzw. *„Korrektheit“*, die je nach Definition auch auf die reale Welt bezogen werden, aber deren Überprüfbarkeit erst bei Eingabe von Daten möglich ist (siehe Abschnitt 2.3.1).

Um den zeitlichen Versatz von der Änderung der realen Welt bis zur Aktualisierung der physischen Datenobjekte durch einen Benutzer zu umgehen, wird auf die zuletzt eingebrachte Änderung bezogen. Hierdurch werden insbesondere Konsistenzaussagen erleichtert (siehe Abschnitt 3.2.2). Man spricht in diesem Zusammenhang vom *„aktuellen Wert“* (*„current value“*). In [DGMD85] wird angemerkt, dass dieser aktuelle Wert „Sinn machen muss“, d.h. der aktuelle Wert eines Objekts beruht auf Transaktionen, die auf dieses Objekt ausgeführt wurden. Da Transaktionen der Integritätskontrolle unterliegen, ist gewährleistet, dass der aktuelle Wert bestimmten Integritätsbedingungen genügt (siehe Abschnitt 2.3.2). Mit Hilfe des aktuellen Wertes wird nun ein aktuelles Replikat definiert:

Definition 5 (nach [DGMD85]) ***Aktuelles Replikat**: Ein Replikat ist ein aktuelles Replikat oder ein Replikat ist aktuell, wenn es den aktuellen Wert repräsentiert. Der aktuelle Wert eines logischen Datenobjekts ist derjenige, der durch die letzte Transaktion, die dieses logische Datenobjekt geändert hat, geschrieben wurde.*

In einigen Fällen wird Replikation auch als Kopie eines Datenbestandes aufgefasst. Das bedeutet, dass ein „ursprünglicher“ Datenbestand besonders ausgezeichnet ist. Somit gibt es ein „Original“ und dazu Replikate, wobei es sich bei dem Original ebenfalls um eine physische Speicherung von logischen Datenobjekten handelt (siehe Definition 1). Die Nachteile dieser Sichtweise werden am Ende dieses Abschnitts diskutiert. An dieser Stelle wird dadurch auch die Frage aufgeworfen, ob mit „mehrere physische Datenobjekte“ in Definition 3 auch der Sonderfall „ein physisches Datenobjekt“ eingeschlossen ist. Auch diese Fragestellung wird am Ende des Abschnitts diskutiert. Es bleibt festzuhalten, dass die Anzahl der Replikate, nicht nur auf Grund dieser Fragestellung, eine Rolle spielt. Dadurch ist folgende Definition motiviert:

Definition 6 (nach [Len97]) ***Replikationsgrad**: Der Replikationsgrad eines logischen Datenobjekts ist die Anzahl der Replikate des logischen Datenobjekts. Der Replikationsgrad einer (verteilten) Datenbank ist der maximale Replikationsgrad aller logischen Datenobjekte.*

Mit Hilfe des Replikationsgrads wird nun definiert, wann von einer replizierten Datenbank bzw. von einer nicht-replizierten Datenbank gesprochen wird:

Definition 7 (nach [BG83b]) ***Replizierte und nicht-replizierte Datenbank**: In einer replizierten Datenbank sind einige logische Datenobjekte jeweils durch mehrere physische Datenobjekte repräsentiert, d.h. der Replikationsgrad der Datenbank ist größer als eins. In einer nicht-replizierten Datenbank (auch Ein-Kopien Datenbank) ist jedes logische Datenobjekt durch genau ein physikalisches Datenobjekt repräsentiert, d.h. der Replikationsgrad der Datenbank ist eins.*

Es wurde bereits erwähnt, dass die Replikate im Allgemeinen an unterschiedlichen Stellen in einem verteilten System lokalisiert sind. Aber in einem verteilten System müssen nicht alle Rechner zwangsläufig Replikate lokal gespeichert haben. Zur Identifizierung der Rechner, die ein Replikat lokal beherbergen, dient folgende Definition:

Definition 8 ***Rechner mit Replikat**: Ein Rechner mit Replikat ist ein Rechner, auf dem mindestens ein Replikat lokal gespeichert ist.*

An Stelle von *„Rechner mit Replikat“* wird in der Literatur auch von *„Knoten mit Replikat“* gesprochen, wobei beide Ausdrücke gleichermaßen verbreitet sind. In [NHW+02] wird *„System mit Replikat“* geschrieben. Hierbei sollte auf den speziellen Kontext des Anwendungsbereichs Systemintegration eingegangen werden: Es wurde ein Datenabgleich zwischen Anwendungssystemen mit lokalen Daten durch Replikation realisiert. Die Zugriffe auf die Daten, also auf die Replikate, erfolgt ausschließlich über die Anwendungssysteme. Obwohl dieser Sachverhalt der Replikationstransparenz (siehe Abschnitt 3.1) widerspricht, sei diese Definition hier angeführt:

Definition 9 (nach [NHW+02]) ***System mit Replikat****: Ein System mit Replikat ist ein Anwendungssystem, das jeweils genau ein Replikat eines logischen Datenobjekts kontrolliert, d.h. Zugriffe auf diese Replikate erfolgen über das Anwendungssystem.*

Um „ordnungsgemäße“ Zugriffe auf die Replikate zu gewährleisten, müssen diese Zugriffe koordiniert werden. Nach dem Gedanken der Replikationstransparenz sollte diese Koordination vor dem Benutzer versteckt sein, d.h. sie sollte beispielsweise von der Datenbank-Software übernommen werden [Rah93]. Die Aufgabe, die hierbei neben der Synchronisation der nebenläufigen Verarbeitung zu erfüllen ist, lautet vereinfacht gesagt: Lese- und Schreibzugriffe auf die Replikate sind derart zu koordinieren, dass Lesezugriffe möglichst auf aktuellen Replikaten (siehe Definition 5) durchgeführt werden und dass Schreibzugriffe möglichst alle Replikate aktualisieren. Wie tiefgreifend die Koordination erfolgen muss, liegt an dem von der Anwendung akzeptierten Korrektheitskriterium (siehe Abschnitt 3.2.2). Welcher Art die Koordination sein soll, kann durch eine geeignete Strategie festgelegt werden, was zu folgender Definition führt:

Definition 10 (nach [ASC85]) ***Replikationsstrategie****: Eine Replikationsstrategie ist für die Koordination der Zugriffe auf die Replikate verantwortlich, wobei ein bestimmtes Korrektheitskriterium verwendet wird. Dabei wird ein Schreibzugriff auf ein logisches Datenobjekt in eine Menge von Schreibzugriffen auf physikalische Datenobjekte und ein Lesezugriff auf ein logisches Datenobjekt in eine Menge von Lesezugriffen auf ein oder mehrere physikalische Datenobjekte übersetzt. Eine Replikationsstrategie ist erst bei einem Replikationsgrad größer als eins nötig.*

Statt *„Replikationsstrategie“* sind in der deutschsprachigen Literatur auch die Begriffe *„Replikationsverfahren“* oder *„Replikationsprotokoll“* üblich. In der englischsprachigen Literatur sind Begriffe wie *„replication scheme“* oder *„replication control“* gebräuchlich. Insbesondere die im Abschnitt 3.3 vorgestellten Replikationsstrategien werden häufig als Replikationsverfahren bezeichnet.

In [ASC85] wird in der Definition davon gesprochen, dass eine Replikationsstrategie die *„1-Kopien-Serialisierbarkeit“* erfüllen muss (siehe Abschnitt 3.2.2). Hier wurde die Definition dahingehend erweitert, dass auch *„abgeschwächte Korrektheitskriterien“* zulässig sind, d.h. es wird von einer Replikationsstrategie gesprochen, wenn bei der Replikation ein beliebiges Korrektheitskriterium eingesetzt wird. Zusätzlich zur Koordination der betroffenen Replikate muss eine nebenläufige Verarbeitung über eine geeignete Transaktionsverarbeitung synchronisiert werden (siehe Abschnitt 2.2), um das gewünschte Korrektheitskriterium zu erreichen. So muss bei einem starken Korrektheitskriterium wie dem der 1-Kopien-Serialisierbarkeit auf die betroffenen Replikate in einer so genannten ACID-Transaktion zugegriffen werden, während bei schwächeren Korrektheitskriterien auch schwächere Transaktionskonzepte verwendet werden können (siehe abgeschwächte Serialisierbarkeitskriterien in Abschnitt 2.2.4 bzw. erweiterte Transaktionskonzepte in Abschnitt 2.2.5).

Weiterhin wird in Definition 10 angeführt, dass eine Replikationsstrategie erst dann benötigt wird, wenn der Replikationsgrad größer als eins ist. Das ist insofern selbstverständlich, als dass bei einem Replikationsgrad von eins keine Koordination zwischen mehreren Replikaten eines logischen Datenobjekts nötig ist.

Ein wichtiges Unterscheidungsmerkmal von Replikationsstrategien ist es, ob grundsätzlich alle Replikate geändert, sprich geschrieben, werden dürfen oder ob es Replikate gibt, die nur gelesen

werden. Im letzten Fall kann die Koordination im Allgemeinen vereinfacht werden. Es muss aber sichergestellt werden, dass die Replikate nach irgendwelchen Regeln „aufgefrischt“, also aktualisiert, werden. Zur Kennzeichnung dieser Replikate dient folgende Definition:

Definition 11 ***Lese-Replikat:*** *Ein Lese-Replikat ist ein Replikat, das von einem Anwendungsprogramm nur gelesen wird. Ein Lese-Replikat wird nur in bestimmten Intervallen oder nach bestimmten Ereignissen von einem Rechner mit Replikat aktualisiert, d.h. mit einem aktuellen Replikat abgeglichen.*

In [NHW+02] wurden zwei Spezialfälle für Replikate betrachtet, deren Definitionen folgend angegeben sind:

Definition 12 ***Unikat:*** *Ein Unikat ist ein Replikat, wobei der Replikationsgrad gleich eins ist.*

Definition 13 ***Duplikat:*** *Ein Duplikat ist ein Replikat, wobei der Replikationsgrad größer als eins ist. Aber es wird keine Replikationsstrategie eingesetzt, um die Duplikate bei einer Änderung automatisch zu aktualisieren.*

Wie bereits oben erwähnt, wird bei Unikaten keine Replikationsstrategie benötigt, weil keine Koordination nötig ist. Es handelt sich also um eine nicht-replizierte Datenbank. Duplikate hingegen treten dann auf, wenn es zwar mehrere physische Datenobjekte eines logischen Datenobjekts gibt, aber Zugriffe auf diese physischen Datenobjekte nicht koordiniert werden. Es gibt also keine Replikationsstrategie, die dafür sorgt, dass „automatisch“ bei einer Änderung alle physischen Datenobjekte mit dem gleichen Wert aktualisiert werden. Die Folge ist, dass die Duplikate divergieren.

Durch die nebenläufige Verarbeitung im Mehrbenutzerbetrieb ist es möglich, dass unterschiedliche Prozesse auf das selbe physische Datenobjekt zugreifen. Handelt es sich bei diesem konkurrierenden Zugriff jeweils um Lesezugriffe, entstehen keine Probleme. Falls jedoch mindestens ein Zugriff ein Schreibzugriff ist, können unter Umständen Nebenläufigkeitsanomalien (siehe Abschnitt 2.2.3) auftreten. Durch diese konkurrierenden Zugriffe können so genannte *„Konflikte“* entstehen:

Definition 14 (nach [ES83, ACPT99]) ***Konflikt:*** *Eine Aktion a_i steht in Konflikt zu Aktion a_j, wobei $i \neq j$, wenn beide auf das selbe physische Datenobjekt zugreifen und mindestens eine der beiden Aktionen eine Schreibaktion ist. Es gibt einerseits* ***Schreib-/Lesekonflikte****, d.h. eine der beiden Aktionen ist eine Leseaktion und die andere Aktion ist eine Schreibaktion, und andererseits* ***Schreib-/Schreibkonflikte****, d.h. beide Aktionen sind Schreibaktionen.*

Zwei Transaktionen stehen demnach in Konflikt, wenn eine Aktion (Operation, Lese- bzw. Schreibzugriff) der einen Transaktion in Konflikt zu einer Aktion in einer anderen Transaktion steht. Wenn eine Transaktionsausführung das Korrektheitskriterium *„Konfliktserialisierbarkeit“* (siehe Abschnitt 2.2.4) erfüllen soll, dann müssen derartige Konflikte vermieden werden, was z.B. durch Abbruch von konfligierenden Transaktionen erreicht wird. Wenn abgeschwächte Serialisierbarkeitskriterien verwendet werden und damit mit abgeschwächter Konsistenz (siehe Abschnitt 2.3.3) gearbeitet wird, dann können ggf. die genannten Konflikte auftreten und es kommt zu den bereits erwähnten Nebenläufigkeitsanomalien (siehe Abschnitt 2.2.3).

In nicht-replizierten Datenbanken, in denen jedes logische Datenobjekt durch genau ein physisches Datenobjekt repräsentiert wird, ist somit ein Konflikt auf ein physisches Datenobjekt gleichbedeutend mit konfligierenden Zugriffen auf das entsprechende logische Datenobjekt. Wenn die Konflikte nicht vermieden werden, treten z.B. bei Schreib-/Schreibkonflikten *„lost updates“* auf. In replizierten Datenbanken sind bei Zugriffen mehrere physische Datenobjekte (Replikate) eines logischen Datenobjekts involviert. Falls die konfligierenden Aktionen unterschiedliche Replikate betreffen, können Erweiterungen zu den in Definition 14 genannten Schreib-/Lese- bzw. Schreib-/Schreibkonflikten definiert werden, die zur Unterscheidung mit dem Index R gekennzeichnet werden:

Definition 15 ***Schreib-/Lesekonflikt$_R$:*** *Eine Aktion a_i steht in Schreib-/Lesekonflikt$_R$ zu einer Aktion a_j $(i \neq j)$, wenn beide auf unterschiedliche Replikate eines logischen Datenobjekts zugreifen, eine Aktion eine Schreibaktion und die andere Aktion eine Leseaktion ist. Die Leseaktion erfolgt auf ein nicht-aktuelles Replikat.*

Definition 16 ***Schreib-/Schreibkonflikt$_R$:*** *Eine Schreibaktion a_i steht in Schreib-/Schreibkonflikt$_R$ zu einer Schreibaktion a_j $(i \neq j)$, wenn beide auf unterschiedliche Replikate eines logischen Datenobjekts mit unterschiedlichen Werten schreibend zugreifen.*

Je nach Replikationsstrategie werden diese Sonderfälle von Konflikten unter Umständen toleriert, was zu *„abgeschwächter Konsistenz“* in replizierten Datenbanken führt (vergleiche Definition 20). In diesem Fall speichern die Replikate eines logischen Datenobjekts zumindest temporär unterschiedliche Werte und die Zugriffe auf die Replikate werden nicht derart koordiniert, dass die 1-Kopien-Serialisierbarkeit gewährleistet ist. Einfach gesagt, treten die genannten Konflikte dann auf, wenn die Schreibaktionen „zeitverzögert“ bzw. „verspätet“ propagiert werden.

Bei Schreib-/Lesekonflikten$_R$ wird auf Replikate zugegriffen, die nicht aktuell sind. Wenn diese „veralteten“ Daten aus Anwendungssicht dennoch brauchbar sind, so können derartige Konflikte akzeptiert werden. Während zugelassene Schreib-/Schreibkonflikte in nicht-replizierten Datenbanken zu lost updates führen, koexistieren bei zugelassenen Schreib-/Schreibkonflikten$_R$ zwei unabhängige Änderungen auf ein logisches Datenobjekt. Bei der Zusammenführung, d.h. wenn die konfligierenden Schreibaktionen ein Replikat erreichen, muss ein Abgleich im Rahmen einer Konfliktbehandlung durchgeführt werden (siehe Abschnitt 3.3.2). Im Allgemeinen muss entweder eine überlebende Änderung oder eine resultierende Änderung bestimmt werden.

Anmerkung:
Abschließend soll in diesem Abschnitt noch diskutiert werden, ob erst dann von Replikation gesprochen wird, wenn der Replikationsgrad größer als eins ist, also bezugnehmend auf Definition 3, ob es „echt“ mehrere physische Datenobjekte zu einem logischen Datenobjekt geben muss. Nach Definition 7 liegt eine replizierte Datenbank erst dann vor, wenn zumindest für einige Objekte der Replikationsgrad größer als eins ist. Dieser Auffassung wird sich hier angeschlossen, was auch in der Definition 10 zum Ausdruck kommt: Eine Replikationsstrategie wird erst dann benötigt, wenn die Anzahl der Replikate für ein logisches Datenobjekt größer als eins ist. Die Unterscheidung, ein physisches Datenobjekt (also eine bestimmte Kopie) sei das Original und erst die weiteren Kopien sind die Replikate, wird hier nicht als sinnvoll erachtet. Zwar gibt es die Replikationsstrategie Primary Copy (siehe Abschnitt 3.3.1), bei dem eine bestimmte Kopie eine Sonderstellung einnimmt, aber bei den anderen Replikationsstrategien sind die Replikate „gleichberechtigt“. Auch im Sinne der Replikationstransparenz ist eine Unterscheidung Original und („weitere“) Replikate abwegig.

3.2.2. Konsistenz in replizierten Datenbanken

Eine wichtige Qualitätseigenschaft von Replikationsstrategien ist die Konsistenz der Replikate bzw. das Korrektheitskriterium, nach dem die Replikationsstrategie die Zugriffe auf die Replikate koordiniert. In Abschnitt 2.3 wurden die Begriffe Konsistenz und Korrektheit, die in dieser Arbeit synonym verwendet werden, im allgemeinen Datenbank-Kontext erläutert. Die allgemeinen Begriffsdefinitionen und Anmerkungen gelten ebenfalls für replizierte Datenbanken. Insbesondere müssen auch Replikate die in der Datenbank hinterlegten Integritätsbedingungen einhalten, was u.a. auch eine geeignete Transaktionsverarbeitung und damit eine Synchronisation im Mehrbenutzerbetrieb einschließt.

Bei einer nicht-replizierten Datenbank ist bei Änderung eines logischen Datenobjekts genau ein physisches Datenobjekt betroffen, für das die Integritätsbedingungen eingehalten werden müssen. Bei einer replizierten Datenbank sind bei einer Änderung eines logischen Datenobjekts mehrere physische Datenobjekte, also Replikate, betroffen, die jeweils die Integritätsbedingungen

erfüllen und mit dem gleichen Wert aktualisiert werden müssen. Welche Replikate des logischen Datenobjekts betroffen sind, ist von der gewählten Replikationsstrategie abhängig. Neben der Integritätskontrolle (siehe Abschnitt 2.3.2) ist bei replizierten Datenbanken also zusätzlich eine Replikationsstrategie nötig. Es sei angemerkt, dass in diesem Zusammenhang anstatt von Integritätskontrolle und Replikationsstrategie in der Literatur häufig von Nebenläufigkeitskontrolle und Replikationsstrategie (concurrency control und replication control [BG84]) gesprochen wird.

In ähnlicher Weise wird in [ES83] der konsistente Zustand einer replizierten Datenbank definiert: *„Eine replizierte Datenbank ist genau dann in einem konsistenten Zustand, wenn erstens alle Kopien jedes logischen Objekts den gleichen Wert haben und zweitens dieser Wert den Integritätsbedingungen genügt. In [Tho79] werden diese Anforderungen „wechselseitige Konsistenz“ (mutual consistency) und „interne Konsistenz“ (internal consistency) genannt.“* In der Literatur wird häufig der Konsistenzbegriff *„wechselseitige Konsistenz“* derart aufgefasst, dass alle Replikate eines logischen Datenobjekts den gleichen Wert repräsentieren müssen (siehe z.B. [GA02]), allerdings wird in der oben genannten Definition von [ES83] die Referenz [Tho79] falsch wiedergegeben: *„Mit „wechselseitiger Konsistenz“ meinen wir, dass alle Kopien zum gleichen Zustand konvergieren und identisch sind, wenn Änderungsoperationen unterbleiben.“* Nach dieser Definition dürfen also temporär *„Inkonsistenzen“* auftreten, d.h. Replikate eines logischen Datenobjekts beinhalten zumindest zeitweise unterschiedliche Werte.

Eine weitere Variante der Begriffsdefinition *„wechselseitiger Konsistenz“* ist in [DGMD85] zu finden: *„Alle Kopien eines logischen Datenobjekts müssen sich exakt auf einen aktuellen Wert für das logische Datenobjekt verständigen.“* Auch hier müssen also nicht alle Replikate den gleichen Wert gespeichert haben, aber es muss eine Möglichkeit geben, den aktuellen Wert (siehe Definition 5) zu identifizieren. Basierend auf diesen drei Varianten, werden die folgenden Begriffe definiert, die ebenfalls in der Literatur verbreitet sind:

Definition 17 (nach [Len97]) ***Replikationskonsistenz****: Die Replikationskonsistenz ist genau dann gewährleistet, wenn alle Replikate eines logischen Datenobjekts stets den selben Wert besitzen.*

Definition 18 (nach [BHG87]) ***1-Kopien-Serialisierbarkeit*** *(1SR, one-copy-serializability): Eine nebenläufige Ausführung von verteilten Transaktionen ist 1-Kopien-serialisierbar, wenn sie zu einer sequentiellen Ausführung dieser Transaktionen auf einer nicht-replizierten Datenbank äquivalent ist.*

Definition 19 (nach [TPST98]) ***Letztendliche Konsistenz*** *(eventual consistency): Eine replizierte Datenbank ist dann „letztendlich konsistent“, wenn alle Replikate letztendlich alle Schreiboperationen empfangen und je zwei Replikate, die die gleiche Menge an Schreiboperationen empfangen haben, einen identischen Wert repräsentieren.*

In [Len97] wird Replikationskonsistenz in globale und lokale Replikationskonsistenz unterteilt, wobei bei der globalen Replikationskonsistenz alle Replikate eines logischen Datenobjekts angesprochen sind und bei der lokalen Replikationskonsistenz ein einzelnes Replikat eines logischen Datenobjekts. Lokale Replikationskonsistenz ist gleichbedeutend damit, dass das Replikat aktuell ist (siehe Definition 5). Daher ist in dieser Arbeit bei Replikationskonsistenz grundsätzlich die globale Replikationskonsistenz gemeint und *„wechselseitige Konsistenz“* (siehe oben) wird als Replikationskonsistenz betrachtet.

Die *„1-Kopien-Serialisierbarkeit“* (1SR) ist der wichtigste Konsistenzbegriff bzw. das wichtigste Korrektheitskriterium für replizierte Datenbanken. 1SR geht mit der so genannten *„Replikationstransparenz“* einher, die nach [TGGL82] wie folgt definiert ist: *„Obgleich ein logisches Datenobjekt auf mehreren Rechnern repliziert gespeichert ist, erscheint es einem Benutzer, als wäre es nur einmal auf einem Rechner gespeichert.“* Zur Gewährleistung der 1SR-Konsistenz ist eine geeignete Integritätskontrolle inklusive Konfliktserialisierbarkeit und eine geeignete Replikationsstrategie nötig, d.h. eine Replikationsstrategie, die die Schreibzugriffe derart koordiniert,

dass Lesezugriffe immer auf das aktuelle Replikat zugreifen (vergleiche die Definition *„wechselseitige Konsistenz“* von [DGMD85], siehe oben). Die Replikationskonsistenz ist hierfür nicht nötig. Einige Replikate sind ggf. nicht aktuell, aber es wird sich auf aktuelle Replikate verständigt (siehe z.B. Votierungsverfahren in Abschnitt 3.3.1). Wenn bei Schreib- oder Lesezugriffen die Replikationsstrategie umgangen wird, kann es folglich zu inkonsistenten Zugriffen kommen.

Die Gewährleistung der *„Letztendlichen Konsistenz“* ist gleichbedeutend damit, dass alle Replikate eines logischen Datenobjekts zum selben Zustand konvergieren, falls keine weiteren Schreiboperationen auftreten. Zwischenzeitlich können Inkonsistenzen auftreten. In Definition 19 wird außerdem explizit darauf eingegangen, dass ausgelassene Schreiboperationen auch die Replikate erreichen sollten. Wenn dafür nicht Sorge getragen wird, dann findet durch die Replikationsstrategie keine *„Propagierung“* statt, d.h. es handelt sich um Duplikate (siehe Definition 13). In [Len97] wird analog zur letztendlichen Konsistenz der Konsistenzbegriff *„Konvergenz“* für Replikationsstrategien definiert, wobei zusätzlich gefordert wird, dass freigegebene Änderungen nicht rückgängig gemacht werden dürfen.

3.2.3. Abgeschwächte Konsistenzbegriffe in replizierten Datenbanken

Bei der 1-Kopien-Serialisierbarkeit wird der Vorteil der konsistenten Zugriffe durch einen hohen Synchronisationsaufwand erkauft. Die Folge ist, dass insbesondere bei Schreibzugriffen die Performance leidet und möglicherweise Transaktionen abgebrochen werden. Wenn Anwendungen auch mit „weniger“ Konsistenz zu Recht kommen, dann kann zu Gunsten der Performance und der Verfügbarkeit (siehe Abschnitt 3.1) mit so genannter *„abgeschwächter Konsistenz“* gearbeitet werden:

Definition 20 ***Abgeschwächte Konsistenz in replizierten Datenbanben***: *Abgeschwächte Konsistenz in replizierten Datenbanken liegt dann vor, wenn die 1-Kopien-Serialisierbarkeit nicht eingehalten wird. Insbesondere kann in replizierten Datenbanken eine Abschwächung der Konsistenz dadurch toleriert werden, dass die Replikate eines logischen Datenobjekts unterschiedliche Werte repräsentieren.*

In Abschnitt 2.3.3 wurde abgeschwächte Konsistenz in nicht-replizierten Datenbanken diskutiert, in denen eine Abschwächung dadurch realisiert wird, dass eingeschränkte bzw. abgeschwächte Integritätsbedingungen verwendet werden. Da die 1-Kopien-Serialisierbarkeit neben der Integritätskontrolle eine geeignete Replikationsstrategie benötigt, ist in replizierten Datenbanken zusätzlich eine Abschwächung der Konsistenz dadurch möglich, dass eine Replikationsstrategie eingesetzt wird, die temporäre Inkonsistenzen der Replikate eines logischen Datenobjekts zulässt. Dieser Fokus auf Replikation, der dieser Arbeit zugrunde liegt, wird durch den zweiten Satz der Definition 20 verdeutlicht.

Ein Beispiel für die abgeschwächte Konsistenz ist die letztendliche Konsistenz (siehe Definition 19). Weil hier kaum Aussagen darüber gemacht werden, wann genau die Konsistenz wieder erreicht wird bzw. welche Garantien zwischenzeitlich gewährleistet werden, handelt es sich um eine sehr schwache Form von Konsistenz bei replizierten Datenbanken. Angemerkt sei, dass die Verletzung der 1-Kopien-Serialisierbarkeit in Definition 20 eine notwendige Voraussetzung für die Zuordnung zur abgeschwächten Konsistenz darstellt. Votierungsverfahren (siehe Abschnitt 3.3.1), die die 1-Kopien-Serialisierbarkeit gewährleisten, lassen zwar unterschiedliche Werte von Replikaten eines logischen Datenobjekts zu, aber zählen dennoch nicht zu den Replikationsstrategien mit abgeschwächter Konsistenz.

Während die 1-Kopien-Serialisierbarkeit konsistente Zugriffe gewährleistet, sind bei der letztendlichen Konsistenz inkonsistente Zugriffe möglich, wobei Angaben über die „Güte“ der Daten nicht möglich sind. Daher wurden in der Forschung Konzepte entwickelt, die einen *„Konsistenzgrad“* (degree of consistency, siehe z.B. [Len96]) beinhalten, mittels dem einzuhaltende Garantien für Zugriffe auf Replikate spezifiziert werden können. Folgende abgeschwächte Konsistenzbegriffe werden detaillierter diskutiert:

- Session-Garantien
- Kohärenzbedingungen und Aktualisierungszeitpunkte
- Datenfrische
- Probabilistische Konsistenzbegriffe

Session-Garantien

Nach [Dat03] bedeutet Replikationstransparenz, dass die replizierte Speicherung vor dem Anwender verborgen bleibt. Bei Zugriffen auf ein logisches Datenobjekt wird durch die Replikationsstrategie bestimmt, auf welches Replikat bzw. auf welche Replikate zugegriffen wird. Damit besteht die Möglichkeit, dass ein Anwender bzw. ein Prozess bei zwei Lesezugriffen auf das gleiche logische Datenobjekt auf unterschiedliche Replikate zugreift. Wenn nun die Replikationsstrategie nicht der 1-Kopien-Serialisierbarkeit genügt und es Replikate eines logischen Datenobjekts gibt, die unterschiedliche Werte gespeichert haben, dann werden unterschiedliche Werte für ein logisches Datenobjekt gelesen. Insbesondere könnte der zweite Lesezugriff auf ein „älteres“ Replikat zugreifen als der erste Lesezugriff, d.h. auf eine ältere Version. Es ist auch möglich, dass ein Prozess den eigenen Schreibzugriff bei einem folgenden Lesezugriff nicht „sieht“, weil für den Lesezugriff ein Replikat zugewiesen wird, bei dem die Änderung noch nicht angelangt ist.

Diese Besonderheiten bei replizierten Datenbanken ergeben sich dann, wenn abgeschwächte Konsistenz verwendet wird und der Anwender bzw. „Client“ keinen Einfluss darauf hat, welche Replikate für einen Zugriff herangezogen werden. Um in diesem Fall Abhilfe zu schaffen, wurden in [TDP+94] vier so genannte Session-Garantien definiert, die auch als Client-zentrierte Konsistenz bezeichnet werden:

Definition 21 (nach [TDP+94]) ***Session-Garantie****: Eine Session-Garantie ist eine Garantie innerhalb einer Session, in der Schreib- und Lesezugriffen auf eine replizierte Datenbank mit abgeschwächter Konsistenz durchgeführt werden. Es werden vier Session-Garantien unterschieden:*

1. ***Read your Writes****: Wenn in einer Session ein Lesezugriff einem Schreibzugriff folgt und der Lesezugriff auf das Replikat r erfolgt, dann beinhaltet das Replikat r den Schreibzugriff der Session.*
2. ***Monotonic Reads****: Wenn in einer Session ein Lesezugriff* L_2 *einem Lesezugriff* L_1 *folgt,* L_1 *das Replikat* r_1 *und* L_2 *das Replikat* r_2 *liest, wobei* r_1 *und* r_2 *Replikate des selben logischen Datenobjekts sind, dann beinhaltet* r_2 *alle Schreibzugriffe von* r_1*.*
3. ***Writes follows Read****: Wenn in einer Session ein Schreibzugriff einem Lesezugriff folgt und der Schreibzugriff auf das Replikat r erfolgt, dann beinhaltet r mindestens alle Schreiboperationen, die das gelesene Replikat auch beinhaltet hat.*
4. ***Monotonic Writes****: Wenn in einer Session ein Schreibzugriff* S_2 *einem Schreibzugriff* S_1 *folgt,* S_1 *das Replikat* r_1 *und* S_2 *das Replikat* r_2 *schreibt, wobei* r_1 *und* r_2 *Replikate des selben logischen Datenobjekts sind, dann beinhaltet* r_2 *mindestens alle Schreibzugriffe von* r_1*.*

Bei der Entwicklung der Replikationsstrategie Bayou [TTP+95] wurde von einer Systemumgebung ausgegangen, bei der mobile Systeme zum Einsatz kommen. Diese mobilen Systeme verfügen nicht über eine ständige Netzanbindung und wechseln unter Umständen den Server, den sie kontaktieren. Die Session-Garantien werden in Bayou verwendet, um den mobilen Clients zumindest aus ihrer Sicht eine bestimmte Konsistenz zu gewährleisten.

Kohärenzbedingungen und Aktualisierungszeitpunkte

Wenn Lesezugriffe auf „veraltete“ Daten aus Anwendungssicht akzeptiert werden können, dann können aus dieser Kenntnis Vorteile erzielt werden: Replikate können lokal gepuffert werden (caching) und eine Transaktionssynchronisation auf diese Replikate ist nicht nötig, wenn die Transaktionen ausschließlich lesend zugreifen. Es handelt sich somit um Lese-Replikate (siehe Definition 11) und Schreib-/Lesekonflikte$_R$ (siehe Definition 15) werden toleriert. Um aber bei Lesezugriffen auf diese mitunter veralteten Daten bestimmte Garantien gewährleisten zu

können, werden die Daten nach speziellen Regeln aktualisiert bzw. als ungültig erklärt, sofern die Aktualisierung fehlschlägt.

Grundsätzlich können zwei Methoden unterschieden werden, wann die Replikate aktualisiert werden: Entweder wird ein „Abstand“ des veralteten Replikats zum aktuellen Replikat als Bedingung herangezogen oder es werden Zeitpunkte definiert, an denen eine Aktualisierung durchzuführen ist. Abstandsmaße bzw. Kohärenzbedingungen wurden in der Replikationsstrategie *„Quasi-Copies“* [ABGMA88] vorgestellt, wobei drei Typen von Bedingungen identifiziert wurden:

Definition 22 (nach [ABGMA88]) ***Kohärenzbedingung*** *(coherency condition): Eine Kohärenzbedingung ist eine Bedingung, mit der der Abstand eines „veralteten“ Lese-Replikats zu einem aktuellen Replikat spezifiziert werden kann. Eine Default-Bedingung ist, dass das Lese-Replikat irgendwann den gleichen Wert wie das aktuelle Replikat hatte. Es können folgende drei Typen von Kohärenzbedingungen unterschieden werden:*

1. ***Zeitverzug*** *(Delay Condition): Hierüber kann eine Zeitspanne angegeben werden, die ein Lese-Replikat gegenüber dem „ersten“ Änderungszeitpunkt des aktuellen Replikats in Verzug sein darf.*
2. ***Versionsabstand*** *(Version Condition): Mit einem Versionsabstand wird festgelegt, wie viele Versionen ein Lese-Replikat gegenüber einem aktuellen Replikat in Rückstand sein darf. Die Version eines Replikats ist ein Zähler, der bei einem Schreibzugriff auf das Replikat inkrementiert wird.*
3. ***Wertdifferenz*** *(Arithmetic Condition): Die Aktualisierung eines Lese-Replikats erfolgt, wenn das aktuelle Replikat außerhalb eines Wertintervalls liegt. Das Intervall kann absolut oder relativ angegeben werden, ggf. wird eine Abstandsfunktion benötigt.*

Der Versionsabstand kann als Zähler für Änderungstransaktionen aufgefasst werden. Damit ergibt sich eine Analogie zur Epsilon-Serialisierbarkeit (siehe Abschnitt 2.2.4). Während bei *„Quasi-Copies“* ein Abstandsmaß vom Lese-Replikat zum aktuellen Replikat dient, um eine Aktualisierung anzustoßen, werden hierfür bei *„Snapshots“* [AL80] ausschließlich spezielle Zeitpunkte bestimmt. Dabei gibt es zwei Varianten:

Definition 23 (nach [AL80]) ***Snapshot-Aktualisierungszeitpunkt****: Ein Snapshot-Aktualisierungszeitpunkt ist der Zeitpunkt, an dem ein Snapshot aktualisiert wird, wobei zwei Varianten unterschieden werden:*

1. ***Ereignisorientierte Aktualisierung****: Ein Snapshot wird von einem Anwender bzw. einer Anwendung auf Anforderung aktualisiert (on demand), d.h. es handelt sich um eine ereignisorientierte Aktualisierung.*
2. ***Periodische Aktualisierung****: Beim Anlegen des Snapshots wird festgelegt, nach welchen Zeitintervallen eine Aktualisierung erfolgt, d.h. es wird eine periodische Aktualisierung vorgenommen.*

Es sei angemerkt, dass ein Snapshot nicht nur aus Lese-Replikaten bestehen muss, sondern dass auch aggregierte Daten in einem Snapshot vorkommen können. Aus Datenbanksicht handelt es sich um eine materialisierte Sicht (View), d.h. um einen View, dessen Daten physikalisch gespeichert werden. Gegenüber der Implementierung von Quasi-Copies, wo die „Differenz“ von Lese-Replikaten zu den aktuellen Replikaten ermittelt werden muss, ist die Implementierung von Snapshots relativ einfach, weil die Aktualisierung unabhängig vom Zustand der Lese-Replikate zu bestimmten Zeitpunkten vorgenommen wird.

Datenfrische (Data Freshness)

In der Literatur wurde für abgeschwächte Konsistenz von Lese-Replikaten auch häufig der Begriff *„Datenfrische“* (data freshness) geprägt, wobei je nach Anwendungskontext unterschiedliche

Definitionen und Metriken eingeführt wurden. In [BP04] wird eine Taxonomie hinsichtlich verschiedener Arbeiten vorgestellt. An dieser Stelle folgt eine allgemeine Definition, die die beiden Hauptkategorien der verschiedenen Vorschläge aus der Literatur aufgreift:

Definition 24 ***Datenfrische***: *Die Datenfrische ist ein Maß für abgeschwächte Konsistenz von Lese-Replikaten, wobei verschiedene Metriken zur Messung der Konsistenz bzw. Inkonsistenz herangezogen werden können. Diese Metriken können grob in zwei Kategorien unterteilt werden:*

1. ***Abstandsmaße***: *Es wird ein Abstand von einem „veralteten“ Lese-Replikat zu einem aktuellen Replikat als Maß spezifiziert.*
2. ***Quote***: *Es wird eine Quote von aktuellen Replikaten zum gesamten Datenbestand bzw. einer Teilmenge ermittelt.*

Die erste Kategorie, die Abstandsmaße, stellen eine Analogie zu den Kohärenzbedingungen dar (siehe Definition 22), wobei je nach Anwendungsbereich auf einige der drei Typen eingeschränkt wird und/oder Erweiterungen vorgenommen werden. In [PS00] wird z.B. der Versionsabstand herangezogen, wobei zusätzlich eine Normierung vorgenommen wird, die als *„Frischegrad“* (degree of freshness) bezeichnet wird. Der Zeitverzug wird in [RBSS02] verwendet, um hierüber einen normierten *„Frischeindex“* (freshness index) zu definieren. Alle drei Typen der Kohärenzbedingungen sowie lineare Kombinationen werden in [PGV04] genutzt, wobei zusätzlich Abstandsfunktionen für komplexe Datentypen angegeben werden.

Mit der zweiten Kategorie von Definition 24, der Quote, wird ein Verhältnis von konsistenten Daten zum gesamten Datenbestand ausgedrückt. Hierbei werden im Allgemeinen die physischen Datenobjekte gezählt, die aktuell sind, und in Relation zu der Anzahl physischer Datenobjekte in der Datenbank gesetzt. In [CGM00] wird ein physisches Datenobjekt dann als *„frisch“* eingestuft, wenn es „topaktuell“ ist (up-to-date), wobei topaktuell bedeutet, dass das physische Datenobjekt der realen Welt entspricht. Die Schwierigkeiten beim Vergleich eines physischen Datenobjekts mit der realen Welt wurden bereits in Abschnitt 2.3.1 diskutiert. Daher werden nachfolgend Definitionen angegeben, die Datenfrische auf replizierte Datenbanken beziehen und auf dem aktuellen Replikat basieren (siehe Definition 5):

Definition 25 (nach [CGM00]) ***Frische eines Replikats***: *Die Frische F(r,t) eines Replikats r zum Zeitpunkt t ist definiert als:*

$$F(r,t) = \begin{cases} 1, & \textit{falls r gleich dem aktuellen Replikat zum Zeitpunkt t ist} \\ 0, & \textit{sonst} \end{cases}$$

Definition 26 (nach [CGM00]) ***Frische einer replizierten Datenbank***: *Die Frische F(D,t) einer replizierten Datenbank D mit n Replikaten zum Zeitpunkt t ist definiert als:*

$$F(D,t) = \frac{1}{n} \sum_{i=1}^{n} F(r,t)$$

In [CGM00] wird durch den Limes $t \to \infty$ des Produkts von $1/t$ und dem Integral vom Zeitpunkt 0 bis zum Zeitpunkt t über die Funktionen F(r,t) bzw. F(D,t) eine *„mittlere Frische“* für ein Replikat bzw. eine lokale Datenbank definiert. Anstatt der binären Funktion F(r,t) könnte auch jeder Typ der Kohärenzbedingungen herangezogen werden. So wird in [CGM00] neben der Frische auch das *„Alter eines Replikats“* über den Zeitverzug definiert und damit in analoger Weise das Alter einer replizierten Datenbank bzw. das mittlere Alter.

In analoger Weise zu den Definitionen 25 bzw. 26 wird in [LR03] die Frische einer Webseite bzw. die Frische eines Webrepositories definiert. Dabei setzt sich die Frische einer Webseite aus der Frische der Elemente der Webseite zusammen. In [HCH04] wird zusätzlich eine Gewichtung in die Frische-Definitionen aufgenommen, um die „Wichtigkeit“ von Webseiten berücksichtigen

zu können. Eine ähnliche Quote wird in [KSSA02] gebildet, wobei hier aber die Zugriffe bewertet werden: Es wird das Verhältnis von der Anzahl der Zugriffe auf frische Daten zu der Anzahl der gesamten Zugriffe in einem bestimmten Zeitraum gebildet. Diese Frische wird als *„bemerkte Frische“* bezeichnet:

Definition 27 (nach [KSSA02]) ***Bemerkte Frische****: Die bemerkte Frische (PF, perceived freshness) wird in Prozent gemessen und ist durch folgende Formel definiert:*

$$PF = 100 \cdot \frac{N_{fresh}}{N_{accessed}} \quad [\%]$$

mit N_{fresh}*: Anzahl Zugriffe auf aktuelle Replikate,* $N_{accessed}$*: gesamte Anzahl Zugriffe auf Replikate in einem bestimmten Zeitraum.*

Probabilistische Konsistenzbegriffe

Die Datenfrische (siehe Definition 24) kann als Verhältnis der Anzahl aktueller Replikate zu der Gesamtanzahl Replikate ausgedrückt werden bzw. bei der bemerkten Frische (siehe Definition 27) wird entsprechend das Verhältnis der Zugriffe gebildet. Ein anderer Ansatz ist es, die Wahrscheinlichkeit dafür anzugeben, dass bei einem Lesezugriff ein aktuelles Replikat gelesen wird. Da grundsätzlich ein Lesezugriff auf ein logisches Datenobjekt auf eine Menge von Lesezugriffen auf Replikate übersetzt wird (siehe Definition 10), kann eine Wahrscheinlichkeit dafür ermittelt werden, dass bei der Menge von zugegriffenen Replikaten ein aktuelles Replikat betroffen ist. Dieser Konsistenzbegriff wird zunächst allgemein definiert:

Definition 28 ***Probabilistische Konsistenz****: Die probabilistische Konsistenz ist ein wahrscheinlichkeitstheoretisches Maß dafür, dass bei einem Lesezugriff auf eine Menge von Replikaten eines logischen Datenobjekts ein aktuelles Replikat des logischen Datenobjekts gelesen wird.*

In [LPST95, LPST96] wird die probabilistische Konsistenz als Vertrauenslevel (level of confidence, LOC) bezeichnet. Bei einem Lesezugriff wird der LOC dem Wert des Replikats beigefügt, d.h. das gelesene Replikat ist mit der Wahrscheinlichkeit „LOC“ aktuell. Bei der Bestimmung des LOC wird in der genannten Arbeit von Quoren-basierten Replikationsstrategien ausgegangen (siehe Abschnitt 3.3.1). Dabei werden Schreibzugriffe mit einem vollständigen Schreibquorum durchgeführt, bei einem Lesezugriff muss jedoch kein vollständiges Lesequorum erreicht werden. Der LOC ist nun die Wahrscheinlichkeit dafür, dass bei dem unvollständigen Lesequorum, also der Menge der gelesenen Replikate, ein aktuelles Replikat enthalten ist.

In [ZJ98, ZSJ00] werden so genannte *„probabilistische Datenzugriffsgarantien“* vorgeschlagen, die hier in Abgrenzung zur Definition 28 als *„probabilistische Frische“* bezeichnet werden und für ein Replikat wie folgt definiert ist:

Definition 29 (nach [ZSJ00]) ***Probabilistische Frische****: Die probabilistische Frische* P_i^R *eines Replikats* i*, das am Rechner* R *lokalisiert ist, zum Zeitpunkt* t *ist definiert als:*

$$P_i^R(t) = \begin{cases} 1 & t \leq t_u^i + \delta \\ 0 & t \geq t_u^i + p_i + \delta \\ 1 - p_{u,-R}^i & \textit{sonst, falls zufällige Aktualisierung} \\ 1 - (t - \delta - t_u^i)/p_i & \textit{sonst, falls periodische Aktualisierung} \end{cases}$$

mit t_u^i*: Updatezeitstempel des Replikats* i*,* δ*: Latenzzeit,* p_i*: Aktualisierungsperiode,* $p_{u,-R}^i$*: Wahrscheinlichkeit, dass an einem anderen Rechner wie Rechner* R *eine Änderung des entsprechenden Replikats durchgeführt wurde.*

Das Replikat i ist bei der probabilistischen Frische genau dann konsistent, d.h. hat den Wert 1, wenn der Zeitpunkt t gegenüber dem Zeitstempel t_u^i eine Latenzzeit nicht übersteigt. Wenn der Zeitpunkt t den Zeitstempel t_u^i um die Latenzzeit und die Aktualisierungsperiode p_i übersteigt,

hat die probabilistische Frische den Wert 0. Im dazwischen liegenden Zeitraum wird bei der Berechnung der probabilistischen Frische unterschieden, ob die Replikate periodisch oder zufällig aktualisiert bzw. geändert werden.

Bei der periodischen Aktualisierung wird angenommen, dass Lese-Replikate (siehe Definition 11 auf Seite 32) periodisch von einem ausgezeichneten Rechner aktualisiert werden, an dem Änderungen vorgenommen werden dürfen. Die probabilistische Frische wird in diesem Fall im Wesentlichen als Verhältnis der gegenüber dem Zeitstempel t_u^i vergangenen Zeit und der Aktualisierungsperiode p_i gebildet. Bei der zufälligen Aktualisierung wird angenommen, dass Änderungen unabhängig an allen Rechnern vorgenommen werden können. Für die Berechnung der probabilistischen Frische wird dann die Wahrscheinlichkeit $p_{u,-R}^i$ benötigt, d.h. die Wahrscheinlichkeit, dass das Replikat i an einem anderen Rechner als Rechner R geändert wurde. Die Berechnung der $p_{u,-R}^i$ stellt im Allgemeinen ein größeres Problem dar. In [ZSJ00] wird $p_{u,-R}^i$ über $p_{u,r}^i$ berechnet, wobei $p_{u,r}^i$ die Wahrscheinlichkeit dafür ist, dass mindestens eine Änderung des Replikats i am Rechner $r \neq R$ im Zeitintervall $[t_u^i + \delta, t]$ vorgenommen wurde. Für $p_{u,r}^i$ wird eine exponentielle Wahrscheinlichkeitsdichtefunktion angenommen.

3.2.4. Klassifizierungen

In diesem Abschnitt werden Klassifizierungsmöglichkeiten von Replikationsstrategien gezeigt, die einerseits zum Verständnis der Funktionsweise von Replikationsstrategien beitragen und andererseits eine Basis für die strukturierte Vorstellung einzelner Replikationsstrategien in Abschnitt 3.3 bieten. In [War05] werden sechs Dimensionen vorgestellt, nach denen eine Klassifizierung von Replikationsstrategien vorgenommen werden kann, wobei die Dimensionen nicht absolut orthogonal zueinander sind:

D1 Korrektheitskriterium (siehe Abschnitt 3.2.2): Replikationsstrategien können dahin gehend klassifiziert werden, welcher Konsistenzbegriff bzw. welches Korrektheitskriterium ihnen zugrunde liegt.

D2 Verteilung der Replikate: Es kann differenziert werden, wie viele Replikate angelegt und wo sie lokalisiert werden. Zusätzlich ist eine Unterscheidung möglich, ob alle logischen Datenobjekte repliziert werden oder eine Teilmenge bzw. ob logische Datenobjekte fragmentiert werden und nur bestimmte Fragmente repliziert werden.

D3 Verteilung der Updates: Zunächst kann zwischen eifriger und träger Replikation bzw. synchroner und asynchroner Replikation getrennt werden (siehe unten). Weiterhin kann die Verteilung in direkte und indirekte Verteilung zerlegt werden. Bei der indirekten Verteilung kann ein so genannter *„Propagierungsgraph“* oder eine *„epidemische Kommunikation“* [AAS97] genutzt werden. Auch hinsichtlich der betroffenen Replikate kann unterteilt werden, nämlich ob alle Replikate betroffen sind oder jeweils ein bestimmtes *„Quorum“* (siehe Abschnitt 3.3.1).

D4 Grad der Zentralisierung: Hiermit wird unterschieden, auf welche Replikate bei Schreiboperationen zugegriffen wird. Es ergibt sich eine Unterscheidung in *„primary copy“* und *„update everywhere“*.

D5 Reaktion auf Partitionierung: Es erfolgt eine Abgrenzung zwischen *„pessimistischen Strategien“* und *„optimistischen Strategien“*. Die pessimistischen Strategien schränken im Fall einer Netzpartitionierung die Verfügbarkeit derart ein, dass weiterhin Konsistenz gewährleistet wird. Im Allgemeinen sind damit Schreibzugriffe bestenfalls in einer Partition zulässig. Die optimistischen Strategien schränken nicht die Verfügbarkeit ein. Die Folge ist, dass bei Behebung der Netzpartitionierung eine Zusammenführung (siehe Abschnitt 3.3.2) erfolgen muss.

D6 Synchronisation der Zugriffsoperationen: Hierdurch wird eine Klassifizierung nach *„syntaktischen Strategien“* und *„semantischen Strategien“* möglich. Die Einteilung wird nach [DGMD85] wie folgt vorgenommen: *„Syntaktische Strategien verwenden als Korrektheits-*

kriterium die 1-Kopien-Serialisierbarkeit, während semantische Strategien die Semantik der Transaktionen oder der Datenbank nutzen. Letzteres kann auch als „Sonstiges“ aufgefasst werden.“

In [CHKS91] werden zur Klassifizierung von Replikationsstrategien die Dimensionen D4, D3 (feingranularere Unterteilung als eifrige und träge Replikation) und, eingeschränkt, D1 verwendet. Im Kontext des Anwendungsbereichs mobiler Systeme wird in [GHOS96] eine Klassifizierung bzgl. der Dimensionen D3 (eifrige und träge Replikation) und D4 (Master und Group) vorgeschlagen. Für Replikationsstrategien, die zur eifrigen Replikation (D3) zählen, wird in [WSP+00] eine Unterteilung nach folgenden Dimensionen vorgenommen: D4, D3 (constant und linar interaction) und *„Transaction Termination“*. Ein Überblick inklusive Klassifizierung für optimistische Replikation (beinhaltet die träge Replikation) ist in [SS05] zu finden, wobei im Wesentlichen alle sechs Dimensionen genutzt werden.

Eine häufig referenzierte Klassifizierung wurde in [DGMD85] vorgestellt, wobei die Dimensionen D6 und D5 verwendet werden. Diese Klassifizierung wurde in [BD96] aufgegriffen und hinsichtlich der Einordnung von Replikationsstrategien genutzt. Die Kombination der Dimensionen D6 und D5 führt zu syntaktisch-pessimistischen, syntaktisch-optimistischen, semantisch-pessimistischen und semantisch-optimistischen Replikationsstrategien. Problematisch sind die syntaktisch-optimistischen Strategien, weil einerseits die 1-Kopien-Serialisierbarkeit erhalten bleiben soll, aber andererseits keine Einschränkung der Verfügbarkeit bei Netzpartitionierung erfolgen darf. In [Dav84], wo das *„optimistische Protokoll“* als syntaktisch-optimistische Replikationsstrategie vorgestellt wird, wird das Commit der Transaktionen bis zur Reparatur der Partitionierung verzögert, um ggf. konfligierende Transaktionen zurückzusetzen.

In [GHOS96] wird Replikation in *„eifrige Replikation“* und *„träge Replikation“* unterteilt (eager and lazy replication), wobei folgende Definition zugrunde gelegt wird:

Eifrige Replikationsstrategien: Eifrige Replikationsstrategien sind diejenigen, die alle Replikate in einer originären Transaktion aktualisieren.

Träge Replikationsstrategien: Träge Replikationsstrategien sind diejenigen, bei der ein Replikat in der originären Transaktion geändert wird und die übrigen Replikate asynchron aktualisiert werden.

Unter einer *„originären Transaktion“* wird eine Transaktion verstanden, die den ACID-Eigenschaften [HR83] genügt, also insbesondere atomar ist und isoliert ausgeführt wird. An der obigen Einteilung ist zu kritisieren, ob es nicht auch Mischformen geben kann bzw. wo diese zuzuordnen sind. Eine Strategie, die beispielsweise eine bestimmte Anzahl größer als eins in der originären Transaktion und den Rest asynchron aktualisiert, ist nicht eindeutig zu klassifizieren. Um alle Replikationsstrategien eindeutig zuordnen zu können, erscheint es besser, für eifrige bzw. synchrone Replikation eine Definition zu geben und den „Rest“ als träge bzw. asynchrone Replikation zu bezeichnen. Weiterhin kann festgestellt werden, dass nach [GHOS96] von eifriger Replikation nur dann gesprochen wird, wenn alle(!) Replikate beteiligt sind. Damit würde nur die ROWA-Strategie (siehe Abschnitt 3.3.1) zur eifrigen Replikation zählen.

Eine Klassifizierung in synchrone und asynchrone Replikationsstrategien wird in [PL91] vorgenommen. Zu den synchronen Replikationsstrategien zählen diejenigen, die in einer atomaren Transaktion „einige Replikate“ aktualisieren, um wechselseitige Konsistenz zu erreichen (siehe Abschnitt 3.2.2). Auch hier bestehen kleine Ungenauigkeiten in der Einteilung, da „wechselseitige Konsistenz“ und „einige Replikate“ nur vage festgelegt sind. Daher wird in [NHHT03], wo ebenfalls nach synchronen und asynchronen Replikationsstrategien klassifiziert wird, von den „notwendigen Replikaten“ gesprochen, wobei über „notwendig“ die 1-Kopien-Serialisierbarkeit (siehe Definition 18) erreicht werden soll.

Auch in der D6 erfolgt die Unterteilung in syntaktische und semantische Replikationsstrategien über die 1-Kopien-Serialisierbarkeit. Für die 1-Kopien-Serialisierbarkeit ist nach [HAA99, WSP+00] eine eifrige Replikation notwendig und die eifrige Replikation wird gleich der syn-

chronen Replikation gesetzt. Hiermit wird nochmals die Wichtigkeit des Konsistenzbegriffs bzw. des Korrektheitskriteriums 1-Kopien-Serialisierbarkeit unterstrichen, die damit prädestiniert ist für eine Klassifizierung von Replikationsstrategien. Wenn die Definitionen der Klassifizierungen syntaktisch / semantisch, eifrig / träge, pessimistisch / optimistisch bzw. synchron / asynchron über dieses Kriterium erfolgen, so führen sie zu gleichen Einteilungen und können synonym verwendet werden. Da für die 1-Kopien-Serialisierbarkeit eine Synchronisation nebenläufiger Transaktionen und damit letztendlich ein geeignetes Kommunikationsmodell benötigt wird (siehe Abschnitt 2.1.2), wird in dieser Arbeit analog zur Klassifizierung von synchroner und asynchroner Kommunikation im Weiteren nach synchroner und asynchroner Replikation klassifiziert:

Definition 30 ***Synchrone Replikationsstrategie****: Synchrone Replikationsstrategien sind diejenigen, die die 1-Kopien-Serialisierbarkeit (1SR) gewährleisten.*

Definition 31 ***Asynchrone Replikationsstrategie****: Asynchrone Replikationsstrategien sind diejenigen, die die 1-Kopien-Serialisierbarkeit (1SR) nicht gewährleisten.*

Mittels der Definitionen 30 und 31 können Replikationsstrategien somit zunächst in zwei Klassen unterteilt werden. Innerhalb einer Klasse können dann die oben aufgeführten Dimensionen für eine weitere Gliederung verwendet werden. Auf diese Weise wird in Abschnitt 3.3 eine detaillierte Klassifizierung der in dieser Arbeit vorgestellten Replikationsstrategien vorgenommen.

3.3. Traditionelle Replikationsstrategien

Um die adaptiven Replikationsstrategien, die in Abschnitt 3.4.2 vorgestellt werden, als verwandte Arbeiten besser einordnen zu können, werden in diesem Abschnitt „traditionelle“ Replikationsstrategien kurz präsentiert. Ausführliche Überblicke sind z.B. in [DGMD85, CHKS91, BD96, Len97, WSP$^+$00, SS05] zu finden. Mit „traditionelle“ Replikationsstrategien sind an dieser Stelle eher ältere Replikationsstrategien gemeint, bei denen Dynamik und Adaption keine bzw. eine untergeordnete Rolle spielen. Weiterhin wird angenommen (siehe Abschnitt 4.1.1), dass eine vollreplizierte Datenbank vorliegt, d.h. die Rechner mit Replikat sind bekannt und speichern jeweils alle Replikate. Daher werden in dieser Arbeit keine Replikationsstrategien betrachtet, die sich mit der Lokalisation von Replikaten beschäftigen. Hierfür siehe z.B. in [WJH97, LA00, CN01, RF02].

Wie bereits in Abschnitt 3.2.4 erwähnt, gibt es unterschiedliche Möglichkeiten der Klassifizierung von Replikationsstrategien, die sich in den oben genannten Überblicken widerspiegeln. Gemäß den Definitionen 30 und 31 erfolgt hier zunächst eine Unterteilung in synchrone Replikationsstrategien, die in Abschnitt 3.3.1 beschrieben werden, und in asynchrone Replikationsstrategien, die in Abschnitt 3.3.2 erörtert werden. In Abbildung 3.2 wird diese erste Aufteilung in einer zweiten Stufe weiter untergliedert und für jede Klasse zwei Beispiele angegeben.

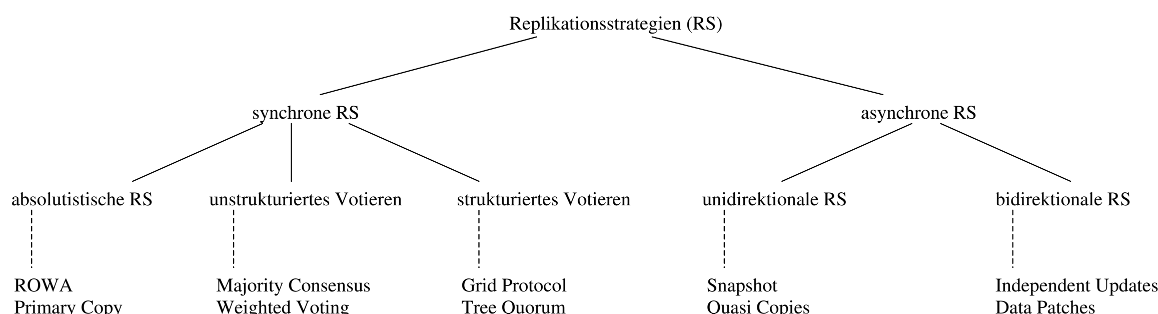

Abbildung 3.2.: Klassifizierung von Replikationsstrategien mit jeweils zwei Beispielen

3.3.1. Synchrone Replikationsstrategien

Die synchronen Replikationsstrategien (siehe Definition 30 auf Seite 42) nach der hier verwendeten Klassifizierung gewährleisten Konsistenz nach der 1-Kopien-Serialisierbarkeit (1SR, siehe Abschnitt 3.2.2), d.h. Zugriffe müssen derart koordiniert werden, dass Schreib-/Lesekonflikte$_R$ und Schreib-/Schreibkonflikte$_R$ (siehe Definitionen 15 und 16 auf Seite 33) vermieden werden. Eine derartige Koordination bedeutet Aufwand, der entweder zu Lasten der Schreib- und/oder der Lesezugriffe geht. Die Koordination erfolgt derart, dass bei Zugriffen die involvierten Replikate entweder durch so genannte *„absolutistische Replikationsstrategien“* festgelegt werden oder durch Abstimmung bei den so genannten *„Votierungsstrategien“* ermittelt werden. Gemäß der Abbildung 3.2 werden die synchronen Replikationsstrategien somit wie folgt unterteilt:

- Absolutistische Replikationsstrategien
- Unstrukturierte Votierungsstrategien
- Strukturierte Votierungsstrategien

Neben synchronen Replikationsstrategien, die eindeutig einer der oben genannten Gruppen zugeordnet werden können, wurden auch so genannte *„Mischverfahren“* entwickelt. Diese Verfahren reagieren auf Rechnerausfälle durch Wechsel der Replikationsstrategie und werden in Abschnitt 3.4.2 erläutert.

Absolutistische Replikationsstrategien

Bei absolutistichen Replikationsstrategien wird vorab festgelegt, welche Replikate bei Schreibzugriffen von einem Anwendungsprogramm geändert werden. Anschließend sorgt die Replikationsstrategie ggf. für eine Propagierung der Änderung an die weiteren Replikate. Lesezugriffe können im Allgemeinen an jedem Replikat durchgeführt werden. Hierbei gibt es im Wesentlichen zwei Varianten, nämlich dass möglichst alle Replikate geschrieben werden oder eine ausgezeichnete Stelle.

Bei der Replikationsstrategie **Read One Write All** (**ROWA**, [TGGL82]) müssen bei einem Schreibzugriff alle Replikate geändert werden. Dadurch wird Replikationskonsistenz (siehe Definition 17 auf Seite 34) erreicht, d.h. alle Replikate haben stets den gleichen, aktuellen Wert. Daher ist bei einem Lesezugriff der Zugriff auf jedes beliebige Replikat zulässig. Bei steigendem Replikationsgrad sinkt allerdings die Schreibverfügbarkeit, weil selbst bei Ausfall nur eines Rechners, auf dem ein zu änderndes Replikat gespeichert ist, die Schreiboperation nicht durchgeführt wird. Daher eignet sich die ROWA-Strategie eher bei niedrigem Replikationsgrad oder dann, wenn Schreibzugriffe im Vergleich zu Lesezugriffen selten vorkommen. Um die Schreibverfügbarkeit zu erhöhen, wurde die Replikationsstrategie **Read One Write All Available** (**ROWA-A**, [BG84]) als Erweiterung der ROWA-Strategie entwickelt. Bei Schreibzugriffen werden im Gegensatz zur ROWA-Strategie nicht alle Replikate benötigt, sondern nur diejenigen, die verfügbar sind. Somit sind die Replikate, die nicht erreicht werden konnten, nicht mehr aktuell. Daher muss ein Rechner, der zwischenzeitlich ausgefallen ist, bei Wiederanlauf „seine“ Replikate auf den aktuellen Stand bringen. Ein Problem bei ROWA-A ist, dass zwischen Rechnerausfall und Kommunikationsfehler (Netzpartitionierung) unterschieden werden muss, weil im Falle von Netzpartitionierung die 1-Kopien-Serialisierbarkeit nicht gewährleistet werden kann.

Bei den Replikationsstrategien **Primary-Site** [AD76] bzw. **Primary-Copy** [Sto79] wird jeweils ein Replikat für jedes logische Objekt zur so genannten Primärkopie, also zur ausgezeichneten Stelle, bestimmt. Alle Schreibzugriffe eines Anwendungsprogramms werden auf den Primärkopien durchgeführt, wodurch Schreib-/Schreibkonflikte$_R$ erkannt und vermieden werden. Die Primärkopie repräsentiert immer den aktuellen Wert und die Aktualisierung der übrigen Replikate, die nicht Primärkopie sind, erfolgt asynchron, ausgehend von dem Rechner, auf dem die Primärkopie gespeichert ist. Damit nun die 1-Kopien-Serialisierbarkeit gewährleistet wird und damit die Zuordnung zu den synchronen Replikationsstrategien gerechtfertigt ist, müssen auch Schreib-/Lesekonflikte$_R$ vermieden werden, wozu Sperren verwendet werden: Entweder werden

bei einem Schreibzugriff auf die Primärkopie alle übrigen Replikate bis zur endgültigen Aktualisierung gesperrt oder aber bei einem Lesezugriff wird die Primärkopie gegen Schreibzugriffe gesperrt. Wenn auf Sperren verzichtet wird, dann kann die 1-Kopien-Serialisierbarkeit nicht garantiert werden und es können veraltete Replikate gelesen werden. Dann gehört Primary Copy zu den asynchronen Replikationsstrategien. Insbesondere bei kommerziellen Datenbanken wird beim Primary Copy häufig von **Master/Slave-Replikation** [Len96] gesprochen.

Durch die Verwendung von Sperren wird beim Primary Copy zwar das Korrektheitskriterium 1SR gewährleistet, aber entweder muss analog zur ROWA-Strategie bei einem Schreibzugriff auf alle Rechner mit Replikat zugegriffen werden (wenn auch nur, um Sperren zu setzen) oder bei jedem Lesezugriff muss auch der Rechner mit der Primärkopie einbezogen werden, womit sich der Vorteil der Replikation zumindest schmälert. Als Vorteil bleibt insbesondere eine Erhöhung der Verfügbarkeit, wenn bei Ausfall des Rechners mit den Primärkopien auf einen anderen Rechner ausgewichen werden kann. Eine Erweiterung zum Primary Copy ergibt sich dadurch, dass die Lokalisation der Primärkopie sich dynamisch ändern kann. In diesem Fall muss die Primärkopie gekennzeichnet sein, z.B. durch ein so genanntes *„Token“*. Ein Beispiel hierfür ist die Replikationsstrategie **True Copy Token** [MW82].

Unstrukturierte Votierungsstrategien

Bei *„Votierungsstrategien“* wird darüber abgestimmt, ob ein Zugriff gültig ist oder nicht. Ein Zugriff ist genau dann gültig, wenn ein bestimmtes Quorum an Stimmen erreicht ist. Dabei wird zwischen Schreibquoren bei Schreibzugriffen und Lesequoren bei Lesezugriffen unterschieden. Ein Replikat kann eine oder mehrere Stimmen erhalten, z.B. bei Gewichtung. Gegenüber den absolutistischen Replikationsstrategien bieten die Votierungsstrategien eine erhöhte Schreibverfügbarkeit selbst bei Netzpartitionierung, weil im Allgemeinen nicht alle Replikate geschrieben werden, d.h. aber auch, dass es veraltete Replikate gibt. Daher ist die Leseverfügbarkeit im Allgemeinen niedriger, weil bei Lesezugriffen mehrere Replikate gelesen werden müssen. Zusätzlich werden Zeitstempel oder Versionsvektoren benötigt (siehe Abschnitt 2.1.1), um innerhalb der gelesenen Replikate das Replikat mit dem aktuellen Wert zu identifizieren. Damit Votierungsstrategien die 1-Kopien-Serialisierbarkeit gewährleisten, müssen so genannte Überschneidungsregeln eingehalten werden. Sei r das Lesequorum, w das Schreibquorum und s die Summe aller verfügbaren Stimmen, dann muss gelten:

$$2 * w > s \tag{3.1}$$

$$w + r > s \tag{3.2}$$

Mit der Bedingung 3.1 wird gewährleistet, dass zwei beliebige Schreibquoren mindestens ein Replikat gemeinsam enthalten, woran konfligierende Schreibzugriffe erkannt werden können, d.h. Schreib-/Schreibkonflikte$_R$ können vermieden werden. Schreib-/Lesekonflikte$_R$ werden anhand der Bedingung 3.2 erkannt, weil dadurch gewährleistet wird, dass mindestens ein gemeinsames Replikat in einem Lese- und einem Schreibquorum liegt. Allgemein gesagt, müssen sich also je zwei Quoren überlappen, d.h. mindestens ein gemeinsames Replikat enthalten.

Bei den *„unstrukturierten Votierungsstrategien“* werden die Replikate nicht wie bei den *„strukturierten Votierungsstrategien“* in irgendeiner logischen Struktur angeordnet. Durch eine logische Struktur kann die Menge der benötigten Replikate bei einem Zugriff, also die Quorengröße, reduziert werden (siehe unten). Nachfolgend werden einige unstrukturierte Votierungsstrategien grob erläutert. Für detaillierte Beschreibungen sei auf die genannten Quellen verwiesen.

Die wohl einfachste Votierungsstrategie ist die Strategie **Majority Consensus** [Tho79]. Alle Replikate eines logischen Objekts sind gleichberechtigt, d.h. sie erhalten genau eine Stimme. Ein Schreib- oder Lesezugriff ist genau dann gültig, wenn die „einfache“ Mehrheit der Replikate erreicht wird. Wenn also der Replikationsgrad des logischen Objekts n ist, dann müssen bei einem Zugriff $[\frac{n}{2}]+1$ Replikate erreicht werden. Mit $[\frac{n}{2}]$ ist ganzzahliges Teilen gemeint. Schreib-

und Lesequoren sind also gleich groß. Die oben genannten Überschneidungsregeln werden durch die Mehrheitsbildung erfüllt.

Beim **Weigthed Voting** [Gif79], einer Verallgemeinerung der Votierungsstrategie Majority Consensus, werden jedem Replikat eine oder mehrere Stimmen zugeordnet. Dadurch ist eine Gewichtung möglich, um z.B. einem Rechner mit hoher Verfügbarkeit ein höheres Gewicht zu vergeben. Weiterhin ist nicht die Mehrheit bei Zugriffen nötig, aber die Quoren müssen so gebildet werden, dass die oben genannten Überschneidungsregeln eingehalten werden, z.B. könnte bei 9 Gesamtstimmen das Schreibquorum aus 7 Stimmen bestehen und das Lesequorum aus drei Stimmen. Hierdurch kann unterschiedlichen Schreib-/Leseraten Rechnung getragen werden. Angemerkt sei, dass die Replikationsstrategie ROWA als Spezialfall des gewichteten Votierens aufgefasst werden kann: Die Replikate erhalten jeweils eine Stimme. Das Schreibquorum besteht dann aus allen Replikaten und das Lesequorum aus einem Replikat.

Um die Verfügbarkeit zu erhöhen bzw. Speicherressourcen zu sparen, wurde die Strategie Weighted Voting dahingehend erweitert, dass Rechner ohne Replikat eine Stimme erhalten, z.B. beim **Voting with ghosts** (Votieren mit Geistern, [vRT88]), beim **Voting with bystanders** (Votieren mit Zuschauern, [Par89]) oder beim **Voting with witnesses** (Votieren mit Zeugen, [Par90]). Eine universellere Strategie als Weighted Voting ist die Verwendung von einer so genannten **Coterie** [GMB85]. Eine Coterie ist eine Teilmenge der Potenzmenge der Replikate eines logischen Objekts, die folgende Bedingungen erfüllt: (1) Eine Coterie enthält nichtleere Mengen. (2) Die Schnittmenge zweier beliebiger Mengen, die in einer Coterie enthalten sind, ist nicht leer. (3) Keine Menge, die in der Coterie enthalten ist, ist eine echte Teilmenge einer anderen Menge, die auch in der Coterie enthalten ist. In [The93a] wird gezeigt, dass Coterien gebildet werden können, die nicht durch Stimmenvergabe beim Weigthed Voting nachgebildet werden können, aber umgekehrt jede Stimmenvergabe durch Coterien. Dagegen sind laut [The93a] Coterien und **Multidimensional voting** [AAC91], bei dem Vektoren anstatt eines Skalars für die Stimmenanzahl verwendet wird, gleichuniversell. In [The93b] wird das verallgemeinerte, mehrstufige Votierungsverfahren **General Structured Voting** (GSV) vorgestellt, mit dem verschiedene Votierungsstrategien, z.B. auch die hier vorgestellten, abgebildet werden können.

Als Reaktion auf Rechnerausfälle kann zur Steigerung der Verfügbarkeit entweder die Quorengröße angepasst werden oder die Votierungsstrategie geändert werden. Letzteres wird bei den adaptiven Replikationsstrategien in Abschnitt 3.4.2 beschrieben. Eine Änderung der Quorengröße ist dann sinnvoll, wenn entweder mehrere Rechner mit Replikat ausgefallen sind und damit z.B. kein Schreibquorum mehr erreicht werden kann oder wenn neue Rechner mit Replikat hinzukommen, z.B. auch bei Wiederanlauf ausgefallener Rechner. Beispiele für dynamische Votierungsstrategien sind **Dynamic Vote Reassignment** [BGMS86], **Dynamic Weighted Voting** [Dav89], **Dynamic Voting** [JM87], **Dynamic Linear Voting** [JM90] und **dynamic General Structured Voting** (dGSV, [TS99]). Bei einem Wechsel der Quorengröße muss besondere Sorgfalt an den Tag gelegt werden, um die 1-Kopien-Serialisierbarkeit zu gewährleisten. In [RL93] wurden so genannte *„Epochen“* eingeführt. Hierbei handelt es sich um eine Ganzzahl, die bei einem Epochenwechsel, d.h. einem Wechsel der Quorengröße, inkrementiert wird. Bei einem Wechsel ist die Vereinigung eines alten und neuen Schreibquorums nötig.

Strukturierte Votierungsstrategien

Durch Anordnung der Rechner mit Replikat in eine logische Struktur kann, wie bereits erwähnt, bei den strukturierten Votierungsverfahren die Menge der Rechner mit Replikat bei Zugriffen gegenüber den unstrukturierten Votierungsstrategien reduziert werden. Als logische Strukturen werden Gitter- oder Baumstrukturen verwendet. Der Nachteil besteht im erhöhten Aufwand und die Einschränkung auf vorgegebene Kombinationen bei Zugriffen. Beim **Grid Protocol** [CAA90, CAA92] werden die Rechner mit Replikat logisch in einem Gitter als $n\ x\ m$ - Matrix angeordnet. Die Überlappungsregeln (siehe oben) werden dadurch gewährleistet, dass bei einem Schreibzugriff eine Spalte komplett und jeweils ein Replikat der anderen Spalten geschrieben wird und bei einem Lesezugriff jeweils ein Replikat einer Spalte gelesen wird. Gegenüber dem

Majority Consensus, bei dem bei 16 Rechnern mit Replikat auf 9 Replikate zugegriffen werden muss, werden beim Grid Protocol mit einer 4 x 4 - Matrix nur 7 Replikate für einen Schreibzugriff und 4 Replikate für einen Lesezugriff benötigt.

Eine Baumstruktur wird von der Votierungsstrategie **Tree Quorum Protocol** [AA90] bzw. **Generalized Tree Quorum Protocol** [AA92a] genutzt. Hierbei ist die Idee, dass bei Ausfall eines Vaterknotens auf die Mehrzahl der direkten Kinderknoten zugegriffen wird. Die Überlappungsregeln hängen hier von der Höhe h des Baumes und der Anzahl v direkter Kinder der Knoten ab, d.h. die oben genannten Regeln 3.1 und 3.2 müssen jeweils für h und v an Stelle der Gesamtstimmen s gelten. Eine Variante zum Tree Quorum Protocol ist **Hierarchical Quorum Consensus** [Kum91], bei dem die Rechner mit Replikat ausschließlich an den Blättern liegen. Hierdurch wird unter anderem bei Rechnerausfall von „wurzelnahen“ Rechnern, also der höher priorisierten Rechner, geringerer Kommunikationsaufwand erreicht.

Auch bei den strukturierten Votierungsverfahren wurden dynamische Varianten vorgeschlagen, um bei Rechnerausfällen effizientere Zugriffe zu ermöglichen. In [AA92b, AA96] wurde die Rekonfiguration des Grid Protocols und des Tree Quorum Protocols vorgestellt, z.B. im Protokoll **Reconfigurable Tree Quorum**. Im **dynamic Grid Protocol** [RL92] können Stellen im Gitter nicht besetzt werden, sodass auch mit Rechneranzahlen gearbeitet werden kann, die ansonsten nicht in eine n x m - Matrix passen würden.

3.3.2. Asynchrone Replikationsstrategien

Die asynchronen Replikationsstrategien (siehe Definition 31 auf Seite 42) gewährleisten nicht die 1-Kopien-Serialisierbarkeit und nehmen damit zumindest temporär Inkonsistenzen in Kauf, um sowohl die Verfügbarkeit als auch die Performance von Zugriffen zu erhöhen (siehe Abbildung 3.1 auf Seite 28). Somit können Schreib-/Lesekonflikte$_R$ und/oder Schreib-/Schreibkonflikte$_R$ auftreten (siehe Definitionen 15 und 16 auf Seite 33), wobei Schreib-/Lesekonflikte$_R$ im Allgemeinen toleriert werden. Zwar werden dann „veraltete“ Daten gelesen, die aber dennoch als brauchbar angesehen werden. Schreib-/Schreibkonflikte$_R$ müssen im Allgemeinen behandelt werden, damit die Replikate nicht divergieren und zumindest das Korrektheitskriterium *„Letztendliche Konsistenz“* (siehe Definition 19 auf Seite 34) erfüllt wird.

In Abbildung 3.3 sind die Aufgaben eines Konfliktmanagements als UML-Diagramm dargestellt. Ein Konfliktmanagement beinhaltet eine Konfliktvermeidung, eine Konflikterkennung und eine Konfliktbehandlung. Wie bereits erwähnt, vermeiden die synchronen Replikationsstrategien Konflikte (siehe Abschnitt 3.3.1). Bei asynchronen Replikationsstrategien müssen zunächst Konflikte erkannt werden, worauf optional eine Konfliktbehandlung initiiert wird. Die Konfliktbehandlung kann entweder automatisch, z.B. durch Bestimmung einer überlebenden oder resultierenden Änderung, oder manuell, z.B. durch einen Administrator, erfolgen.

Schreib-/Schreibkonflikte$_R$ können grundsätzlich an den Daten selbst, z.B. durch Mitführen des Vorher-Zustandes (before image), oder an Zeitstempeln bzw. Versionsvektoren (siehe Ab-

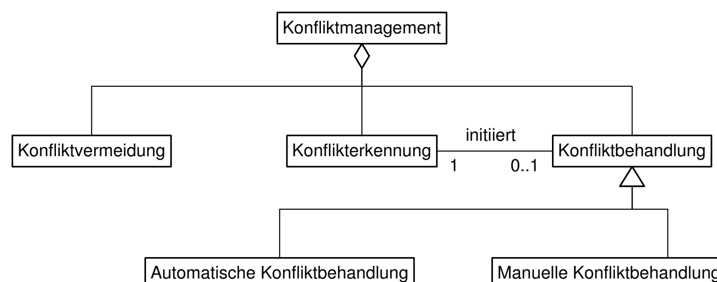

Abbildung 3.3.: Aufgaben des Konfliktmanagements als UML-Klassendiagramm

schnitt 2.1.1) erkannt werden. Weil für die Konflikterkennung im Allgemeinen eine logische Reihenfolge genügt, werden in klassischen Verfahren Versionsvektoren verwendet, siehe z.B. [PPR83, Bre83]. Ein weiteres klassisches Verfahren ist das **optimistische Protokoll** [Dav84], in dem ein so genannter *„precedence graph“* gepflegt wird, mittels dem partitionsbezogene Konflikte erkannt und beseitigt werden können. Die Konfliktbehandlung ist häufig speziell für das Anwendungsgebiet, für die eine Replikationsstrategie entwickelt wurde, angepasst und wird unten bei den vorgestellten Beispielen diskutiert.

Ein wichtiges Unterscheidungsmerkmal von asynchronen Replikationsstrategien ist es, ob Schreib-/Schreibkonflikte$_R$ auftreten können oder nicht. Im Allgemeinen erfolgt die Vermeidung von Schreib-/Schreibkonflikten$_R$ dadurch, dass Schreibzugriffe von Anwendungsprogrammen auf ein ausgezeichnetes Replikat, der Primärkopie bzw. dem Master, wie bei der absolutistischen Replikationsstrategie Primary Copy ausgeführt werden (siehe Abschnitt 3.3.1). Von dem Master werden Änderungen zu den anderen Replikaten, den Slaves, propagiert, wobei bei der asynchronen Variante auf Sperren verzichtet wird, sodass die 1-Kopien-Serialisierbarkeit nicht gewährleistet ist. Schreib-/Schreibkonflikte$_R$ treten dann auf, wenn Schreibzugriffe gleichzeitig an mehreren Replikaten ausgeführt werden. Als Unterteilung der asynchronen Replikationsstrategien ergibt sich somit Master versus Group bzw. Everywhere [GHOS96] oder Single-Master versus Multi-Master [SS05]. Hier wird nachfolgende Bezeichnung gewählt, die eher in kommerziellen Systemen verwendet wird und in Abbildung 3.2 auf Seite 42 mit jeweils zwei Beispielen dargestellt ist:

- Unidirektionale Replikationsstrategien (entspricht im Allgemeinen Single-Master)
- Bidirektionale Replikationsstrategien (entspricht im Allgemeinen Multi-Master)

In [SS05] sind neben Single-Master versus Multi-Master weitere Dimensionen genannt, nach denen asynchrone Replikationsstrategien klassifiziert werden können.

Unidirektionale Replikationsstrategien

Bei den unidirektionalen Replikationsstrategien treten keine Schreib-/Schreibkonflikte$_R$ auf, weil Schreibzugriffe ausschließlich über festgelegte Replikate erfolgen, wobei entweder alle Replikate auf einem Rechner, dem Master, liegen oder partitioniert auf mehreren Mastern, die dann eine disjunkte Menge von Replikaten kontrollieren [Len97]. Änderungen werden vom Master an die Slaves, d.h. an die anderen Replikate des entsprechenden logischen Objekts, propagiert. Bei Verzicht auf Sperren während der Propagierung kann es zu Schreib-/Lesekonflikten$_R$ kommen, wobei dieses Lesen veralteter Daten im Allgemeinen toleriert wird. Der Oberbegriff für diese Replikationsstrategien lautet **Master/Slave-Replikation**.

Die Master/Slave-Replikationsstrategien gewährleisten zumindest das Korrektheitskriterium *„Letztendliche Konsistenz“* (siehe Definition 19 auf Seite 34), weil nach Abschluss aller Aktualisierungen jedes Replikat eines jeden logischen Objekts wieder den gleichen Wert repräsentiert. Um detaillierte Aussagen zur Konsistenz, also zum „Alter“ der Replikate, vornehmen zu können, wurden Replikationsstrategien entwickelt, die speziellen abgeschwächten Konsistenzbegriffen (siehe Abschnitt 3.2.3) genügen. Bei der Replikationsstrategie **Snapshots** [AL80] werden so genannte Snapshot-Aktualisierungszeitpunkte (siehe Definition 23 auf Seite 37) festgelegt, zu denen die Slaves, also die Snapshots, spätestens aktualisiert werden. Falls die Aktualisierung fehl schlägt, dann wird der Snapshot ungültig gesetzt.

Bei der Replikationsstrategie **Quasi Copies** [ABGMA88] können so genannte Kohärenzbedingungen (siehe Definition 22 auf Seite 37) vereinbart werden, mit denen ein Abstand eines lokalen Replikats zum aktuellen Replikat in Form von Zeitverzug, Versionsabstand und/oder Wertedifferenz spezifiziert werden kann. Weil z.B. die Wertedifferenz nicht allein vom Slave geprüft werden kann, sendet der Master in festen Zeitabständen so genannte *„Alive-Nachrichten“*. Erhält ein Slave länger als die vorgegebene Zeit keine Alive-Nachricht, dann geht er von einem Rechnerausfall des Masters aus und setzt seine lokalen „Quasi-Kopien“ ungültig. Ähnliche Ansätze zur Abstandsspezifikation beinhalten die unidirektionalen Replikationsstrategien **Identity Connections** [WQ87, WQ90] und **Data Dependency Descriptors** [RSK91].

Bidirektionale Replikationsstrategien

Die bidirektionalen Replikationsstrategien erlauben nicht nur Änderungen an einem Master wie bei den unidirektionalen Replikationsstrategien, sondern an einer Gruppe von Rechnern mit Replikaten oder sogar an allen Rechnern. Im letzten Fall wird allgemein auch von **Peer-to-Peer-Replikation** [Len97] gesprochen, weil alle Replikate bzw. Rechner mit Replikat gleichberechtigt sind. Weil durch das „gleichzeitige" Schreiben unterschiedlicher Replikate des gleichen logischen Objekts bei Verzicht auf eine übergreifende Synchronisation Schreib-/Schreibkonflikte$_R$ auftreten, unterscheiden sich die bidirektionalen Replikationsstrategien in erster Linie in der Art der Konfliktbehandlung.

Bei der Replikationsstrategie **Independent Updates** [CHKS92, CHKS95] werden Versionsvektoren, so genannte *„Reception Vectors"*, an jedem Rechner für jedes Replikat geführt, um Inkonsistenzen zu erkennen. Zusätzlich wird an jedem Rechner ein Änderungsprotokoll, ein so genanntes *„History Log"*, gespeichert. Wenn über den Reception Vector ein Konflikt erkannt wird, dann werden die History Logs ausgetauscht, um den Konflikt zu behandeln. Ähnlich funktioniert die Replikationsstrategie **Timestamped Anti Entropy** (TSAE, Golding92). Bei TSAE werden Änderungen jedoch epidemisch verbreitet, d.h. ein Replikat tauscht sich periodisch mit anderen Replikaten aus, wobei die Auswahl zufällig sein kann. Damit kann wenig über die Konsistenz von Replikaten gesagt werden, sodass sich TSAE nur für bestimmte Anwendungen eignet, wie z.B. eine Literaturdatenbank, für die TSAE ursprünglich entwickelt wurde. Bei der **Lazy Replication** (auch **Gossip** genannt, [LLS90, LLSG92]) wurden Operationen eingeführt, mit denen verschiedene Konsistenzgrade spezifiziert werden können, um genauere Aussagen zumindest zu einem Teil der Replikate zu ermöglichen.

Bei der Replikationsstrategie **Data Patch** [GMAB+83] wird kein Änderungsprotokoll für die Konfliktbehandlung verwendet, sondern beim Datenbankentwurf werden Regeln spezifiziert, wie auf Schreib-/Schreibkonflikte$_R$ bei Netzpartitionierung reagiert wird. Dabei wird zwischen so genannten *„Tupel-Einfügeregeln"* und *„Tupel-Integrationsregeln"* unterschieden. Tupel-Einfügeregeln kommen dann zum Zug, wenn in einer Partition ein Tupel eingefügt wurde und in einer anderen nicht, wobei das Tupel dann verworfen oder in allen Partitionen beibehalten werden kann. Tupel-Integrationsregeln greifen, wenn in zwei Partitionen das gleiche Tupel geändert wurde. Hier kann die letzte Änderung oder ein priorisiertes Tupel überleben oder eine resultierende berechnet werden. In beiden Fällen kann der Konflikt auch durch ein Programm oder durch Benachrichtigung eines Administrators, der manuell eingreift, gelöst werden.

Speziell in der Anwendungsdomäne mobiler Systeme wurden viele Arbeiten zu bidirektionalen Replikationsstrategien veröffentlicht. Zwar ist die Problemstellung der Netzpartitionierung nichts Neues [BGM94], weil hier die Rechner mit Replikat aber häufig verbindungslos sind und somit eine Synchronisation nicht jederzeit möglich ist, werden spezielle Replikationsstrategien benötigt, wie z.B. **CODA** [Sat02], **Bayou** [TTP+95] oder **ROAM** [RRP99] mit seinen Vorgängern **Rumor** [GRG+98], **Ficus** [GHM+90] und **Locus** [WPE+83]. Gemeinsam haben diese Strategien, dass Zustände, wie „verbunden", „verbindungslos" oder „zusammenführend", unterschieden werden. Bei der Zusammenführung bzw. Reintegration erfolgt die Konfliktbehandlung z.B. über spezielle Anwendungsprogramme oder manuell über einen Administrator, der ggf. benachrichtigt wird. Bayou unterstützt zusätzlich als abgeschwächte Konsistenz die so genannten Session-Garantien (siehe Definition 21 auf Seite 36).

Auch im Bereich von Peer-to-Peer-Systemen [Sch01a, SW04a] (basierend auf einer Peer-to-Peer-Architektur, nicht zu verwechseln mit dem Begriff Peer-to-Peer Replikation, siehe oben) wurden Replikationsstrategien weiterentwickelt bzw. neu konzipiert. Neben der häufigen Verbindungslosigkeit kommt hier die Dynamik der beteiligten Rechner mit Replikat hinzu: Zu jeder Zeit können Teilnehmer aus dem Verbund ausscheiden oder hinzukommen. Ohne detailliert auf die Replikationsstrategien in dieser Domäne einzugehen, seien zumindest die folgenden Verfahren beispielhaft genannt: **Gossiping algorithm** [DHA03], **Deno** [CKBF03], **Decentralized Weighted Voting** [RL03] und **Hybrid Replication** [MNSO04].

3.4. Verwandte Arbeiten

Um einen optimalen Kompromiss zwischen den Zielkonflikten Verfügbarkeit, Performance und Konsistenz (siehe Abbildung 3.1 auf Seite 28) zu ermöglichen, werden geeignete Replikationsstrategien benötigt, die sich z.B. zur Laufzeit den Zustandsänderungen der zugrunde liegenden Systemlandschaft anpassen. Dies ist insbesondere bei großen Datenmengen (Stichwort: *„very large databases“*) der Fall. Eine redundante Verteilung der Daten kann den Durchsatz erhöhen, wofür aber häufig eine adaptive Replikationsstrategie benötigt wird, um den Anforderungen in komplexen Systemlandschaften gerecht zu werden. Ein weiterer Aspekt ist die Integration von Informationssystemen, speziell die Datenintegration. Insbesondere in großen Unternehmen werden aus verschiedenen Gründen im Allgemeinen mehrere heterogene Informationssysteme eingesetzt, die gemeinsame Daten verwalten. Um diese gemeinsamen Daten konsistent zu halten, müssen die Daten abgeglichen werden. Das theoretische Konzept des Datenabgleichs sind Replikationsstrategien.

Weil die in dieser Dissertation vorgestellte Replikationsstrategie RegRess (siehe Abschnitt 4) speziell für heterogene, autonome Informationssysteme entwickelt wurde, wird zunächst in Abschnitt 3.4.1 das Themengebiet Datenintegration im Hinblick auf Replikation betrachtet. Bei RegRess erfolgt die Koordination von Schreib- und Lesezugriffen regelbasiert, wodurch u.a. Zustandsänderungen der Systemlandschaft berücksichtigt werden können und somit eine Adaption zur Laufzeit ermöglicht wird. Daher zählen insbesondere die adaptiven Replikationsstrategien zu den verwandten Arbeiten. Eine Auswahl dieser adaptiven Replikationsstrategien wird in Abschnitt 3.4.2 vorgestellt.

3.4.1. Replikation und Datenintegration

Die Daten von großen Unternehmen sind häufig auf mehrere Informationssysteme verteilt, wobei oft jede Organisationseinheit eine eigene IT-Infrastruktur besitzt, sodass eine heterogene Landschaft mit oftmals autonomen Systemen entsteht [Has00]. Die Integration dieser Informationssysteme wird als *„Enterprise Application Integration“* (EAI, [CHK05]) bezeichnet. Gründe für eine Integration sind die Verbesserung der Kundenbeziehungen, verstärkte Unterstützung der Logistikkette, Rationalisierung interner Prozesse und/oder schnellere Inbetriebnahme neuer Anwendungen [RMB00]. Nach [Lin99, DLPR02] kann die Integration auf Datenebene, auf API-Ebene, auf Prozessebene und/oder auf Userebene erfolgen.

Hier soll die Integration auf Datenebene, auch Datenintegration genannt, detaillierter betrachtet werden, die laut [SS03] in redundanzfreie und redundante Lösungen unterschieden werden kann. Bei den redundanzfreien Lösungen gibt es nur Unikate (siehe Definition 12 auf Seite 32), d.h. die beteiligten Informationssysteme greifen auf eine zentralisierte Datenbank oder auf spezielle lokale Datenbanken zu, die Partitionen der Gesamtdaten speichern. Bei der redundanten Lösung gibt es mehrere Datenkopien, die entweder lokal zu jedem Informationssystem gespeichert sind oder von mehreren Informationssystemen jeweils gemeinsam genutzt werden. Wenn mehrere Datenkopien verwendet werden, dann müssen die Daten in bestimmten Intervallen abgeglichen werden. Wie bereits erwähnt, bilden Replikationsstrategien das theoretische Konzept für den Datenabgleich, d.h. Replikationsstrategien sorgen für die Propagierung von Änderungen an die Datenkopien, sprich den Replikaten.

In homogenen Systemlandschaften, insbesondere beim Einsatz gleicher Datenbankmanagementsystemen (DBMS, siehe z.B. [HS00, Dat03]), ist die Datenintegration eher unproblematisch. Im Allgemeinen bieten die DBMS-Hersteller in ihren Produkten schon Replikationsstrategien an (siehe z.B. [Len97]), sodass ein redundantes Speichern von Daten unter Berücksichtigung spezieller Konsistenzeigenschaften hier einfach zu realisieren ist. In heterogenen Systemlandschaften hingegen ist ein wichtiger Aspekt die Überwindung der Heterogenität. Neben der technischen Kopplung dieser Systeme müssen häufig unterschiedliche Datenmodelle bzw. unterschiedliche Schemata (Schemaintegration, siehe z.B. [Con02]) integriert werden. In dieser Dissertation soll

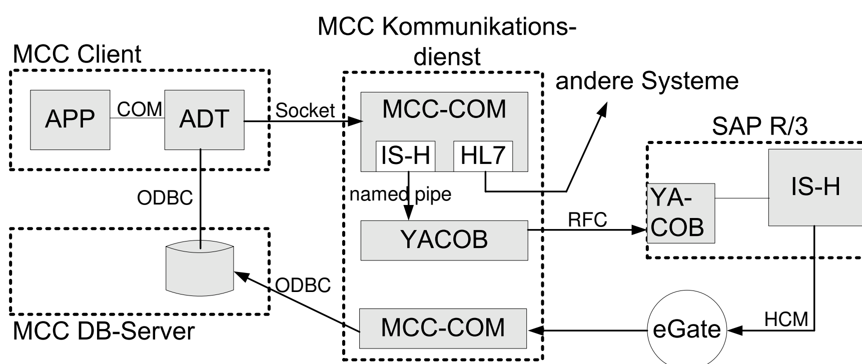

Abbildung 3.4.: Enge Kopplung klinischer Informationssysteme [NHW+02]

nicht vertieft auf Datenintegration eingegangen werden, sondern es werden kurz einige Arbeiten vorgestellt, die hinsichtlich der Replikation von Daten relevant sind.

Um heterogene Systeme miteinander interagieren zu lassen, werden häufig Kommunikationsserver eingesetzt (siehe Abschnitt 2.1.2). Hierbei handelt es sich um Middleware [BN96], die einen Nachrichtendienst realisiert und die zusätzliche Dienste wie Datentransformation bietet. Wenn ein logisches Objekt geändert wird, dann erfolgt die Änderung zunächst an einem Replikat. Das System (bzw. der Rechner), auf dem das Replikat gespeichert ist, propagiert die Änderung über den Kommunikationsserver an die anderen Systeme bzw. Rechner, die ebenfalls ein Replikat des logischen Objekts speichern. Weil die Nachrichten im Allgemeinen gepuffert werden, handelt es sich um eine asynchrone Kommunikation (siehe Abschnitt 2.1.2). Je nachdem, ob es sich um ein ausgezeichnetes Replikat handelt, dass geändert werden darf, oder ob jedes Replikat geändert werden kann, ist die zugrunde liegende Replikationsstrategie entweder die Master/Slave-Replikation oder die Peer-To-Peer-Replikation (siehe Abschnitt 3.3.2).

Ein weiterer Ansatz, Heterogenität zu überwinden, ist die Verwendung von Multidatenbanksystemen (MDBS, siehe z.B. [Rah94]), bei der lokale DBS eng oder lose gekoppelt sein können. Als Beispiel soll die Architektur von DIGAME dienen [dLPC04], bei der autonome und heterogene DBS lose gekoppelt sind, um Daten und Schemata zu replizieren. Dabei verteilt ein lokales DBS Änderungen an registrierte DBS, wobei die Master/Slave-Replikation umgesetzt wird.

Föderierte Datenbanksysteme (FDBS, siehe z.B. [Con97]), ein Spezialfall der MDBS, realisieren im Allgemeinen eine engere Kopplung. In [Has97] wird die Replikation von Patientendaten in einem klinischen Informationssystem vorgestellt. Hierbei wird standardmäßig die Master/Slave-Replikation realisiert, allerdings kann auch die Peer-To-Peer-Replikation verwendet werden. Auch in [Fer05] wird Replikation in einem FDBS betrachtet. Dabei wird auf eine zeitnahe Propagierung Wert gelegt, die dann zu *„quasi-konsistenten“* Zuständen führt.

Das Projekt EKKIS (Enge Kopplung klinischer Informationssysteme [NHW+02]) war die Motivation für diese Dissertation (siehe Abschnitt 1.1). Neben der losen Kopplung über eine asynchrone Kommunikation wurde eine enge Kopplung über eine synchrone Kommunikation beispielhaft für die klinischen Systeme MCC (http://www.meierhofer.de) und SAP IS-H (http://www.sap.com) realisiert. In Abbildung 3.4 ist die Kopplung der Systeme dargestellt, wobei der MCC Kommunikationsdienst die Schnittstelle des MCC-Systems bildet. Wenn ein MCC Client eine Datenänderung vornimmt, wird diese Änderung sowohl lokal an dem MCC DB-Server durchgeführt als auch über den MCC Kommunikationsdienst synchron an das SAP IS-H System geleitet. Eine Datenänderung im SAP IS-H System wird hingegen asynchron über den Kommunikationsserver „eGate“ (http://www.seebeyond.com) versendet. Durch die synchrone Kommunikation können synchrone Replikationsstrategien wie z.B. ROWA (siehe Abschnitt 3.3.1) realisiert werden, durch die asynchrone Kommunikation wird die Peer-To-Peer-Replikation umgesetzt.

Allgemein kann gesagt werden, dass unter dem Stichwort Datenintegration eher Arbeiten angesiedelt sind, die sich mit der Kopplung der Systeme zum Zweck des Datenaustausches beschäftigen. Der Fokus liegt auf der Überwindung von Heterogenität, wobei die Anbindung der beteiligten Systeme eine wichtige Rolle spielt, die auch in dieser Dissertation benötigt wird (siehe Abschnitt 7.1). Bei den zugrunde liegenden Replikationsstrategien wird im Allgemeinen auf traditionelle Replikationsstrategien zurückgegriffen, wobei auf Grund der asynchronen Kommunikation eher asynchrone Replikationsstrategien verwendet werden (siehe Abschnitt 3.3.2). Adaptive Replikationsstrategien werden in der Forschung eher unter dem Stichwort Replikation behandelt. Einige dieser Strategien werden im folgenden Abschnitt vorgestellt.

3.4.2. Adaptive Replikationsstrategien

Die statischen und dynamischen Replikationsstrategien, die in Abschnitt 3.3 vorgestellt wurden, zeichnen sich dadurch aus, dass die Koordination bei Schreib- und Lesezugriffen fest vorgegeben ist bzw. im dynamischen Fall „parametrisiert“ wird, z.B. durch Anpassung der Quorengröße. Die Abgrenzung zu den adaptiven Replikationsstrategien soll hier in der Form vorgenommen werden, dass bei der Adaption entweder mehrere der tradionellen Verfahren eingesetzt werden oder bei der Koordination derart flexibel auf spezielle Ereignisse reagiert wird, dass keine Zuordnung zu einer tradionellen Replikationsstrategie möglich ist. Die Einteilung ist sicherlich fließend und letztlich von geringer Bedeutung. Das Augenmerk liegt in diesem Abschnitt auf den Replikationsstrategien, die der in dieser Dissertation vorgestellten Replikationsstrategie RegRess nahe verwandt sind. Weil auch die adaptiven Replikationsstrategien entweder die 1-Kopien-Serialisierbarkeit erfüllen oder nicht, werden sie gemäß nachfolgender Einteilung vorgestellt:

- Adaptive synchrone Replikationsstrategien (vergleiche Definition 30 auf Seite 42)
- Adaptive asynchrone Replikationsstrategien (vergleiche Definition 31 auf Seite 42)

Wenn die Adaption durch Mischverfahren erfolgt, d.h. es wird z.B. zwischen traditionellen Replikationsstrategien bei Rechnerausfall gewechselt, und eine der möglichen Replikationsstrategien, zu der gewechselt werden kann, gewährleistet nicht die 1-Kopien-Serialisierbarkeit, dann wird das Mischverfahren zu den adaptiven asynchronen Replikationsstrategien gezählt.

Adaptive synchrone Replikationsstrategien

Bei den adaptiven synchronen Replikationsstrategien handelt es sich im Allgemeinen um so genannte *„Mischverfahren“*, d.h. um Replikationsstrategien, die tradionelle synchrone Replikationsstrategien kombinieren (siehe Abschnitt 3.3). Das Ziel dabei ist es, bei Rechnerausfall oder Netzpartitionierung die Verfügbarkeit von Zugriffen durch Wechsel zu einer anderen Replikationsstrategie zu erhöhen. Wenn die am Mischverfahren beteiligten Replikationsstrategien die 1-Kopien-Serialisierbarkeit gewährleisten, dann gewährleistet auch das Mischverfahren die 1-Kopien-Serialisierbarkeit, sofern zusätzlich beim Wechsel mögliche inkonsistente Zugriffe vermieden werden, z.B. kann von einem Votierungsverfahren nur dann zum ROWA gewechselt werden, wenn alle Änderungen an alle Replikate propagiert sind.

Das Mischverfahren **Missing Updates** (auch als **Missing Writes** bezeichnet, [ES83]) ist eine Kombination aus ROWA und einem Votierungsverfahren, dass im Wesentlichen dem Majority Consensus entspricht [DGMD85]. Im Normalmodus wird die ROWA-Strategie genutzt, sodass Lesezugriffe „billig“ sind, weil auf jedes Replikat zugegriffen werden kann. Im Fehlermodus, der dann aktiv ist, wenn eine Transaktion bei einem Schreibzugriff ein Replikat nicht erreichen konnte, wird zum Votierungsverfahren gewechselt. Im Fehlermodus sind Lesezugriffe „teuer“, weil nun ein Lesequorum erreicht werden muss. Außerdem ist die Verwaltung im Fehlermodus aufwändig. Alle Transaktionen, die nach der Transaktion ausgeführt werden, die den Fehlermodus ausgelöst hat, müssen im Fehlermodus ausgeführt werden und über die ausgelassenen Änderungen (*„missing updates“*) informiert werden. Angemerkt sei, dass eine Lesetransaktion

entscheiden kann, ob trotz ausgelassener Änderung ein Replikat gelesen wird, d.h. das Lesen veralteter Daten kann erlaubt werden.

Beim Mischverfahren **Virtual Partition** [ASC85, AT89] werden virtuelle Partitionen zur Abstraktion der realen Partitionen verwendet, um widersprüchliche reale Partitionierungen, die einzelne Rechner „sehen“ könnten, zu vermeiden. In einer virtuellen Partition können Zugriffe durchgeführt werden, wenn sie die Mehrheit der Rechner beinhaltet. Innerhalb einer virtuellen Partition konnte zunächst die ROWA-Strategie genutzt werden, später wurden auch Votierungsstrategien zugelassen. In einem nicht idealisierten Netzwerk ist das Lesen veralteter Daten möglich [BHG87].

Das **adaptive dynamic General Structured Voting** (adGSV, [ST06]) erlaubt gegenüber seinem Vorgänger dGSV (siehe Abschnitt 3.3) zur Laufzeit das Entfernen und Hinzunehmen von Rechnern mit Replikat. Beim adGSV können nicht nur Quorengrößen angepasst werden, sondern abhängig vom Replikationsgrad (siehe Definition 6 auf Seite 30) kann zu unterschiedlichen Votierungsstrategien gewechselt werden. Dazu wird in einem Register festgehalten, bei welchem Replikationsgrad n welche Votierungsstrategie verwendet werden soll, z.B. bei $n = 1, .., 8$ ein Weighted Voting mit an n angepassten Quorengrößen, bei $n = 9$ das Grid Protocol und bei $n = 10, 11, ...$ das Majority Consensus. Die Einträge in dem Register können durch so genannte *„Strukturgeneratoren“* erzeugt werden.

Adaptive asynchrone Replikationsstrategien

Die adaptiven asynchronen Replikationsstrategien, die nachfolgend vorgestellt werden, sind der in dieser Dissertation entwickelten Replikationsstrategie RegRess am nächsten verwandt, weil auch bei RegRess eine Adaption zur Laufzeit vorgenommen wird. Das Ziel der adaptiven asynchronen Replikationsstrategien ist dabei ebenfalls die Erhöhung der Verfügbarkeit und der Performance. Im Gegensatz zu den adaptiven synchronen Replikationsstrategien wird jedoch die 1-Kopien-Serialisierbarkeit nicht gewährleistet, insbesondere werden Schreib-/Lesekonflikte$_R$ im Allgemeinen toleriert.

Die Replikationsstrategie **Application-oriented SPEcification of Consistency Terms** (ASPECT, [Len96, Len97]), die auf dem **Distributed Data Management System** (DDMS, [JRW90]) aufbaut, hat eine Analogie zum Primary Copy Verfahren (siehe Abschnitt 3.3.1), wobei eine Aktualisierung der so genannten *„Sekundärkopien“* durch Prädikate gesteuert wird. Bei ASPECT wird aber nicht eine feste Primärkopie verwendet, sondern eine *„virtuelle Primärkopie“* (Virtual Primary Copy), die auch aus mehreren Replikaten bestehen kann. Diese Menge von Replikaten wird als *„Konsistenzinsel“* bezeichnet, weil alle Replikate, die zur Konsistenzinsel gehören, konsistente Zugriffe erlauben. Dazu ist es nicht nötig, dass alle Replikate der Konsistenzinsel stets den aktuellen Wert repräsentieren, sondern dass diese Replikate gemäß einer synchronen Replikationsstrategie koordiniert werden.

Für die *„Sekundärkopien“*, also die Replikate, die nicht der Konsistenzinsel angehören, können Prädikate für Konsistenzgarantien spezifiziert werden, die sowohl eine zeitliche Dimension als auch räumliche Dimension aufweisen. Die zeitliche Dimension entspricht den Aktualisierungszeitpunkten von Snapshots (siehe Definition 23 auf Seite 37) und die räumliche Dimension den Kohärenzbedingungen der Quasi-Copies (siehe Definition 22 auf Seite 37). Damit eine Sekundärkopie asynchron aktualisiert werden kann, muss sie eine Referenz zu mindestens einem Replikat der Konsistenzinsel besitzen. Zusätzlich kann die Konsistenzinsel sich dynamisch ändern, d.h. Replikate treten der Konsistenzinsel bei oder verlassen die Konsistenzinsel, wobei mindestens ein Replikat der Konsistenzinsel angehören muss. Um eine synchrone Koordination der Konsistenzinsel zu gewährleisten, müssen sich die Replikate der Konsistenzinsel „kennen“, d.h. es müssen auch hierfür Referenzen verwaltet werden.

Das **Freshness Aware Scheduling** (FAS, [RBSS02]) ist an Primary Copy angelehnt, wobei zur Koordination der Zugriffe eine Middleware (siehe Abschnitt 2.1.2) verwendet wird. Ein Schreibzugriff wird zunächst an dem so genannten *„OLTP Node“* durchgeführt und dann asynchron an so genannte *„OLAP Nodes“* propagiert. Für die Replikate wird ein *„Frischeindex“*

definiert (siehe Anmerkungen zu Datenfrische nach Definition 24 auf Seite 38), mittels dem das Alter der Replikate auf den OLAP Nodes bestimmt wird. Der OLTP Node hat die Frische 1,0. Bei Lesezugriffen wird dann ein gewünschter Wert für den Frischeindex angegeben und die Middleware wählt einen geeigneten OLAP Node für den Zugriff aus, ggf. muss vorher ein so genanntes *„Freshness-Update“* durchgeführt werden.

Bei der **Configured Replication** [LH00], einem Vorschlag für Replikation in mobilen Anwendungen, wird einerseits die Datenreduktion wegen der ressourcenschwachen Systeme betrachtet und andererseits eine Kombination von synchronen und asynchronen Replikationsstrategien empfohlen, wobei zusätzlich semantisches Wissen wie z.B. Standort des Nutzers ausgenutzt wird. Je nach Anwendung soll ein optimales Mischverfahren hinsichtlich der Zielkonflikte Konsistenz, Performance und Verfügbarkeit (siehe Abbildung 3.1 auf Seite 28) ausgewählt werden.

Auch die **Flexible Replication Architecture for a Consistency Spectrum** (Fracs, [ZZ03]) implementiert einen Algorithmus, um den Zielkonflikt möglichst gut zu lösen. Der zugrunde liegende Algorithmus, der im FRACS-Prototyp implementiert wurde, ist dezentral. Weiterhin wird ein so genanntes *„update window“* für jedes Replikat definiert, mit dem der Umfang der Inkonsistenz bestimmt wird, wobei als Beispiel in [ZZ03] die Anzahl fehlender Updates verwendet wird. Änderungen an einem lokalen Replikat werden so lange gepuffert, bis die Fenstergrenze erreicht ist. Dann werden keine weiteren Änderungen des Replikats angenommen, bis die gepufferten Änderungen propagiert sind. Zusätzlich wird jedem Replikat ein *„Gewicht“* zugeordnet, dass z.B. die Änderungsrate wiedergibt und an einem Replikat gemessen wird. Darüber werden Formeln für die Kosten und die Verfügbarkeit hergeleitet.

Die „jüngeren“ Arbeiten im Forschungsgebiet Datenreplikation konzentrieren sich nach wie vor häufig auf eine Adaption hinsichtlich der Zielkonflikte Konsistenz, Performance und Verfügbarkeit, wobei oft sehr große Systeme als Ziel der Arbeiten dienen. Bei **Tunable Availability and Consistency Trade-Offs** (TACT, [YV02]) wird ein dreidimensionaler Vektor für verschiedene Abstandsmaße verwendet, um Inkonsistenzen zu begrenzen. **Refresco** [PGV04, PG06] nutzt unterschiedliche Aktualisierungsstrategien, um insbesondere die Verfügbarkeit von Lesezugriffen zu erhöhen. Beim **Availability/Consistency Balancing Replication Model** (ACBRM, [OFG07]) werden Netzpartitionierungen behandelt, wobei in einer Partition die 1-Kopien-Serialisierbarkeit gewährleistet wird, aber mögliche Schreib-/Schreibkonflikte$_R$ bei der Zusammenführung der Partitionen behandelt werden müssen. Mittels ACBRM kann z.B. das **Adaptive Voting Protocol** [OFGG06] implementiert werden. Auf sehr große, skalierbare Systeme im Umfeld von Webdiensten (*„Web Services“*) zielen die Replikationsstrategien **Google's File System** (GFS, [GGL03]), **Chain Replication** [vRS04] und **Niobe** [MTJ$^+$08], bei denen neben der Erhöhung der Verfügbarkeit auch die Lokalisation der Replikate eine Rolle spielt.

Zu Beginn dieser Dissertation wurde der adaptive Replikationsmanager **ARM** [NHHT03] entwickelt und prototypisch implementiert (siehe Abschnitt 6.1 bzw. Abbildung 6.1 auf Seite 154). Dabei wurde eine regelbasierte Replikationsstrategie umgesetzt, die als erste Version der in Kapitel 4 vorgestellten Replikationsstrategie RegRess aufgefasst werden kann. Bei jedem Schreibzugriff auf ein logisches Objekt werden die zugehörigen Replikate in synchron und asynchron zu aktualisierende Replikate partitioniert. Die Aufteilung kann durch so genannte Replikationsregeln konfiguriert werden, die zur Laufzeit ausgewertet werden. Weil protokollierte Kennzahlen berücksichtigt werden, ist somit eine Adaption an den aktuellen Systemstatus möglich. Bei den Regeln können sowohl funktionale Anforderungen, z.B. zeitlicher Rückstand, als auch technische Anforderungen, z.B. Rechnerausfall bzw. Rechnerperformance, berücksichtigt werden.

Der regelbasierte Ansatz ist zwar aufwändig, weil bei jedem Zugriff eine Regelauswertung durchgeführt werden muss, bietet aber gegenüber den oben genannten Replikationsstrategien eine höhere Flexibilität. Es können grundsätzlich sämtliche in Abschnitt 3.2.3 genannten abgeschwächten Konsistenzbegriffe realisiert werden, wenn die entsprechenden Kennzahlen zur Verfügung stehen. Weiterhin ist auch eine Reaktion auf technische Veränderungen wie reduzierte Performance eines Rechners möglich, die z.B. bei der Replikationsstrategie ASPECT unberück-

sichtigt blieb. Auch bei Lesezugriffen können mittels Regeln andere Kriterien herangezogen werden als die Datenfrische, wie z.B. bei FAS (Anmerkung: Beim ARM wurden Lesezugriffe ausschließlich lokal durchgeführt, d.h. Lesezugriffe wurden nicht regelbasiert koordiniert).

Eine regelbasierte Replikationsstrategie bietet weiterhin die Möglichkeit, unterschiedlichste Konsistenzanforderungen für unterschiedliche logische Objekte oder sogar Replikate zu spezifizieren. Zusätzlich können auf jeder Ebene wie z.B. für ein einzelnes Replikat auch zeitlich unterschiedliche Regeln festgelegt werden, d.h. ein Replikat könnte tagsüber anderen Konsistenzanforderungen genügen wie nachts. Die hohe Flexibilität wird, wie bereits erwähnt, durch einen erhöhten Rechenaufwand und Konfigurationsaufwand erkauft. Da aber insbesondere komplexe, große Systeme häufig eine hohe Flexibilität benötigen, ist die Entwicklung der in dieser Dissertation vorgestellten regelbasierten Replikationsstrategie RegRess (siehe Kapitel 4) gerechtfertigt. Ein Vergleich von adaptiven Replikatiosstrategien zur Replikationsstrategie RegRess befindet sich in Abschnitt 7.3.4.

Teil II.

Regelbasierte Replikationsstrategie

4. Regelbasierte Replikationsstrategie RegRess

Die Replikationsstrategie RegRess (**Reg**elbasierte **Re**plikation**ss**trategie), die in diesem Kapitel vorgestellt wird, ist eine Replikationsstrategie, bei der die Zugriffe auf die Replikate durch Regeln koordiniert werden. In der Definition 10 auf Seite 31 wird eine Replikationsstrategie dahingehend erklärt, dass ein Schreib- bzw. Lesezugriff auf ein logisches Datenobjekt in eine Menge von Schreib- bzw. Lesezugriffen auf physische Datenobjekte, den Replikaten, übersetzt wird. Bei RegRess werden diese Mengen für jeden Zugriff durch Auswertung und Schlußfolgerung von Regeln bestimmt, um ein Optimum hinsichtlich der Zielkonflikte Konsistenz, Verfügbarkeit und Performance zu erreichen (siehe Abschnitt 3.1). Ein geeigneter Kompromiss zwischen diesen konträren Zielen wird durch die Spezifikation anwendungsbezogener Regeln möglich.

In Abschnitt 3.3.1 wurden synchrone Replikationsstrategien vorgestellt, die die betroffenen Replikate für alle Schreibzugriffe bzw. für alle Lesezugriffe starr festlegen. Selbst die Votierungsstrategien, bei denen die zu den Schreib- bzw. Lesequoren gehörenden Replikate variieren können, legen zumindest die Quorengröße fest. Durch das starre Prinzip können die synchronen Replikationsstrategien hohen Konsistenzanforderungen genügen, d.h. sie gewährleisten die 1-Kopien-Serialisierbarkeit. Dieser Vorteil geht aber zu Lasten der Verfügbarkeit und der Performance. Demgegenüber bieten die asynchronen Replikationsstrategien (siehe Abschnitt 3.3.2) im Allgemeinen eine höhere Verfügbarkeit bzw. Performance, was allerdings auf Kosten der Konsistenz geschieht. Dabei werden entweder nicht so hohe Ansprüche an die Synchronisation nebenläufiger Prozesse gelegt (siehe Abschnitt 2.2.3) oder die starre Festlegung der betroffenen Replikate bei Schreib- und Lesezugriffen wird zumindest aufgeweicht.

Durch eine Kombination von synchroner und asynchroner Replikation können situationsabhängig die Vorteile beider Varianten genutzt werden. Mit situationsabhängig ist gemeint, dass eine Veränderung von fachlichen Anforderungen oder eine Modifikation der technischen Eigenschaften des verteilten Systems berücksichtigt werden kann. Um diesen Ansprüchen zu genügen, erfolgt bei RegRess zunächst eine Konfiguration durch Bestimmung geeigneter Regeln. Zur Laufzeit werden dann diese Regeln und ggf. protokollierte Systemkennzahlen herangezogen, um die Menge der betroffenen Replikate bei einem Zugriff zu ermitteln. RegRess ist somit eine konfigurierbare, adaptive Replikationsstrategie.

Das wichtigste Kriterium einer Replikationsstrategie ist die Korrektheit bzw., in dieser Arbeit synonym, die Konsistenz (siehe Abschnitt 3.2.2). Da grundsätzlich auch eine asynchrone Aktualisierung in RegRess realisierbar ist und Lesezugriffe auf veraltete Daten toleriert werden, wird die 1-Kopien-Serialisierbarkeit nicht garantiert. Um aber Aussagen hinsichtlich der abgeschwächten Konsistenz zu ermöglichen, wird ein so genannter *„Konsistenzgrad“* definiert. Der Konsistenzgrad ist abhängig von den Regeln, die für eine spezielle Anwendung formuliert werden. Hierbei handelt es sich um einen probabilistischen Wert, d.h. ein Wert, der die Wahrscheinlichkeit angibt, mit dem die Zugriffe auf konsistente Daten erfolgen.

Im Folgenden wird das Konzept von RegRess vorgestellt. Dazu werden zunächst in Abschnitt 4.1 Annahmen zur Abstraktion über das zugrunde liegende System vorgenommen sowie die in dieser Arbeit verwendete Terminologie erläutert. In Abschnitt 4.2 wird das Protokoll von RegRess spezifiziert, d.h. die Koordination der Schreib- und Lesezugriffe wird festgelegt. Anschließend wird in Abschnitt 4.3 die von RegRess gewährleistete Konsistenz und Korrektheit diskutiert, wobei insbesondere die möglichen Regeln definiert werden. Inwieweit die Regeln Einfluss auf einen optimalen Kompromiss hinsichtlich der Replikationsziele haben, wird abschließend in Abschnitt 4.4 dargestellt.

4.1. Annahmen und verwendete Notationen

Im Titel dieser Arbeit wird explizit ausgedrückt, dass Replikationsstrategien für heterogene, autonome Informationssysteme untersucht werden. *„Autonomie“* bekundet dabei, dass die beteiligten Informationssysteme durch die Replikationsstrategie nicht bzw. nicht zu stark eingeschränkt werden dürfen. Da Replikationstransparenz und Autonomie gegenläufige Ziele sind (siehe Abschnitt 3.1), wird hier dementsprechend auf vollständige Transparenz bei der Replikation verzichtet, d.h. letztendlich, dass eine abgeschwächte Konsistenz der Replikate toleriert wird (siehe Abschnitt 3.2.3), insbesondere dann, wenn die Autonomie einzelner Informationssysteme gefährdet ist.

„Heterogenität“ bedeutet im Allgemeinen, dass höhere Ansprüche an eine Implementierung einer Replikationsstrategie gestellt werden als in einer homogenen Systemlandschaft. Wenn z.B. einheitliche Datenbankmanagementsysteme (DBMS) mit identischen Schemata auf den lokalen Rechnern eingesetzt werden, kann eine Replikation häufig vom DBMS selbst übernommen werden, weil die meisten Hersteller von Datenbanksystemen unterschiedliche Replikationsstrategien innerhalb ihrer Systeme implementiert haben [Bur97, Len97]. In heterogenen Systemlandschaften müssen zusätzlich die Heterogenitäten überwunden werden, z.B. ist im Allgemeinen eine Schematransformation notwendig. Häufig wird eine geeignete Infrastruktur benötigt, um eine Interaktion der beteiligten Systeme zu erlauben. Diese höheren Anforderungen und Lösungsvorschläge werden z.B. in [NHW+02] diskutiert, wo Kopplungsstrategien für heterogene Anwendungssysteme im Krankenhaus präsentiert werden.

Da die hier vorgestellte Replikationsstrategie RegRess speziell für heterogene autonome Informationssysteme entwickelt wurde, wird bei der Evaluation nochmals auf die Überwindung von Heterogenität eingegangen (siehe Kapitel 6). Bei der Konzeption von RegRess wird jedoch von einem idealisierten System bzgl. der Datenstrukturen ausgegangen, um sich auf das Wesentliche einer Replikationsstrategie zu konzentrieren, nämlich der Koordination der Zugriffe. Daher werden in Abschnitt 4.1.1 zunächst Annahmen zur Abstraktion über das zugrunde liegende System getroffen. Abschließend werden in Abschnitt 4.1.2 die in dieser Arbeit verwendeten Abkürzungen und Notationen festgelegt.

4.1.1. Annahmen

In diesem Abschnitt werden ohne Beschränkung der Allgemeinheit einige Annahmen getroffen, die die Konzeption der Replikationsstrategie RegRess und die Entwicklung der Regelsprache RRML (siehe Kapitel 5) erleichtert. Hierdurch wird auch erreicht, dass die prototypische Implementierung (siehe Kapitel 7) sich auf die wesentlichen Funktionen der Datenreplikation beschränkt. Das zugrunde liegende Konzept von RegRess bleibt durch die getroffenen Annahmen unangetastet, befreit jedoch von unwesentlichen Funktionen wie beispielsweise Schematransformationen. Anschließend an die folgende Aufzählung der Annahmen erfolgt neben einer Erläuterung eine Beschreibung der Aktivitäten, die bei Verzicht auf die Vereinfachungen notwendig würden:

- Voll-replizierte Datenbanken
- Identische Schemata der lokalen Datenbanken
- Identische Konsistenzbedingungen der lokalen Datenbanken
- Einfache Datenobjekte
- Lese- und Schreiboperationen auf logische bzw. physische Datenobjekte
- Keine Seiteneffekte bei Schreiboperationen
- Kommunikationsfehler: Rechnerausfall und Netzpartitionierung

In [EN02] wird zwischen voller und partieller Replikation unterschieden. Bei voll-replizierten Datenbanken ist jedes logische Datenobjekt auf jedem beteiligten Rechner physikalisch gespeichert, d.h. der Replikationsgrad (siehe Definition 6 auf Seite 30) aller logischen Datenobjekte ist

gleich n, wobei n die Anzahl der Rechner mit Datenbanken ist. Bei partiell-replizierten Datenbanken liegt der Replikationsgrad eines jeden logischen Datenobjekts zwischen 1 und n, wobei mindestens bei einem logischen Datenobjekt der Replikationsgrad kleiner als n ist. In dieser Arbeit wird von voll-replizierten Datenbanken ausgegangen, um nach [Rah94] Fragmentierungs- und Allokationsprobleme zu umgehen. Bei partiell-replizierten Datenbanken muss zusätzlich für jedes logische Datenobjekt mitgeführt werden, wo die jeweiligen physikalischen Datenobjekte gespeichert sind, um bei Zugriffen die „richtigen" Rechner zu kontaktieren. Diese Informationen könnten z.B. in den Regeln abgelegt werden.

In dieser Arbeit wird bei der Konzeption der Replikationsstrategie RegRess davon ausgegangen, dass die lokalen Datenbanken jeweils über das gleiche Schema verfügen, d.h. es wird keine Schemaintegration benötigt. Zwar zeichnen sich heterogene Systeme im Allgemeinen dadurch aus, dass unterschiedliche Schemata verwendet werden, trotzdem soll dieser Aspekt nicht bei der Replikationsstrategie betrachtet werden, sondern bei der Software-Architektur (siehe Abschnitt 6.1). Die Aufgabe der Schematransformation (siehe z.B. [Con02]) wird als zur Replikation getrennte Funktionalität betrachtet, obwohl beispielsweise Lesezugriffe auf Replikate performanter sind, wenn keine Datentransformationen erforderlich sind. Ggf. könnten derartige Aspekte in den Regeln berücksichtigt werden.

Auf den lokalen Datenbanken sollen identische Konsistenzbedingungen (synonym: Integritätsbedingungen) definiert sein (siehe Abschnitt 2.3.2), d.h. eine Datenmanipulation hält entweder auf jeder lokalen Datenbank die Konsistenzbedingungen ein und kann durchgeführt werden oder sie hält auf keiner lokalen Datenbank die Konsistenzbedingungen ein und wird abgebrochen. Durch diese Annahme wird gesichert, dass eine Änderung, die bereits an einem Replikat endgültig durchgeführt wurde, bei einem anderen Replikat desselben logischen Datenobjekts, das zeitversetzt aktualisiert werden soll, ebenfalls durchgeführt werden kann. Insbesondere bei asynchronen Replikationsstrategien könnten unterschiedliche Konsistenzbedingungen der lokalen Datenbanken zu Problemen führen, d.h. ein nachträgliches Rücksetzen oder Kompensieren von Transaktionen erforderlich machen. Vorbeugend könnten nur solche Änderungen zugelassen werden, die auf allen lokalen Datenbanken die jeweiligen Konsistenzbedingungen erfüllen.

Mit „einfachen" Datenobjekten ist gemeint, dass die logischen und somit die physikalischen Datenobjekte bzw. deren Elemente aus einfachen Datentypen bestehen. Insbesondere Methoden der Datenobjekte bzw. Zugriffsoperationen auf die Datenobjekte lesen oder schreiben die Objekte als eine Einheit. In dieser Arbeit werden beispielsweise keine XML-Dokumente betrachtet, die „gleichzeitig" eine Änderung an verschiedenen Stellen innerhalb eines Dokuments erlauben und geeignet zusammengeführt werden müssen. Auf derartige, so genannte *„Merge"*-Funktionalität wird hier verzichtet.

Es werden ausschließlich Lese- und Schreiboperationen auf logische bzw. physische Datenobjekte betrachtet, die zum Lesen oder Manipulieren der Objekte in Frage kommen. Bei einer Leseoperation wird das Datenobjekt komplett gelesen, ggf. werden alle Attribute des Objekts gelesen. Bei einer Schreiboperation wird das Datenobjekt komplett geschrieben, ggf. werden einzelne Attribute mit dem gleichen Wert geschrieben. Wenn andere Operationen untersucht werden müssen, können diese durch Lese- bzw. Schreiboperationen abgebildet werden (siehe Abschnitt 2.2.1).

Bei Schreiboperationen sollen keine so genannten Seiteneffekte auftreten, d.h. beim Schreiben eines logischen Datenobjekts werden keine anderen Datenobjekte z.B. durch *„Trigger"* (siehe [HS00]) manipuliert. Hiermit wird sichergestellt, dass bei Änderung eines logischen Datenobjekts ausschließlich dessen Replikate betrachtet werden müssen. Wenn solche Seiteneffekte nicht ausgeschlossen werden und diese möglicherweise bei den lokalen Datenbanken des verteilten Systems unterschiedlich aussehen, dann müssen diese Seiteneffekte in allen lokalen Datenbanken gleichermaßen durchgeführt werden, um die Replikate konsistent zu halten.

Eine wichtige Eigenschaft von Replikationsstrategien ist die Reaktion auf Kommunikationsfehler (siehe Abschnitt 2.1.2), wobei in dieser Arbeit ausschließlich Rechnerausfälle und Netz-

partitionierungen betrachtet werden. Ein Rechnerausfall bzw. ein Dienstausfall wird auch als Fail-Stop [SS83b] bezeichnet: Ein Rechner oder auch ein Dienst ist nicht mehr verfügbar, d.h. es erfolgt keine Anfragebearbeitung bzw. keine Antwort. Fehlerhafte Antworten z.B. durch Störungen, d.h. so genannte *„Byzantinische Fehler“* (siehe z.B. [MR98, CL99]), werden nicht betrachtet. Derartige Probleme werden z.B. bei fehlertoleranten Systemen gelöst [Ech90, LA90]. Bei Fehlern auf Kommunikationskanälen wird davon ausgegangen, dass ein oder mehrere Rechner nicht mehr erreicht werden und dass es zu Netzpartitionierungen kommt. Es wird nicht betrachtet, ob es zu Verfälschungen auf dem Kommunikationskanal kommt.

4.1.2. Begriffe und Notationen

Bevor die in dieser Arbeit verwendeten Abkürzungen und Notationen festgelegt werden, erfolgt zunächst die Definition einiger Begriffe. In der Literatur wird speziell der Begriff *„Komponente“* unterschiedlich definiert (siehe z.B. [Gri98, SGM02]) und kontrovers diskutiert. In dieser Arbeit erfolgt die Modellierung und Dokumentation der entwickelten Konzepte und Architekturen in der Unified Modeling Language, Version 2 (UML 2, siehe z.B. [RJB04, RHQ+05]). Daher soll hier auch der Komponentenbegriff aus der so genannten UML 2 Superstructure Spezifikation [OMG06] verwendet werden:

Definition 32 (nach [OMG06]) ***Komponente**: Eine Komponente repräsentiert einen modularen Teil eines Systems, der seinen Inhalt kapselt und dessen Erscheinungsform innerhalb seiner Umgebung austauschbar ist. Eine Komponente definiert ihr Verhalten in Form von angebotenen und benötigten Schnittstellen.*

Auf eine detaillierte Diskussion der unterschiedlichen Definitionen des Begriffs *„Komponente“* wird in dieser Arbeit verzichtet, da eine grobe Vorstellung dessen, was eine Komponente ist, an dieser Stelle ausreicht. Vereinfacht gesagt, ist hier mit einer Komponente eine ausführbare Software-Komponente gemeint, d.h. ein Artefakt, das im Softwareentwicklungsprozess entsteht. Auch die Granularität von Komponenten soll nicht weiter beleuchtet werden. So kann es „kleine“ Komponenten oder „große“ Komponenten geben. In diesem Sinne sollen hier z.B. Datenbankmanagementsysteme (DBMS), Applikationsserver oder Enterprise Resource Planing Systeme (ERP-Systeme) als Komponente bezeichnet werden.

Um das in diesem Abschnitt angeführte Beispiel einer Systemumgebung mit replizierten Datenbanken besser zu veranschaulichen, soll der Begriff *„Replikationsmanager“* definiert werden. Da ein Replikationsmanager eine Implementierung darstellt, also Software ist, und auch die weiteren Eigenschaften der Definition 32 beinhalten soll, fließt der Begriff Komponente in die Definition ein:

Definition 33 ***Replikationsmanager**: Ein Replikationsmanager ist eine Komponente, die eine Replikationsstrategie implementiert. Ein Replikationsmanager kann eine Komponente innerhalb der lokalen Datenbankmanagementsysteme oder eine eigenständige Anwendung sein. Weiterhin ist eine zentrale oder dezentrale Implementierung des Replikationsmanagers möglich.*

Wesentlich ist also, dass ein Replikationsmanager die Realisierung einer Replikationsstrategie ist. Somit übernimmt ein Replikationsmanager die Koordination der Schreib- und Lesezugriffe auf die Replikate. Zusätzlich sind in Definition 33 Implementierungsaspekte aufgenommen, die für die Koordination eher irrelevant sind. Ob ein Replikationsmanager innerhalb eines DBMS implementiert oder eine eigenständige Anwendung ist, hängt im Allgemeinen davon ab, ob eine homogene oder heterogene Systemlandschaft vorliegt (siehe Abschnitt 2.1.3). Die Vor- und Nachteile einer zentralen bzw. dezentralen Realisierung werden in Kapitel 6 diskutiert.

Da die Replikate in einem verteilten System an unterschiedlichen Stellen lokalisiert sind, kann der Replikationsmanager die Schreib- oder Lesezugriffe nicht direkt durchführen, sondern es werden geeignete Schnittstellen benötigt, die z.B. von dem entsprechenden DBMS oder einer

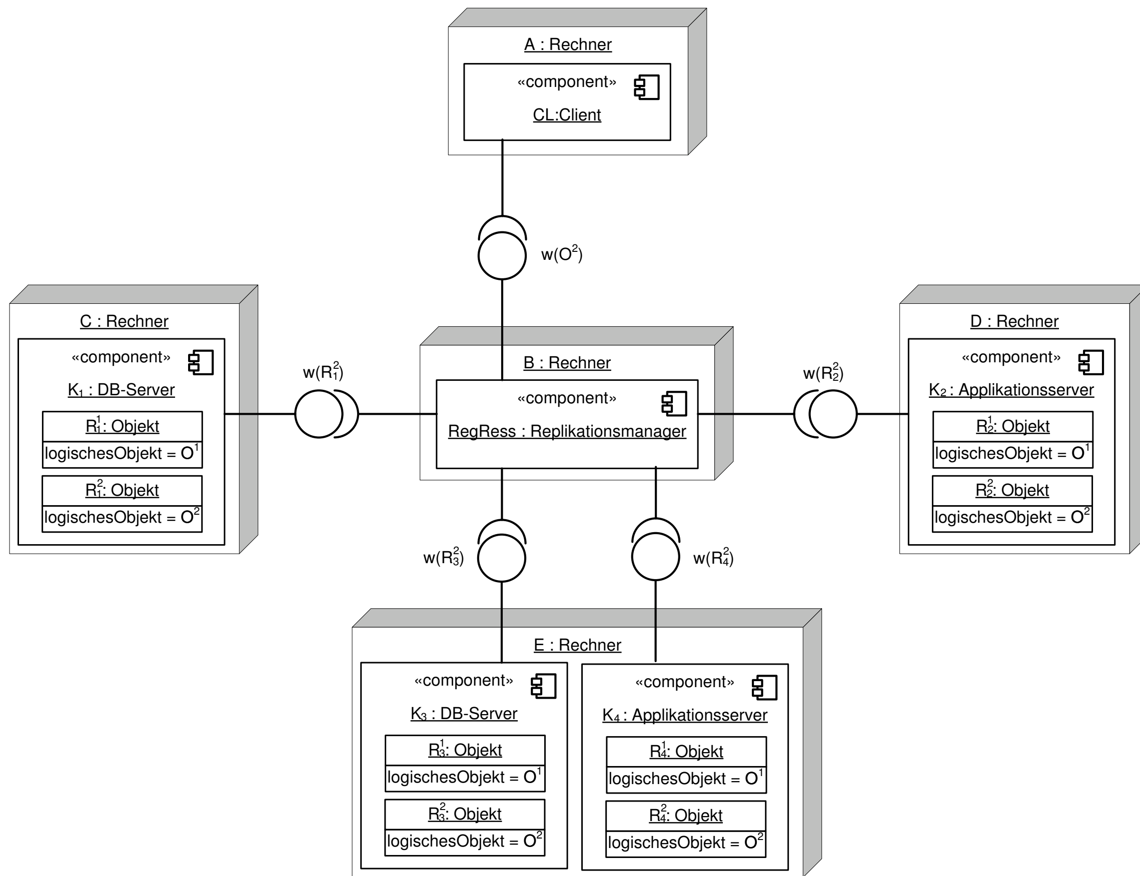

Abbildung 4.1.: Beispiel einer Systemumgebung als UML-Verteilungsdiagramm

Applikation bereitgestellt werden. In der Terminologie der UML 2 werden also Schnittstellen entsprechender Komponenten benötigt, oder anders gesagt, Komponenten speichern Replikate und kontrollieren die Zugriffe auf die Replikate. Hierdurch ist folgende Definition motiviert, in der anstatt *„Rechner mit Replikat“* (siehe Definition 8 auf Seite 30) bzw. *„System mit Replikat“* (siehe Definition 9 auf Seite 31) nun der Begriff Komponente gemäß der UML 2 verwendet wird:

Definition 34 ***Komponente mit Replikat***: *Eine Komponente mit Replikat ist eine Komponente, die jeweils genau ein Replikat eines logischen Datenobjekts speichert, d.h. Zugriffe auf dieses Replikat erfolgen über Schnittstellen dieser Komponente. In einer voll-replizierten Datenbank mit Replikationsgrad n speichert eine Komponente mit Replikat genau n Replikate.*

Anhand der Abbildung 4.1 werden nun die in dieser Arbeit verwendeten formalen Bezeichnungen erläutert. Im Beispiel ist eine einfache Systemumgebung mit replizierten Daten als UML-Verteilungsdiagramm dargestellt, wobei fünf Knoten A bis E, die vom Typ „Rechner“ sind, existieren. Gemäß UML 2 sind auf den Knoten Artefakte verteilt, d.h. im Beispiel die Komponenten Client, Replikationsmanager und vier Komponenten mit Replikaten. Die Replikate sind als Artefakte innerhalb der Komponenten in Form von Objekten illustriert. Grundsätzlich können mehrere Komponenten auf einem Knoten lokalisiert sein, z.B. sind zwei Komponenten mit Replikaten auf dem Rechner E eingesetzt. Ein Replikat könnte auch auf einem anderen Knoten als seine zugehörige Komponente gespeichert sein. Dieser Fall wird ohne Beschränkung der Allgemeinheit in dieser Arbeit nicht betrachtet.

Im Beispiel wird angenommen, dass zwei logische Datenobjekte O^1 und O^2 existieren. Logische Datenobjekte (oder einfach Objekte) werden mit dem Buchstaben „O“ gekennzeichnet, wobei ggf. eine laufende Nummerierung hochgestellt wird (Eselsbrücke: **o**ben beginnt wie **O**bjekt mit

einem „o“). Die Objekte O^o, $o = 1, 2$ können potenziell von vier Komponenten K_1, K_2, K_3 bzw. K_4 physikalisch gespeichert sein, wobei hier eine Indizierung tief gestellt wird.

Da laut Annahme (siehe Abschnitt 4.1.1) von einer voll-replizierten Datenbank ausgegangen wird, sind die logischen Datenobjekte in jeder Komponente lokalisiert, d.h. zu jedem Datenobjekt existieren vier Replikate und jede Komponente mit Replikat beherbergt zwei Replikate. Ein Replikat wird mit „R“ abgekürzt, wobei ein hochgestellter Index die laufende Nummer des Objekts und ein tiefgestellter Index die laufende Nummer der Komponente angibt, z.B. bezeichnet R_3^2 das Replikat des logischen Datenobjekts O^2, das auf der Komponente K_3 lokalisiert ist. In der Abbildung 4.1 ist die Zugehörigkeit der acht Replikate R_k^o, $o = 1, 2$; $k = 1, 2, 3, 4$ zu ihrem logischen Datenobjekt O^o dadurch dargestellt, dass ein Attribut mit dem entsprechenden Wert belegt ist.

Der Replikationsmanager ist in diesem Beispiel als externe, zentrale Komponente verwirklicht. Ein Client, der ggf. auch Teilkomponente einer Komponente mit Replikat sein kann, führt Schreib- oder Lesezugriffe durch. Schreibzugriffe werden mit w(P) und Lesezugriffe mit r(P) abgekürzt, wobei der Parameter P ein logisches Datenobjekt oder ein Replikat bezeichnet. Mit w(O^2) ist z.B. gemeint, dass das logische Datenobjekt O^2 geschrieben wird. In der Abbildung 4.1 ist der Schreibzugriff w(O^2), den ein Client an den Replikationsmanager stellt, beispielhaft an die vom Replikationsmanager angebotene Schnittstelle dargestellt.

Da die Replikation transparent sein soll (siehe Abschnitt 3.1), erfolgen Zugriffe nicht direkt auf physische Datenobjekte, sondern Zugriffe eines Clients werden an den Replikationsmanager geleitet, der mittels einer Replikationsstrategie die Zugriffe auf logische Datenobjekte in Zugriffe auf Replikate transformiert. Daher übersetzt der Replikationsmanager den Schreibzugriff w(O^2) in die Menge {w(R_1^2), w(R_3^2), w(R_3^2), w(R_4^2)} von Schreibzugriffen auf die Replikate, d.h. im Beispiel werden die zugehörigen vier Replikate aktualisiert. Dazu benötigt der Replikationsmanager Schnittstellen zu den Komponenten mit Replikat. Die Schreibzugriffe auf die Replikate sind an den von den Komponenten mit Replikat angebotenen Schnittstellen notiert. Wenn der Replikationsmanager bei einem Zugriff mehr als ein Replikat anspricht, dann handelt es sich um eine verteilte Transaktion, so dass vom Replikationsmanager ein geeignetes Transaktionskonzept bzw. Transaktionssystem verwendet werden muss (siehe Abschnitt 2.2).

4.2. Regelbasierte Koordination der Zugriffe

In replizierten Datenbanken ist neben der Synchronisation von Zugriffen nebenläufiger Prozesse zusätzlich eine Koordination der Zugriffe auf Replikate nötig, wofür nach Definition 10 auf Seite 31 eine Replikationsstrategie verantwortlich ist. Während durch die Koordination festgelegt wird, welche Replikate betroffen sind, wird durch die Synchronisation eine transaktionale Verarbeitung gewährleistet. Bei der Implementierung einer Replikationsstrategie beispielsweise in Form eines Replikationsmanagers ist somit ein geeignetes Transaktionskonzept bzw. die Nutzung eines geeigneten Transaktionssystems zu berücksichtigen (siehe Abschnitt 2.2.3). In diesem Abschnitt wird hingegen auf die Koordination fokussiert, d.h. es wird die Verfahrensweise der Replikationsstrategie RegRess vorgestellt.

Von Bedeutung für eine Replikationsstrategie ist das Korrektheitskriterium, das durch die Koordination der Zugriffe erreicht wird. Durch die 1-Kopien-Serialisierbarkeit (siehe Definition 18 auf Seite 34) werden konsistente Zugriffe garantiert, jedoch werden durch dieses Korrektheitskriterium höhere Ansprüche an die Koordination gestellt, so dass es unter Umständen zum Abbruch von Zugriffen kommt. Wenn mit so genannter abgeschwächter Konsistenz (siehe Abschnitt 3.2.3) als Korrektheitskriterium gearbeitet wird, dann werden Inkonsistenzen zumindest temporär toleriert und die Koordination kann im Allgemeinen einfacher gehalten werden. Inwieweit Inkonsistenzen akzeptiert werden, kann z.B. durch spezielle Garantien wie Kohärenzbedingungen (siehe Definition 22 auf Seite 37) festgelegt werden.

Bei RegRess erfolgt die Koordination der Zugriffe regelbasiert, d.h. durch Spezifikation von Regeln und Auswertung (*„Inferenz“*) dieser Regeln zur Laufzeit werden die betroffenen Replikate bei Zugriffen ermittelt. Da zu Gunsten der Verfügbarkeit bzw. Performance auch temporäre Inkonsistenzen toleriert werden, wird abgeschwächte Konsistenz als Korrektheitskriterium verwendet. Im Folgenden wird zunächst die regelbasierte Koordination von RegRess vorgestellt, bevor eine Quantifizierung der abgeschwächten Konsistenz in Abschnitt 4.3 angegeben wird. Angemerkt sei, dass es sich bei der Koordination der Zugriffe und der garantierten Konsistenz um eine wechselseitige Beziehung handelt: Durch die spezifizierten Regeln wird eine bestimmte Konsistenz erreicht. Andererseits, wenn eine bestimmte Konsistenz garantiert werden soll, dann müssen geeignete Regeln spezifiziert werden.

Nachfolgend wird nun die regelbasierte Koordination von RegRess erläutert. Zunächst wird im Abschnitt 4.2.1 allgemein dargestellt, wie der Ablauf bei Zugriffen auf Objekte durchgeführt wird. Hierbei wird angesprochen, in welcher Form und wofür Regeln vergeben werden können. Anschließend wird im Abschnitt 4.2.2 die Koordination von Schreibzugriffen diskutiert, die einen Einfluss auf die Koordination der Lesezugriffe hat, auf die in Abschnitt 4.2.3 eingegangen wird. Die Inferenz der spezifizierten Regeln, der zentrale Punkt bei der Koordination, erfolgt bei Schreib- und Lesezugriffen auf unterschiedliche Weise und führt zu unterschiedlichen Ergebnissen, was in Abschnitt 4.2.4 erläutert wird. Abschließend wird in Abschnitt 4.2.5 das Korrektheitskriterium „Letztendliche Konsistenz“ betrachtet, das von RegRess erreicht wird.

4.2.1. Konfigurierbare, adaptive Replikationsstrategie mittels Regeln

Die Bearbeitung eines Zugriffs, also eines Lese- oder Schreibzugriffs auf ein logisches Datenobjekt in replizierten Datenbanken, kennzeichnet eine Replikationsstrategie. Da sowohl bei einem Lese- als auch bei einem Schreibzugriff mehrere Replikate eines logischen Datenobjekts involviert sein können bzw. müssen, d.h. es ist eine Menge von Replikaten betroffen, bedarf es einer Koordination dieser Zugriffe. Mit Koordination ist im Wesentlichen gemeint, eine zulässige Menge auszuwählen und einen Zugriff auf die Replikate der Menge durchzuführen. Im Allgemeinen unterscheidet sich die Menge der betroffenen Replikate bei Lesezugriffen von der Menge der betroffenen Replikate bei Schreibzugriffen, genauer gesagt, in beiden Fällen kann es je nach Replikationsstrategie mehrere zulässige Mengen geben. Bei den Votierungsverfahren (siehe Abschnitt 3.3.1) sind es z.B. alle gültigen Lese- und Schreibquoren.

Unter einer *„statischen Koordination“* soll hier verstanden werden, dass die Mengen der betroffenen Replikate für alle Zugriffe festgelegt sind. In Abschnitt 3.3 wurden verschiedene „traditionelle“ Replikationsstrategien vorgestellt, die mit einer statischen Koordination arbeiten. Eine *„adaptive Koordination“* bedeutet, dass sich die Mengen im Laufe der Zeit ändern können, d.h. die betroffenen Replikate werden bei einem Zugriff auf Grund irgendeines Ereignisses dynamisch an die neue Situation angepasst. In Abschnitt 3.4.2 wurden entsprechende Replikationsstrategien präsentiert, die die verwandten Arbeiten zu der hier entwickelten Replikationsstrategie RegRess darstellen.

In RegRess erfolgt die Adaption bei jedem Zugriff. Wenn ein Lese- bzw. Schreibzugriff erfolgen soll, dann wird vor dem Zugriff die Menge der betroffenen Replikate bestimmt. Hierdurch wird ein Maximum an Adaption erreicht, weil bei jedem Zugriff situationsabhängig die „bestmögliche“ Menge bestimmt wird. Was nun die bestmögliche Menge ist, hängt einerseits von den Zuständen der beteiligten Rechner bzw. der darauf laufenden Komponenten ab und andererseits von den Prioritäten der Anwender hinsichtlich der Replikationsziele (siehe Abschnitt 4.4). Der Nachteil liegt darin, dass ein erhöhter Aufwand nötig ist, um diese bestmögliche Menge zu bestimmen. Eine detaillierte Betrachtung der Nachteile von RegRess erfolgt am Ende dieses Abschnitts.

In Abbildung 4.2 ist grob der Ablauf als UML-Aktivitätsdiagramm skizziert, der bei der Bearbeitung eines Zugriffs auf ein logisches Datenobjekt von RegRess ausgeführt wird, genauer gesagt, von dem Replikationsmanager, auf dem die Replikationsstrategie RegRess implementiert wurde:

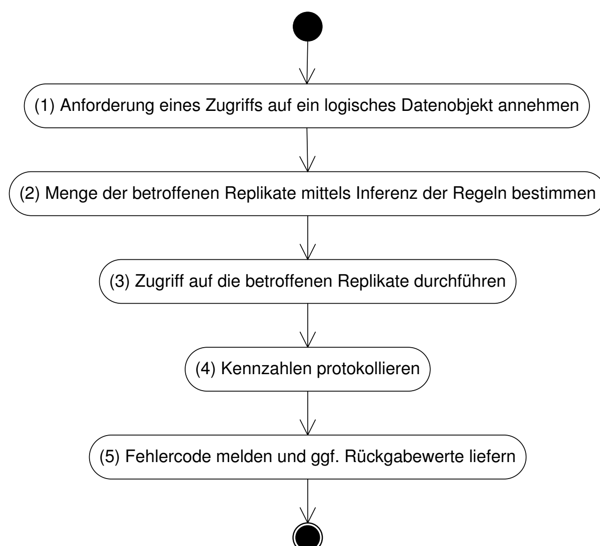

Abbildung 4.2.: Allgemeiner Ablauf eines Zugriffs auf ein logisches Datenobjekt

(1) Zunächst wird eine Anforderung von einem Client (siehe Abbildung 4.1 auf Seite 61) entgegengenommen, die einen Zugriff auf ein logisches Datenobjekt beinhaltet. Das zugrunde liegende Kommunikationsmodell entspricht der synchronen Kommunikation (siehe Abschnitt 2.1.2).
(2) Die Menge der betroffenen Replikate wird ermittelt, in dem eine Inferenz der spezifizierten Regeln durchgeführt wird, d.h. es erfolgt eine Schlußfolgerung auf Basis der Regeln und protokollierter Kennzahlen (siehe auch Aktion (4)).
(3) Der Zugriff auf die Replikate wird durchgeführt, d.h. bei einem Schreibzugriff werden die betroffenen Replikate mit einem per Parameterübergabe gelieferten Wert geschrieben bzw. bei einem Lesezugriff werden zunächst die betroffenen Replikate gelesen und anschließend ein Rückgabewert bestimmt. Letzteres ist dann nötig, wenn mehrere Replikate mit unterschiedlichen Werten gelesen werden. Die Bestimmung des Rückgabewertes wird in Abschnitt 4.2.3 diskutiert.
(4) Nach dem Zugriff werden Kennzahlen protokolliert, z.B. die Performance einer Komponente beim Zugriff auf das Replikat. Diese Kennzahlen können in die Inferenz für nachfolgende Zugriffe einfließen (siehe auch Aktion (2)).
(5) Abschließend kann dem Client der Fehlercode des Zugriffs gemeldet werden, d.h. zum Abschluss der synchronen Kommunikation (siehe auch Aktion (1)) wird dem Client mitgeteilt, ob der Zugriff erfolgreich durchgeführt wurde oder nicht. Im Falle eines Lesezugriffs wird zusätzlich der Rückgabewert zurückgeliefert.

Eine zentrale Bedeutung bei RegRess spielen somit die Regeln, durch die bei der Inferenz die Menge der betroffenen Replikate und damit die Koordination bestimmt wird. Daraus folgt letztendlich die Konsistenz, die der Replikationsstrategie zugrunde liegt (siehe Abschnitt 4.3). Durch die Spezifikation konkreter Regeln ergibt sich eine Konfiguration von RegRess und durch die Inferenz der Regeln zur Laufzeit in Abhängigkeit von protokollierten Kennzahlen eine Adaption, so dass von einer konfigurierbaren, adaptiven Replikationsstrategie gesprochen werden kann. Wie bereits erwähnt, unterscheiden sich im Allgemeinen die Mengen der betroffenen Replikate für Schreibzugriffe von den Mengen der betroffenen Replikate für Lesezugriffe. Daher bietet RegRess die Möglichkeit, für Schreibzugriffe (siehe Abschnitt 4.2.2) und Lesezugriffe (siehe Abschnitt 4.2.3) individuelle Regeln zu vergeben. Zunächst sollen jedoch noch Aspekte diskutiert werden, die in beiden Fällen relevant sind.

Eine gegenseitige Beeinflussung der Koordination von Schreibzugriffen und der Koordination von Lesezugriffen ist insbesondere dann gegeben, wenn durch eine Replikationsstrategie das Korrektheitskriterium 1-Kopien-Serialisierbarkeit erfüllt werden soll. So gibt es z.B. bei den Votierungsverfahren (siehe Abschnitt 3.3.1) die so genannten Überschneidungsregeln: Die Summe der Stimmen von zwei Schreibquoren bzw. einem Schreib- und einem Lesequorum ist größer als die Gesamtanzahl an Stimmen. Vereinfacht gesagt bedeutet dies, dass Schreibzugriffe derart durchgeführt werden, dass hierauf abgestimmte Lesezugriffe mindestens ein aktuelles Replikat liefern. Zwar kann bei RegRess durch eine geeignete Spezifikation von Regeln auch das Korrektheitskriterium 1-Kopien-Serialisierbarkeit erreicht werden, da aber auch abgeschwächte Konsistenz (siehe Abschnitt 3.2.3) toleriert wird, erfolgt keine automatische Abstimmung der Regeln für Schreib- und Lesezugriffe.

Die dynamischen Replikationsstrategien reagieren häufig auf Ausfälle bzw. Wiederherstellung von Komponenten mit Replikat mit einem Wechsel der Replikationsstrategie. Bei der adGSV-Replikationsstrategie [ST06] wird z.B. dann, wenn sich die Anzahl der verfügbaren Komponenten ändert, zu anderen Votierungsverfahren umgeschaltet oder die Stimmen werden neu ausschließlich auf die verfügbaren Komponenten verteilt. Wenn nun weiterhin ein Korrektheitskriterium wie z.B. die 1-Kopien-Serialisierbarkeit eingehalten werden soll, müssen Maßnahmen ergriffen werden, um bei einem Wechsel allen beteiligten Komponenten die neue Replikationsstrategie mitzuteilen. Eine Möglichkeit zur Feststellung eines Wechsels bieten so genannte Epochen [RL93], ein Zähler für den Wechsel von Replikationsstrategien. Bei RegRess ist grundsätzlich nach jedem Zugriff ein Wechsel möglich. Da Kennzahlen zur Verarbeitung eines Zugriffs protokolliert werden, können diese bei folgenden Verarbeitungen von Zugriffen berücksichtigt werden, um ebenfalls ein bestimmtes Konsistenz- bzw. Korrektheitskriterium zu erreichen. Letztendlich ist RegRess jedoch nicht allein auf die Konsistenz fokussiert, sondern bietet die weitergehende Möglichkeit, einen geeigneten Trade-Off hinsichtlich der Replikationsziele Konsistenz, Verfügbarkeit und Performance zu realisieren (siehe Abschnitt 4.4). Daher kann die Reaktion auf eine Veränderung der verfügbaren Komponenten auch eine Veränderung des Korrektheitskriteriums bedeuten, indem entsprechende Regeln spezifiziert werden.

Der Kernpunkt von RegRess ist das regelbasierte System, d.h. die Spezifikation von so genannten Replikationsregeln und die Inferenz dieser Replikationsregeln mit dem Ziel, eine bestmögliche Menge von betroffenen Replikaten für einen Zugriff zu bestimmen. Während die Inferenz eine technische Realisierung darstellt (siehe Kapitel 6), muss die Möglichkeit gegeben sein, Regeln zu formulieren. Hierfür wird in Kapitel 5 die Regelsprache RRML (Replication Rule Markup Language) entwickelt. Welche Typen von Regeln benötigt werden, wird in diesem und den folgenden Abschnitten analysiert. Zunächst werden folgende Fragen diskutiert:

- Wofür werden Regeln spezifiziert?
- Für welche Komponente mit Replikat gelten Regeln?
- Welche Typen von Regeln gibt es?

In Abbildung 4.3 sind Regeln in einem Klassendiagramm der UML zur Klärung der Fragen modelliert. Die Initiierung eines Zugriffs geht zwar immer von einer Komponente aus, ob mit oder ohne Replikat (siehe Abschnitt 4.1.2), aber auch die Daten selbst, auf die zugegriffen werden soll, sind von Bedeutung. So kann es Daten geben, die derart wichtig sind, dass sie immer konsistent vorliegen müssen, d.h. die Replikationskonsistenz (siehe Definition 17 auf Seite 34) oder die 1-Kopien-Serialisierbarkeit (siehe Definition 18 auf Seite 34) ist zu gewährleisten. Ein Beispiel hierfür sind Befunddaten in einem klinischen Informationssystem, von denen lebenswichtige Behandlungen abhängen. Andererseits kann es Daten geben, deren Aktualität keinen so hohen Stellenwert einnimmt, z.B. Adressdaten in einem klinischen Informationssystem. Daher folgt eine Definition der Einheiten, für die Replikationsregeln spezifiziert werden können:

Definition 35 ***Replikationseinheit****: Eine Replikationseinheit (RE) ist entweder eine Komponente, die einen Zugriff auf ein logisches Datenobjekt stellt, oder eine Menge von Objekten.*

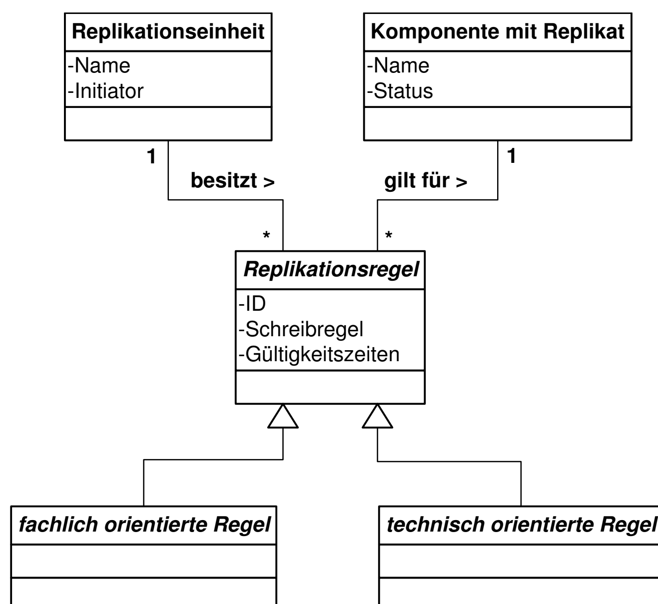

Abbildung 4.3.: Struktur der Replikationsregeln als UML-Klassendiagramm

In Definition 35 sind die oben genannten Forderungen eingebracht: Mit Komponente ist aus Sicht des Replikationsmanagers eine Clientanwendung gemeint (siehe Abbildung 4.1 auf Seite 61), die einen Zugriff auf ein logisches Datenobjekt durchführen möchte. Die Datenbezogenheit wird hier in der objektorientierten Terminologie ausgedrückt, wobei eine Identifizierung der Mengen von Objekten genauso wie der Komponenten möglich sein muss. Beispielsweise werden Objekte mit ähnlichen Eigenschaften zu *„Klassen“* zusammengefasst, wobei nach [SH99] die Menge der aktuell vorhandenen Objekte einer Klasse als *„Objektbehälter“* bezeichnet wird bzw. als *„Instanz“* oder als *„Extension“*. Eine Menge von Objekten kann auch aus einem einzelnen Objekt bestehen. Damit würde z.B. ein Tupel einer relationalen Datenbank als einelementige Menge, also ein einzelnes Objekt, und eine Relation einer relationalen Datenbank als Menge von Objekten bzw. Extension abgebildet werden können und somit der Definition einer Replikationseinheit aus [NHHT03] entsprechen: *„Eine Replikationseinheit ist entweder ein System, das eine Replikationsanforderung (Schreib- oder Lesezugriff) stellt, eine Relation oder ein Tupel.“*

Im UML-Klassendiagramm der Abbildung 4.3 ist erkennbar, dass eine Replikationseinheit mehrere Replikationsregeln besitzen kann, wobei diese Regeln für eine Komponente mit Replikat gelten. Zur Identifizierung einer Replikationseinheit dient das Attribut Name. Das Attribut Initiator kennzeichnet eine Komponente oder eine Menge von Objekten. Durch diese Modellierung wird ermöglicht, dass die Komponenten mit Replikat nicht zwingend gleich behandelt werden, sondern dass durch unterschiedliche Regeln unterschiedliche Konsistenzeigenschaften spezifiziert werden können. Somit wird es z.B. möglich, ein Datawarehouse, das möglicherweise nicht immer aktuelle Daten benötigt, hinsichtlich der Konsistenz anders als ein operatives System, das immer aktuelle Daten benötigt, zu behandeln. Auch eine Komponente mit Replikat wird über ein Attribut Name identifiziert. Das Attribut Status besagt, dass Zustände einer Komponente wie z.B. „verfügbar“ gespeichert werden.

Die Identifizierung einer Replikationsregel erfolgt über das Attribut ID. Weiterhin wird attributiert, ob es sich um eine Regel für einen Schreib- oder einen Lesezugriff handelt und welche Gültigkeitszeiten für diese Regel gelten. Im Modell ist die Klasse Replikationsregel als abstrakte Klasse modelliert, d.h. die konkreten Klassen für Regeln ergeben sich durch Spezialisierung. Bevor im Abschnitt 4.3 diese Konkretisierung vorgenommen wird, soll hier eine grobe Klassifizierung der Regeln erfolgen, die im Modell ebenfalls als abstrakte Klassen modelliert sind: Eine Regel ist fachlich orientiert oder sie ist technisch orientiert. Diese Klassifizierung kann auch als

funktionale und nicht-funktionale Einteilung aufgefasst werden, wobei eine exakte Abgrenzung dieser Begriffe mitunter schwer fällt.

Hier sollen unter fachlich orientierten Regeln solche Regeln fallen, die die Replikate als solches betreffen. Damit ist z.B. gemeint, dass Kohärenzbedingungen (siehe Definition 22 auf Seite 37) für Replikate spezifiziert werden. Mit fachlich orientierter Regel soll zum Ausdruck gebracht werden, dass hierbei anwendungsbezogenes Wissen einfließt. Die technisch orientierten Regeln spezifizieren Reaktionen auf Änderungen der Komponenten- bzw. Rechnerzustände. Dabei soll gegenüber bisherigen Ansätzen nicht nur auf Verfügbarkeit (Rechnerausfall) abgestellt werden, sondern auch auf weitere technische Merkmale wie z.B. Performance, d.h. eine Änderung der Replikationsstrategie kann auch dann erfolgen, wenn sich z.B. das Antwortverhalten einer Komponente ändert.

Abschließend wird eine erste grobe Bewertung der Replikationsstrategie RegRess vorgenommen. Der Vorteil liegt in der flexiblen Verarbeitung eines jeden Zugriffs. Durch die Spezifikation geeigneter Regeln kann RegRess einen optimalen Kompromiss hinsichtlich der Replikationsziele für eine Anwendung erreichen. Durch die situationsabhängige Adaption bei jedem Zugriff wird auch zur Laufzeit eine bestmögliche Verarbeitung erzielbar. Ein Nachteil liegt in der Verarbeitungsgeschwindigkeit, weil bei jedem Zugriff zunächst die Menge der betroffenen Replikate bestimmt werden muss. Daher muss die Regelauswertung des Regelinterpreters möglichst performant sein (siehe Kapitel 7). Ein weiterer Nachteil ist darin zu sehen, dass die Regeln und die Kennzahlen „überall" bekannt sein müssen. Dieser Punkt wird in Abschnitt 4.2.5 diskutiert werden. Weiterhin bedeutet es sicherlich einen gewissen Aufwand, die Regeln zu spezifizieren bzw. eine optimale Konfiguration zu finden. Wenn es sich um einen Anwendungsbereich handelt, der eher „einfache Ansprüche" an die Replikation stellt, dann sind passende traditionelle Replikationsstrategien häufig besser geeignet. Wenn ein hohes Maß an Flexibilität gewünscht ist, dann ist der hohe Aufwand für die Spezifikation der Regeln akzeptabel.

4.2.2. Koordination von Schreibzugriffen

Die Koordination von Schreibzugriffen ist im Allgemeinen aufwändiger als die Koordination von Lesezugriffen, weil Konflikte (siehe Definition 14 auf Seite 32) nur in Verbindung mit Schreibzugriffen auftreten können. Daher müssen bei Replikationsstrategien insbesondere bei Schreibzugriffen Maßnahmen getroffen werden, um Konflikte zu vermeiden oder zu behandeln. Auf das Konfliktmanagement von RegRess wird in Abschnitt 4.2.5 eingegangen. Angemerkt sei an dieser Stelle, dass in RegRess Schreib-/Lesekonflikte$_R$ toleriert und Schreib-/Schreibkonflikte$_R$ vermieden werden sollen (siehe Definitionen 15 und 16 auf Seite 33). Somit wird in diesem Abschnitt auf die Koordination in der Art fokussiert, dass die Menge der betroffenen Replikate sowie der Verarbeitungsablauf bei Schreibzugriffen diskutiert wird.

Einführend wird zunächst betrachtet, in welcher Form die Aktualisierung einer Menge von Replikaten, die genau zu einem logischen Datenobjekt gehören, durchgeführt werden kann. Grundsätzlich handelt es sich hierbei um eine verteilte Transaktion, weil mehrere Replikate auf unterschiedlichen Komponenten betroffen sind. Dabei kann die verteilte Transaktion in mehrere Teiltransaktionen aufgeteilt sein, in denen jeweils eine Teilmenge der betroffenen Replikate aktualisiert wird. Die Teiltransaktionen können ebenfalls verteilte Transaktionen sein und zeitversetzt ausgeführt werden. Damit die in einer Teiltransaktion aktualisierten Replikate für andere Transaktionen stets wechselseitig den gleichen Wert repräsentieren, müssen die Atomaritätseigenschaft und die Isolationseigenschaft der ACID-Eigenschaften (siehe Abschnitt 2.2.1) von den Teiltransaktionen gewährleistet werden. Da aber auch die Konsistenz und Dauerhaftigkeit gewährleistet sein soll, wird in der folgenden Definition auf die ACID-Eigenschaften abgestellt:

Definition 36 ***Synchrone Aktualisierung von Replikaten:*** *M sei die Menge der Replikate eines logischen Datenobjekts, die bei einer Änderung des logischen Datenobjekts aktualisiert werden, und T die Transaktion, die die Replikate aus M ändert. T zerfällt in die Teiltransak-*

tionen T_i, $i = 1, 2, 3, ...,$ die die ACID-Eigenschaften gewährleisten. Sei T_1 die Teiltransaktion, die die Änderung des logischen Datenobjekts initiiert und zeitlich vor den Teiltransaktionen T_j, $j = 2, 3, 4, ...$ beendet wird. In einer Teiltransaktion T_i, $i = 1, 2, 3, ...$ wird die Menge M_i an Replikaten aktualisiert, wobei gilt: $M_i \subseteq M$ und $M_i \cup M_j = \emptyset$, $i \neq j$, d.h. M_i sind disjunkte Teilmengen von M. Dann wird die Aktualisierung der Replikate aus M_1 durch T_1 als synchrone Aktualisierung bezeichnet.

Die Definition 36 bringt zum Ausdruck, dass nicht alle Replikate eines logischen Datenobjekts synchron aktualisiert werden müssen, sondern dass die Replikate zeitversetzt aktualisiert werden können. Dabei wird angenommen, dass die Teiltransaktion T_1 die Teiltransaktion ist, die die Änderung initiiert, z.B. indem ein Client (siehe Abbildung 4.1 auf Seite 61) einen Schreibzugriff startet. In T_1 werden die Replikate, die zu der Menge M_1 gehören, aktualisiert. Die Aktualisierung der übrigen Replikate muss vermerkt bzw. gepuffert werden. Dann kann T_1 erfolgreich gemäß den ACID-Eigenschaften beendet werden, d.h. diese Änderungen sind sichtbar, bevor die gesamte Transaktion T beendet ist. Die Teiltransaktionen T_j, $j = 2, 3, 4, ...$ werden erst nach T_1 zeitversetzt im Sinne einer Sagas-Transaktion (siehe Abschnitt 2.2.5) durchgeführt und beendet, wobei hier Überlappungen zulässig sind. In der Literatur wird die Teiltransaktion T_1 auch als *„originäre“* Transaktion bezeichnet. Damit wären die Replikate, die synchron aktualisiert werden, diejenigen, die in der originären Transaktion aktualisiert werden.

Wenn alle Replikate des logischen Datenobjekts synchron aktualisiert werden, dann zerfällt die Transaktion T im Sinne der Definition 36 in genau eine Teiltransaktion. Es ist auch der Fall möglich, dass kein Replikat synchron aktualisiert wird, d.h. die Menge M_1 ist leer. In diesem Fall werden die Aktualisierungen für alle Replikate gepuffert. Zu diesem Zeitpunkt ist gemäß Definition 5 auf Seite 30 kein Replikat aktuell, weil der aktuelle Wert des logischen Datenobjekts in der Transaktion T_1 geändert wurde, dieser Wert aber derzeit nur im Puffer vorliegt.

Es sei angemerkt, dass mit Definition 36 allein keine Aussagen zum Korrektheitskriterium möglich sind. Erst wenn zusätzlich eine „geeignete“ Menge von Replikaten eines logischen Datenobjekts ausgewählt wird, kann ein bestimmtes Korrektheitskriterium von einer Replikationsstrategie garantiert werden (vergleiche die Anmerkungen zu Definition 10 auf Seite 31), wobei ggf. auch die Koordination der Lesezugriffe berücksichtigt werden muss. Die Replikate, die nicht in der Teiltransaktion T_1 aktualisiert werden, werden zeitversetzt aktualisiert, d.h. beim Lesen dieser Replikate vor der Aktualisierung werden veraltete Werte gelesen. Der Zeitversatz kann dabei kurz oder lang sein, weil z.B. eine bestimmte Komponente mit Replikat nur periodisch aktualisiert werden soll. Diese asynchrone Aktualisierung ist somit wie folgt definiert:

Definition 37 ***Asynchrone Aktualisierung von Replikaten****: Die Aktualisierung von Replikaten, die nicht in einer synchronen Aktualisierung geändert werden, wird als asynchrone Aktualisierung bezeichnet.*

In RegRess werden die Vorteile beider Aktualisierungsvarianten kombiniert, d.h. es ist möglich, einige Replikate konsistent zu halten, während andere Replikate temporär inkonsistent sein dürfen. Das grundsätzliche Ziel bei einer Aktualisierung eines logischen Datenobjekts ist es dabei, dass alle Replikate des logischen Datenobjekts aktualisiert werden. Da in dieser Arbeit von einer voll-replizierten Datenbank mit dem Replikationsgrad n ausgegangen wird (siehe Abschnitt 4.1.1), sind bei einem Schreibzugriff auf ein logisches Datenobjekt somit immer n Replikate betroffen. Bei den Schreibzugriffen auf diese n Replikate wird jedoch die Art der Aktualisierung, d.h. ob synchron oder asynchron, durch Inferenz der Replikationsregeln variiert, wodurch die Konsistenz der Replikate bzw. das Korrektheitskriterium der Replikationsstrategie beeinflusst wird.

Die von RegRess gewährleistete Konsistenz wird in Abschnitt 4.3 detailliert diskutiert. Wie bereits erwähnt, kann durch die Spezifikation geeigneter Regeln zwar eine hohe Konsistenz erreicht werden, aber grundsätzlich wird bei RegRess keine 1-Kopien-Serialisierbarkeit (siehe Definition 18 auf Seite 34) angestrebt. Es findet auch keine Abstimmung der Schreibzugriffe mit

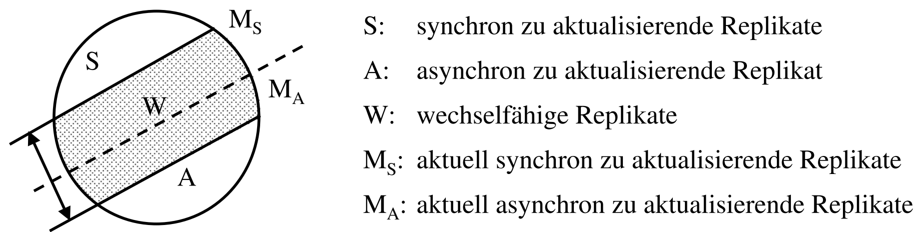

Abbildung 4.4.: Partitionierung in synchron und asynchron zu aktualisierende Replikat

den Lesezugriffen statt, wie es beispielsweise bei den Votierungsverfahren durch die Überschneidungsregeln geschieht (siehe Abschnitt 3.3.1). Da aber alle Schreibzugriffe an alle Komponenten mit Replikat gesendet werden, sofern keine dieser Komponenten dauerhaft ausfällt, und jedes Replikat eines logischen Datenobjekts dann den gleichen aktuellen Wert repräsentieren soll, wird von RegRess zumindest das Korrektheitskriterium „Letztendliche Konsistenz“ eingehalten (siehe Abschnitt 4.2.5).

In Abhängigkeit der spezifizierten Regeln sind detailliertere Aussagen zur Konsistenz möglich. Welche Regeln erlaubt sind, wird in Abschnitt 4.3 bei den Ausführungen zur Konsistenz festgelegt. In diesem Abschnitt wird abschließend die Partitionierung der betroffenen Replikate bei einem Schreibzugriff in synchron und asynchron zu aktualisierende Replikate und der Ablauf der synchronen bzw. asynchronen Aktualisierung spezifiziert.

In Abbildung 4.4 ist die Partitionierung der betroffenen Replikate bei einem Schreibzugriff auf ein logisches Datenobjekt illustriert. Die Menge M der betroffenen Replikate enthält genau n Elemente bzw. Replikate, weil von einer voll-replizierten Datenbank mit Replikationsgrad n ausgegangen wird. Der Ansatz von RegRess ist es nun, dass von diesen n Replikaten eine Teilmenge S stets synchron aktualisiert wird, um immer den aktuellen Wert zu repräsentieren, eine Teilmenge A stets asynchron aktualisiert wird, weil hier die Aktualität nicht erste Priorität hat, und eine Teilmenge W die restlichen Replikate enthält, die *„wechselfähig“* sind, d.h. die entweder synchron oder asynchron aktualisiert werden können. Die Mengen S, A und W sind disjunkt und es gilt:

(4.1) $$M = S \cup A \cup W$$

Die Mengen S und A können auch leer sein und W sollte zumindest für einige Replikationseinheiten ungleich der leeren Menge sein, damit überhaupt eine Adaption stattfindet. Die Adaption wird durch Inferenz der Replikationsregeln erreicht, d.h. die wechselfähigen Replikate werden in Abhängigkeit der aktuellen Zustände entweder den synchron oder den asynchron zu aktualisierenden Replikaten zugeschlagen. Das Ergebnis ist also eine Partitionierung der n betroffenen Replikate in die Menge M_S für die synchron zu aktualisierenden Replikate und M_A für die asynchron zu aktualisierenden Replikate. Somit sind M_S und M_A disjunkt und es gilt:

(4.2) $$M = S \cup A \cup W = M_S \cup M_A$$

(4.3) $$S \subseteq M_S \subseteq S \cup W \text{ und } A \subseteq M_A \subseteq A \cup W$$

Im Aktivitätsdiagramm für einen allgemeinen Ablauf bei Zugriffen (siehe Abbildung 4.2 auf Seite 64) ist dargestellt, dass in Aktion (2) die Menge der betroffenen Replikate bestimmt wird. Für Schreibzugriffe bei RegRess gilt also, dass grundsätzlich alle Replikate eines logischen Datenobjekts betroffen sind. Die zu bestimmende Menge liegt somit fest, jedoch muss diese Menge in die Mengen M_S und M_A für die synchron bzw. asynchron zu aktualisierenden Replikate partitioniert werden. Diese Partitionierung wird durch Inferenz, d.h. durch Auswertung und Schlussfolgerung der spezifizierten Regeln, vorgenommen.

Weil sich die Verarbeitung der synchronen Aktualisierung von der Verarbeitung der asynchronen Aktualisierung unterscheidet, werden beide Abläufe getrennt beschrieben. Die Beschreibung der Abläufe, genauer gesagt der Protokolle bzw. Algorithmen, erfolgt in einer an die Programmiersprache Java angelehnte Pseudo-Sprache, insbesondere werden die Kontrollelemente von Java genutzt. Auch die Fehlerbehandlung von Java wird übernommen: Ein Fehler (exception), der in einem **try**-Block auftritt, kann am Ende des **try**-Blocks durch einen **catch**-Block behandelt werden. Unterschiedliche Fehler werden in verschiedenen **catch**-Blöcken behandelt.

Die synchrone Aktualisierung wird von einem Client (siehe Abbildung 4.1 auf Seite 61) initiiert, der einen Schreibzugriff auf ein logisches Datenobjekt durchführen möchte. Dabei sind die Transaktionsgrenzen der verteilten Transaktion T zu beachten, die durch den Schreibzugriff abgearbeitet wird. Es ist zu unterscheiden, ob der Schreibzugriff die einzige Operation der Client-Transaktion ist, also der Transaktion des Client, die den Schreibzugriff als Operation beinhaltet, oder ob die Client-Transaktion weitere Schreib- bzw. Lesezugriffe beinhaltet. Falls die Client-Transaktion genau aus einer Operation besteht, dann entspricht T der Client-Transaktion. Falls die Client-Transaktion aus mehreren Operationen besteht, dann entspricht T einer Teiltransaktion der Client-Transaktion gemäß einer Teiltransaktion einer geschlossen-geschachtelten Transaktion (siehe Abschnitt 2.2.5). Diese (Teil-)Transaktion T kann gestartet und beendet werden, insbesondere isoliert zurückgesetzt werden. Bei den folgenden Ablaufbeschreibungen wird von dieser Unterscheidung abstrahiert, indem vorausgesetzt wird, dass ein Transaktionsmanager beim Start und beim Beenden der Transaktion T eine geeignete Transaktionsverarbeitung aktiviert (siehe Abschnitt 6.3).

In Listing 4.1 ist die synchrone Aktualisierung der Replikate bei einem Schreibzugriff auf ein logisches Datenobjekt als Pseudocode spezifiziert. Um Daten zu ändern, ruft ein Client die Methode `schreibzugriffSynchron` mit den Übergabeparametern `initiator`, `logObjekt` und `daten` auf. Mit dem Objekt `initiator` wird die initiierende Komponente und mit dem Objekt `logObjekt` das zu schreibenden, logische Objekt bekannt gegeben. Das Objekt `daten` enthält die Werte, die geschrieben werden sollen. Geliefert wird entweder ein Fehlercode oder ein Erfolg als Objekt. Aus Sicht des Clients ist ein Schreibzugriff genau dann erfolgreich ausgeführt, wenn die synchron zu aktualisierenden Replikate geschrieben sind und die Aufträge für die asynchron zu aktualisierenden Replikate in die Warteschlange, die *„Replica Queue“*, eingereiht wurden, wobei eine korrekte transaktionale Verarbeitung eingehalten werden muss. Sowohl im Erfolgsfall als auch bei Abbruch durch Fehler werden Kennzahlen protokolliert, die in nachfolgenden Inferenzen einfließen können.

Beschreibung des Protokolls für den synchronen Schreibzugriff (Listing 4.1)

Zeile 5 Es werden die Listen M_S für die synchron zu aktualisierenden Replikate und M_A für die asynchron zu aktualisierenden Replikate deklariert. Bei `List` handelt es sich um eine Liste von Objekten.

Zeile 7 Im Bedingungsteil der `if`-Anweisung wird eine Sperre auf das zu schreibende, logische Objekt beantragt, wobei die Verwaltung der Sperren für das verteilte System eindeutig realisiert sein muss. Entweder kann eine Sperre auf das logische Objekt gesetzt werden oder nicht, z.B. durch Zeitablauf.

Wenn keine Sperre gesetzt werden konnte, dann wird der Schreibzugriff mit dem Fehlercode `FehlerSperre`, einer Konstanten, abgebrochen. Durch diesen Abbruch werden Schreib-/Schreibkonflikte$_R$ vermieden (siehe oben).

Zeile 8 Mittels Inferenz werden die zu aktualisierenden Replikate in die Mengen M_S für die synchron und M_A für die asynchron zu aktualisierenden Replikate partitioniert (Rückgabeparameter). Die Inferenz liefert in jedem Fall ein gültiges Ergebnis. Bei Zeitablauf oder Fehler während der Inferenz müssen alle Replikate synchron aktualisiert werden (siehe Abschnitt 4.2.4).

Listing 4.1: Synchroner Schreibzugriff

```
1  public Object schreibzugriffSynchron(Object initiator,
2                                       Object logObjekt,
3                                       Object daten) {
4
5    List MS, MA;
6
7    if (not setzeSperre(logObjekt) ) return FehlerSperre;
8    ermittlePartitionierungMittelsInferenz(initiator,logObjekt,MS,MA);
9    if (üprfeChronologieVerletzung(MS,MA) ) return FehlerChronologie;
10
11   Object status := Zugriff;
12   while (status = Zugriff) {
13     try {
14       starteVerteilteTransaktion();
15       aktualisiereReplikateSynchron(MS,logObjekt,daten);ü
16       flleReplicaQueue(MA,initiator,logObjekt,daten);
17       beendeTransaktionErfolgreich();
18       status := Erfolg;
19     }
20     catch (ReplikatKonnteNichtSynchronAktualisiertWerden) {
21       Object R := gibFehlerhaftesReplikat();
22       if (üäprfeWechselfhigkeit(R) )
23         wechseleReplikat(R,MS,MA);
24       else
25         status := FehlerSynchronesReplikat;
26     }
27     catch (TransaktionAnderweitigFehlgeschlagen) {
28         status := FehlerTransaktion;
29     }
30     if (status <> Erfolg) ürolleTransaktionZurck();
31   }
32
33   gibSperreFrei(logObjekt);
34   protokolliereKennzahlen();
35   return status;
36 }
```

Zeile 9 Die Menge M_S der synchron zu aktualisierenden Replikate wird auf Verletzung der Chronologie geprüft. Die Chronologie ist genau dann verletzt, wenn für ein zwingend synchron zu aktualisierendes Replikat in M_S noch Aufträge in der Replica Queue anstehen. Wenn für ein wechselfähiges Replikat in M_S noch Aufträge in der Replica Queue anstehen, dann wird das Replikat bei der Prüfung den asynchron zu aktualisierenden Replikaten M_A zugeordnet.

Wenn die Chronologie verletzt ist, dann wird der Schreibzugriff mit dem Fehlercode `FehlerChronologie` abgebrochen. Hierdurch wird gewährleistet, dass Schreibzugriffe in der gleichen Reihenfolge bei jedem Replikat eines logischen Objekts erfolgen und damit das Korrektheitskriterium Konvergenz erreicht werden kann.

Zeile 11 Nachdem die Vorbedingungen „Sperre auf logisches Objekt erhalten“, „Partitionierung gültig“ und „Chronologie nicht verletzt“ erfüllt sind, kann der Schreibzugriff durchgeführt werden. Hierfür wird zunächst eine Statusvariable initialisiert, die neben der Speicherung des Bearbeitungsstatus auch zur Kontrolle der Schleifendurchläufe (siehe Zeile 12) dient.

Zeile 12 Die Bearbeitung des Schreibzugriffs auf die Replikate wird in einer Schleife durchgeführt, um beim nachfolgend beschriebenen Fehler erneut aufsetzen zu können. Eine Wiederholung findet dann und nur dann statt, wenn wechselfähige Replikate, bei denen Fehler während des synchronen Schreibzugriffs aufgetreten sind, den asynchron zu aktualisierenden Replikaten zugeordnet werden konnten. Die Bearbeitung des Schreibzugriffs in der Schleife terminiert entweder bei erfolgreicher Bearbeitung oder bei Auftreten eines sonstigen Fehlers durch Setzen der Statusvariablen. Die Bearbeitung wird höchstens so oft wiederholt, wie es wechselfähige Replikate in der Menge M_S der synchron zu aktualisierenden Replikate gibt.

Zeile 13 Die nachfolgenden Anweisungen in den Zeilen 14 bis 18 werden in einem **try**-Block ausgeführt, der eine anschließende Fehlerbehandlung für im **try**-Block aufgetretenen Fehler erlaubt. Dabei wird auf die Fehler „synchron zu aktualisierendes Replikat konnte nicht geschrieben werden“ und „Transaktion anderweitig fehlgeschlagen“ reagiert. Bei Auftreten eines Fehlers wird direkt zur Fehlerbehandlung im entsprechenden **catch**-Block gesprungen (siehe unten).

Zeile 14 Der Start der (Teil-)Transaktion wird dem Transaktionssystem mitgeteilt (siehe oben).

Zeile 15 Die synchron zu aktualisierenden Replikate werden innerhalb der (Teil-)Transaktion aktualisiert.

Zeile 16 Für die asynchron zu aktualisierenden Replikate werden innerhalb der (Teil-)Transaktion entsprechende Aufträge in die Replica Queue geschrieben.

Zeile 17 Die (Teil-)Transaktion wird erfolgreich abgeschlossen („Commit“).

Zeile 18 Das erfolgreiche Ende des Schreibzugriffs wird in der Statusvariablen gesetzt.

Zeile 20 Die erste Fehlerbehandlung im **catch**-Block in den Zeilen 21 bis 25 betrifft den Fehler, dass ein synchron zu aktualisierendes Replikat nicht geschrieben werden konnte. Wenn es sich hierbei um ein wechselfähiges Replikat handelt, kann der Schreibzugriff erneut aufgesetzt werden, wobei das entsprechende Replikat nun den asynchron zu aktualisierenden Replikaten M_A zugeordnet wird.

Zeile 21 Die Objektvariable `R` wird mit dem fehlerhaften Replikat initialisiert, das bei der synchronen Aktualisierung nicht geschrieben werden konnte.

Zeile 22 Es wird geprüft, ob das Replikat `R` wechselfähig ist.

Zeile 23 Wenn das Replikat `R` wechselfähig ist, dann wechselt `R` von der Menge M_S zur Menge M_A.

Zeile 25 Wenn das Replikat nicht wechselfähig ist, dann wird die Fehlerbehandlung und damit der Schreibzugriff durch Setzen der Statusvariablen mit dem Fehlercode `FehlerSynchronesReplikat` beendet.

Zeile 27 Bei der zweiten Fehlerbehandlung im **catch**-Block in der Zeile 28 werden alle sonstigen Fehler behandelt, die im **try**-Block während der Bearbeitung der (Teil-)Transaktion aufgetreten sind und die nicht das Schreiben eines synchron zu aktualisierenden Replikats betreffen. Diese Fehler führen zwingend zum Abbruch des Schreibzugriffs.

Zeile 28 Wenn die Transaktion anderweitig abgebrochen wurde, dann wird die Fehlerbehandlung und damit der Schreibzugriff durch Setzen der Statusvariablen mit dem Fehlercode `FehlerTransaktion` beendet.

Zeile 30 Wenn der Schreibzugriff nicht erfolgreich durchgeführt werden konnte, dann wird die im **try**-Block gestartete (Teil-)Transaktion zurückgerollt. Das ist genau dann der Fall, wenn die Statusvariable einen Fehlercode repräsentiert oder weiterhin auf

„Zugriff“ gesetzt ist. Im letzten Fall wird der Schreibzugriff iterativ in einer neuen (Teil-)Transaktion aufgesetzt.

Zeile 33 Zum Ende der Bearbeitung eines Schreibzugriffs auf die Replikate eines logischen Objekts werden in den Zeilen 33 bis 35 abschließende Anweisungen ausgeführt. Zunächst wird die Sperre auf das logische Objekt freigegeben.

Zeile 34 Es werden Kennzahlen für den erfolgreichen bzw. misslungenen Schreibzugriff protokolliert, die ggf. in folgende Inferenzen einfließen.

Zeile 35 Dem Client wird der Status als Rückgabewert für die Bearbeitung des Schreibzugriffs geliefert.

Die Replica Queue sowie das Einreihen (engl. enqueue) und das Entfernen (engl. dequeue) der Aufträge in bzw. aus dieser Warteschlange bestimmen zusammen mit der Koordination der Zugriffe und der Synchronisation der Zugriffe das Korrektheitskriterium von RegRess, worauf detailliert in Abschnitt 4.2.5 eingegangen wird. Während beim synchronen Aktualisieren ausschließlich Aufträge in die Replica Queue eingereiht werden, um sich die noch nicht aktualisierten Replikate zu merken, werden ausschließlich beim asynchronen Aktualisieren Aufträge aus der Replica Queue entfernt. Dabei kann unterschieden werden, ob alle Aufträge eines Replikats abgearbeitet werden oder nur so viele, um eine Grenze einzuhalten, die das asynchron zu aktualisierende Replikat vom aktuellen Replikat abweichen darf. Wenn sich z.B. fünf Aufträge für ein Replikat in der Replica Queue befinden und dieses Replikat drei Aktualisierungen verspätet sein darf, dann können entweder zwei Aufträge bearbeitet werden, um wieder in der Grenze zu sein, oder alle fünf Aufträge, um das Replikat auf den aktuellen Stand zu bringen (siehe Abschnitt 4.2.4).

Die asynchrone Aktualisierung wird entweder periodisch gestartet oder nach dem erfolgreichem Ende der Client-Transaktion. Letzteres dient dazu, um ggf. zeitnah nach einer synchronen Aktualisierung auch die asynchron zu aktualisierenden Replikate zu ändern. In Listing 4.2 ist die asynchrone Aktualisierung der Replikate spezifiziert. Wenn Aufträge in der Replica Queue anstehen, dann werden diese Aufträge jeweils in einer Schleife bearbeitet. Angemerkt dazu sei, dass die Replica Queue nicht wie eine klassische Warteschlange z.B. nach dem FIFO- oder LIFO-Prinzip arbeitet. Zwar werden beim Einreihen die Aufträge ans Ende gestellt, aber auf die Aufträge kann beim Lesen und beim Entfernen indiziert im Sinne eines Feldes zugegriffen werden. Hierdurch wird erreicht, dass ein Auftrag, der derzeit nicht bearbeitet werden kann, keine anderen Aufträge blockiert. Wenn klassische Warteschlangen bei der Implementierung verwendet werden sollen, kann die Blockierung z.B. dadurch verhindert oder abgeschwächt werden, indem für jede Komponente mit Replikat eine eigene Warteschlange aufgesetzt wird.

Beschreibung des Protokolls für den asynchronen Schreibzugriff (Listing 4.2)

Zeile 3 Es wird eine Liste deklariert und als leer initialisiert, mit der festgehalten wird, ob alle Aufträge eines Replikats abgearbeitet werden sollen. Eine solche Liste ist nötig, weil eine erneute Inferenz für das gleiche Replikat ein anderes Ergebnis liefern kann als bei der vorhergehenden Inferenz, weil z.B. eine Grenze für Abweichungen bei der erneuten Inferenz nach Abarbeitung eines Auftrags wieder eingehalten wird.

Zeile 5 Die asynchrone Aktualisierung, d.h. die „verspätete“ Aktualisierung eines Replikats, wird iterativ in einer For-Schleife für jeden Auftrag der Replica Queue einzeln durchgeführt. Die Anzahl der Aufträge bestimmt somit die Anzahl der Durchläufe, wobei auch null Durchläufe zulässig sind, d.h. es befindet sich kein Auftrag in der Replica Queue.

Zeile 6 Der i-te Auftrag wird in der i-ten Iteration aus der Replica Queue gelesen. Angemerkt sei, dass Lesen aus der Replica Queue kein Entfernen meint. In der Objektvariablen `auftrag` sind folgende Informationen gespeichert: `replikat` beschreibt,

Listing 4.2: Asynchroner Schreibzugriff

```
public void schreibzugriffAsynchron() {

  List alleAuftraegeEinesReplikats := Null;

  for(int i; i < gibAnzahlElementeInReplicaQueue(); i++) {
    Object auftrag := liesAuftragAusReplicaQueue(i);

    if (üprfeChronologieVerletzung(auftrag.replikat) ) break;

    if (not inListe(alleAuftraegeEinesReplikats,auftrag.replikat) {
      Object schreiben := ermittleZugriff(auftrag.initiator,
                                           auftrag.logObjekt);
      if (schreiben = KeinZugriff) break;
      if (schreiben = AlleAuftraege)
        anhaengen(alleAuftraegeEinesReplikats,auftrag.replikat);
    }
    try {
      starteVerteilteTransaktion();
      schreibeReplikat(auftrag.replikat,auftrag.daten);
      entferneReplicaQueue(auftrag);
      beendeTransaktionErfolgreich();
    }
    catch (TransaktionFehlgeschlagen) {ü
      rolleTransaktionZurck();
    }
    protokolliereKennzahlen();
  }
  return;
}
```

welches Replikat geändert werden soll, `initiator` speichert die initiierende Komponente, `logObjekt` enthält das zu schreibende, logische Objekt und `daten` sind die Daten, die geschrieben werden sollen.

Zeile 8 Im Bedingungsteil der `if`-Anweisung wird geprüft, ob das zu schreibende Replikat die Chronologie verletzt. Das Replikat `auftrag.replikat` des i-ten Auftrags verletzt genau dann die Chronologie, wenn es einen Auftrag A_j gibt und folgendes gilt: A_j und A_i schreiben das gleiche Replikat R und $j < i$. Wenn die Chronologie verletzt ist, dann wird die Bearbeitung des i-ten Auftrags mittels `break` übersprungen.

Zeile 10 Es wird geprüft, ob sich das zu aktualisierende Replikat `auftrag.replikat` in der Liste `alleAuftraegeEinesReplikats` befindet. Wenn das Replikat in der Liste enthalten ist, dann wurde bereits eine Inferenz durchgeführt und das Replikat kann geschrieben werden. Andernfalls ist eine Inferenz nötig, um die derzeitige Schreiberlaubnis zu prüfen.

Zeile 11 Mittels Inferenz wird geprüft, ob das zu schreibende Replikat derzeit aktualisiert werden darf. Die Inferenz terminiert in jedem Fall und liefert ein gültiges Ergebnis (siehe Abschnitt 4.2.4). Gültige Ergebnisse der Inferenz sind: `KeinZugriff`, d.h. derzeit darf das Replikat nicht geschrieben werden, `EinAuftrag`, d.h. genau der i-te Auftrag darf bearbeitet werden oder `AlleAuftraege`, d.h. alle Aufträge dieses Replikats dürfen bearbeitet werden.

Zeile 13 Wenn ein Schreibzugriff auf das Replikat des i-ten Auftrags derzeit nicht erlaubt ist, dann wird die Bearbeitung des i-ten Auftrags mittels `break` übersprungen.

Zeile 14 Wenn alle Aufträge des Replikats bearbeitet werden sollen, dann wird das Replikat `auftrag.replikat` an die Liste `alleAuftraegeEinesReplikats` angehängt. Dadurch wird eine erneute Inferenz unterbunden (siehe Zeile 10).

Zeile 17 Die nachfolgenden Anweisungen in den Zeilen 18 bis 21 werden in einem **try**-Block ausgeführt, der eine anschließende Fehlerbehandlung für im **try**-Block aufgetretene Fehler erlaubt. Dabei wird keine Unterscheidung von Fehlern vorgenommen, sondern allgemein auf den Fehler „Transaktion fehlgeschlagen“ reagiert. Bei Auftreten eines Fehlers wird direkt zur Fehlerbehandlung im **catch**-Block gesprungen (siehe unten).

Zeile 18 Der Start der verteilten Transaktion wird dem Transaktionssystem mitgeteilt. Es handelt sich um eine verteilte Transaktion, weil sowohl ein Replikat geschrieben wird als auch ein Auftrag aus der Replica Queue entfernt wird (siehe Abschnitt 2.2.5).

Zeile 19 Das Replikat des i-ten Auftrags wird mit den im Auftrag gespeicherten Daten geschrieben.

Zeile 20 Der i-te Auftrag wird aus der Replica Queue entfernt. Die Entfernung eines Auftrags aus der Replica Queue darf die Indizierung nicht beeinflussen.

Zeile 21 Die verteilte Transaktion wird erfolgreich abgeschlossen („Commit“).

Zeile 23 Bei der Fehlerbehandlung im **catch**-Block in der Zeile 24 werden mögliche Fehler behandelt, die während der verteilten Transaktion des **try**-Blocks auftreten können.

Zeile 24 Wenn ein Fehler aufgetreten ist, dann wird die Transaktion zurückgerollt, d.h. das Replikat wurde nicht geändert und der i-te Auftrag verbleibt in der Replica Queue.

Zeile 26 Es werden Kennzahlen für den erfolgreichen bzw. misslungenen Schreibzugriff protokolliert, die ggf. in folgende Inferenzen einfließen.

Zeile 28 Nachdem alle Aufträge einmal bearbeitet wurden, ob erfolgreich oder nicht, wird die asynchrone Aktualisierung beendet. Verbliebene Aufträge in der Replica Queue werden bei einem späteren Aufruf der asynchronen Aktualisierung erneut bearbeitet.

Sowohl bei der synchronen als auch bei der asynchronen Aktualisierung wird geprüft, ob eine Verletzung der chronologischen Verarbeitung vorliegt. Dabei speichert die Replica Queue alle ausgelassenen Aktualisierungen. Somit können in der Replica Queue für ein und dasselbe Replikat mehrere Aktualisierungen anstehen. Die asynchrone Aktualisierung ist derart gestaltet, dass alle Aufträge eines Replikats in der chronologischen Reihenfolge verarbeitet werden (siehe Zeile 8 des Listings 4.2), d.h. jedes Replikat erfährt jede Aktualisierung seines logischen Objekts. Eine Vereinfachung kann dadurch erreicht werden, dass nur die letzte Aktualisierung eines Replikats aus der Replica Queue bearbeitet wird und die vorhergehenden Aktualisierungen des Replikats, die ja mit der letzten Aktualisierung überschrieben werden, ausgelassen werden. Eine derartige Vorgehensweise mit den notwendigen Bedingungen wird in Kapitel 9 diskutiert.

Eine genauere Bestimmung der gewährleisteten Konsistenz von RegRess hängt, wie bereits erwähnt, von den spezifizierten Regeln ab. Hierauf wird in Abschnitt 4.3 eingegangen. Da allgemein bei Replikationsstrategien das Zusammenspiel der Koordination von Schreibzugriffen und der Koordination von Lesezugriffen das zugrunde liegende Korrektheitskriterium bestimmt, wird zunächst in Abschnitt 4.2.3 auf Lesezugriffe eingegangen.

4.2.3. Koordination von Lesezugriffen

Bei vielen Replikationsstrategien gestaltet sich die Koordination der Lesezugriffe einfacher als die Koordination der Schreibzugriffe. Häufig wird auf eine „echte“ Koordination verzichtet, indem dem Client ein Lesezugriff auf ein beliebiges Replikat gestattet wird, sei es, weil bekannt ist, dass alle Replikate konsistent sind, wie z.B. beim ROWA-Verfahren (siehe Abschnitt 3.3.1), oder weil inkonsistentes Lesen toleriert wird, wie z.B. beim Snapshot-Verfahren (siehe Abschnitt 3.3.2).

Beim Lesezugriff auf ein beliebiges Replikat wird im Allgemeinen ein „lokales“ Replikat vom Client gewählt, d.h. der Client kontaktiert die Komponente mit Replikat, mit der er „direkt“ verbunden ist. Da bei RegRess auch abgeschwächte Konsistenz, also das Lesen veralteter Daten, toleriert wird, ist das Lesen beliebiger bzw. lokaler Replikate zulässig.

Bei den Votierungsverfahren (siehe Abschnitt 3.3.1) hingegen muss im Allgemeinen auch beim Lesen auf mehrere Replikate zugegriffen werden, wobei die Menge der Replikate durch geeignete Lesequoren festgelegt ist. Ein konsistentes Replikat wird innerhalb des Lesequorums an dem aktuellsten Zeitstempel erkannt (siehe Abschnitt 2.1.1), d.h. bei Schreibzugriffen muss neben den Attributwerten auch ein Zeitstempel geschrieben werden. Dadurch ist es den Votierungsverfahren möglich, die 1-Kopien-Serialisierbarkeit zu gewährleisten.

Der Zeitverzug eines veralteten Replikats gegenüber dem aktuellen Replikat des selben logischen Objekts wird ebenfalls mittels Zeitstempel berechnet. Somit können bei RegRess Zeitstempel entweder bei der asynchronen Aktualisierung benötigt werden, um den zulässigen Zeitverzug festzulegen, oder bei Lesezugriffen, um Replikate mit einem bestimmten Alter zu identifizieren. Bei RegRess werden hierfür in den Regeln Funktionen definiert (siehe Abschnitt 5.3), die auf Zeitstempel basieren. Wenn die beteiligten Komponenten mit Replikat keine Zeitstempel bieten, dann muss auf die entsprechenden Funktionen in Regeln verzichtet werden. Damit verbleiben zwei Varianten der Koordination von Lesezugriffen in RegRess:

1. Es wird ein beliebiges Replikat gelesen.
2. Es wird über RegRess genau ein Replikat aus der Menge der Replikate geliefert, die bestimmten Eigenschaften genügen.

Im ersten Fall ist keine Inferenz von Regeln durch RegRess nötig, d.h. ein Replikationsmanager muss nicht kontaktiert werden. Durch die asynchrone Aktualisierung ist es jedoch möglich, dass veraltete Replikate gelesen werden. Diese Schreib-/Lesekonflikte$_R$ werden bei RegRess toleriert. Das Alter eines Replikats bzw. die Datenfrische (siehe Definition 24 auf Seite 38) hängt dann von den spezifizierten Regeln für Schreibzugriffe ab und wird in Abschnitt 4.3.3 diskutiert.

Die zweite Variante besteht darin, von RegRess ein Replikat ermitteln zu lassen, das bestimmten Eigenschaften genügt. Diese Eigenschaften, z.B. die Performance bei einem Lesezugriff, werden bei der Inferenz als Parameter verwendet. In Listing 4.3 ist der Ablauf bei einem Lesezugriff über RegRess spezifiziert. Um ein Replikat mit bestimmten Eigenschaften zu lesen, ruft ein Client die Methode `lesezugriff` mit den Parametern `initiator`, `logObjekt` und `eigenschaften` auf. Mit dem Objekt `initiator` wird die initiierende Komponente und dem Objekt `logObjekt` das logische Objekt, das gelesen werden soll, übergeben. Das Objekt `eigenschaften` beschreibt die gewünschten Eigenschaften, die das zu lesende Replikat erfüllen muss.

Aus Sicht des Clients ist der Lesezugriff analog zum Schreibzugriff (siehe Listing 4.1) eine Teiltransaktion, die in eine übergeordnete Client-Transaktion eingebettet ist und isoliert zurückgesetzt werden kann. Da bei einem Lesezugriff aber nur jeweils auf eine Komponente zugegriffen wird, handelt es sich bei dieser Teiltransaktion um eine flache Transaktion. Häufig wird bei Lesezugriffen, insbesondere bei reinen Lesetransaktionen auf die Weitergabe des Transaktionskontextes verzichtet. Hier aber soll der Transaktionskontext bei der Spezifikation berücksichtigt werden, um z.B. die Nebenläufigkeitsanomalie „inkonsistente Analyse“ (non-repeatable read) zu verhindern (siehe Abschnitt 2.2.3).

Beschreibung des Protokolls für den Lesezugriff (Listing 4.3)

Zeile 5 Mittels Inferenz wird die Menge M_L passender Replikate ermittelt, d.h. die Menge der Replikate, die den Kriterien `eigenschaften` genügen. Die Inferenz liefert in jedem Fall eine gültige Menge M_L (siehe Abschnitt 4.2.4), wobei von einer unsortierten Menge mit zufälliger Reihenfolge ausgegangen wird. Die Vor- und Nachteile einer sortierten bzw. unsortierten Menge wird in Kapitel 9 diskutiert.

Listing 4.3: Lesezugriff

```
public Object lesezugriff(Object initiator,
                          Object logObjekt,
                          Object eigenschaften) {

  List M_L := ermittleLeseReplikateMittelsInferenz(initiator,
                                          logObjekt,eigenschaften);
  if (gibAnzahl(M_L) = 0) return FehlerKeinPassendesReplikat;

  for(int i; i < gibAnzahl(M_L); i++) {
    Object replikat := gibReplikat(M_L,i);
    try {
      starteTransaktion;
      liesReplikat(replikat);
      beendeTransaktionErfolgreich();
      protokolliereKennzahlen();
      return replikat;
    }
    catch (TransaktionFehlgeschlagen) {ü
      rolleTransaktionZurck();
      protokolliereKennzahlen();
    }
  }
  return öFehlerZugriffNichtMglich;
}
```

Zeile 7 Wenn die durch die Inferenz bestimmte Menge leer ist, d.h. die Anzahl passender Replikate ist null, dann wird der Lesezugriff abgebrochen und die Fehlermeldung `FehlerKeinPassendesReplikat` geliefert.

Zeile 9 Der Lesezugriff wird iterativ in einer For-Schleife für jedes Replikat in der Menge M_L durchgeführt, wobei ein erfolgreicher Zugriff die Iteration beendet. Die Anzahl der passenden Replikate bestimmt somit die Anzahl der Durchläufe.

Zeile 10 Die Variable `replikat` wird mit dem i-ten Element bzw. Replikat aus der Menge M_L der passenden Replikate gesetzt, d.h. `replikat` repräsentiert das Replikat, das gelesen werden soll.

Zeile 11 Die nachfolgenden Anweisungen in den Zeilen 12 bis 16 werden in einem **try**-Block ausgeführt, der eine anschließende Fehlerbehandlung für im **try**-Block aufgetretene Fehler erlaubt. Dabei wird keine Unterscheidung von Fehlern vorgenommen, sondern allgemein auf den Fehler „Transaktion fehlgeschlagen“ reagiert. Bei Auftreten eines Fehlers wird direkt zur Fehlerbehandlung im **catch**-Block gesprungen (siehe unten).

Zeile 12 Die (Teil-)Transaktion für den Lesezugriff wird gestartet.

Zeile 13 Das Replikat `replikat` wird gelesen.

Zeile 14 Die (Teil-)Transaktion wird erfolgreich abgeschlossen („Commit“).

Zeile 15 Es werden Kennzahlen für den erfolgreichen Lesezugriff protokolliert, die ggf. in nachfolgende Inferenzen einfließen.

Zeile 16 Die Iteration wird beendet, indem das gelesene Replikat dem Client geliefert wird.

Zeile 18 Bei der Fehlerbehandlung im **catch**-Block in den Zeilen 19 bis 20 werden mögliche Fehler behandelt, die während der (Teil-)Transaktion des **try**-Blocks auftreten können.

Zeile 19 Die (Teil-)Transaktion wird zurückgerollt.

Zeile 20 Es werden Kennzahlen für den misslungenen Lesezugriff protokolliert.

Zeile 23 Wenn die Iteration endet, ohne dass ein passendes Replikat erfolgreich gelesen werden konnte, dann wird die Fehlermeldung `FehlerZugriffNichtMöglich` geliefert

Wenn über einen Replikationsmanager mittels RegRess ein Replikat für einen Lesezugriff bestimmt wird, können fachliche und technische Anforderungen berücksichtigt werden (siehe Abschnitt 4.3). Fachliche Anforderungen können z.B. das Alter bzw. die Datenfrische betreffen, d.h. analog zur Replikationsstrategie FAS (siehe Abschnitt 3.4.2) wird ein Replikat bestimmt, das eine spezifizierte Aktualität bzw. Konsistenz aufweist. Technische Anforderungen können z.B. die Performance betreffen, d.h. es wird auf eine Komponente zugegriffen, von der auf Grund protokollierter Kennzahlen von einer bestimmten Verarbeitungsgeschwindigkeit ausgegangen werden kann. Eine detaillierte Betrachtung möglicher Regeln folgt in Abschnitt 4.3.

4.2.4. Inferenz bei Schreib- und Lesezugriffen

In den Abschnitten 4.2.2 und 4.2.3 wurden die Protokolle der hier vorgestellten Replikationsstrategie RegRess für Schreib- und Lesezugriffe spezifiziert, wobei bei Schreibzugriffen zwischen synchronen und asynchronen Schreibzugriffen unterschieden wurde. In den drei Protokollen ist der zentrale Bestandteil die Menge von Replikaten zu bestimmen, auf die zugegriffen werden soll. Da es sich bei RegRess um eine regelbasierte Replikationsstrategie handelt, erfolgt die Bestimmung der betroffenen Replikate eines Zugriffs durch Inferenz auf zuvor spezifizierten Regeln. Welche Regeln spezifiziert werden können, wird in Abschnitt 4.3 diskutiert. In diesem Abschnitt soll zunächst vorgestellt werden, welche Punkte bei der Inferenz grundsätzlich zu beachten sind, ohne dabei auf so genannte *„Inferenzmaschinen“* regelbasierter Systeme (siehe Anhang A) einzugehen. Die Funktionsweise derartiger Inferenzmaschinen hängt vom verwendeten regelbasierten System ab und spielt erst bei der Implementierung eine Rolle (siehe Abschnitt 7.3.2).

Zwar haben alle Regeln, die spezifiziert werden können, die gleiche Struktur (siehe Abbildung 4.3 auf Seite 66), müssen aber auf Grund der unterschiedlichen Anforderungen der drei oben genannten Protokolle bei der Inferenz unterschiedliche Ergebnisse liefern. Daher können drei Arten von Inferenzen unterschieden werden, die in diesem Abschnitt erläutert werden. Dabei werden diese drei Arten wie folgt abgekürzt und klassifiziert:

InferenzSynchron: Inferenz, die beim synchronen Schreibzugriff durchgeführt wird (siehe Zeile 8 des Listings 4.1) und die die zu aktualisierenden Replikate in die Menge M_S für die synchron zu aktualisierenden Replikate und die Menge M_A für die asynchron zu aktualisierenden Replikate partitioniert.

InferenzAsynchron: Inferenz, die beim asynchronen Schreibzugriff durchgeführt wird (siehe Zeile 11 des Listings 4.2) und die prüft, ob der Schreibzugriff auf das asynchron zu aktualisierende Replikat durchgeführt werden darf.

InferenzLesen: Inferenz, die beim Lesezugriff durchgeführt wird (siehe Zeile 5 des Listings 4.3) und die eine Menge von Replikaten ermittelt, wobei die Replikate bestimmte Eigenschaften für den Lesezugriff erfüllen.

Nachfolgend werden Anforderungen festgehalten, die die drei Varianten der Inferenz bei RegRess erfüllen müssen. Diese Anforderungen müssen letztendlich als Regeln spezifiziert werden können, worauf in Kapitel 5 bei der Entwicklung der benötigten Regelsprache eingegangen wird. Was hier unter Regeln verstanden wird, wird in Anhang A.1 erläutert.

Das Ziel der InferenzSynchron ist es, die betroffenen Replikate gemäß den Erläuterungen zu Abbildung 4.4 auf Seite 69 in die synchron und die asynchron zu aktualisierenden Replikate zu partitionieren, d.h. in die Mengen M_S und M_A. Damit zu Beginn der InferenzSynchron

gewährleistet ist, dass alle betroffenen Replikate berücksichtigt werden, wird folgender initialer Startzustand gefordert:

Anforderung 1 ***InferenzSynchron: Startzustand****: Zu Beginn der InferenzSynchron werden alle betroffenen Replikate den synchron zu aktualisierenden Replikaten zugeordnet.*

Durch die Anforderung 1 wird der Schwerpunkt hinsichtlich der Replikationsziele (siehe Abschnitt 3.1) auf die Konsistenz gelegt, d.h. wenn keine weiteren Regeln spezifiziert werden, dann werden alle betroffenen Replikate synchronisiert und damit konsistent geschrieben. Durch die Spezifikation weiterer Regeln, die wie bereits erwähnt in Abschnitt 4.3 diskutiert werden, kann ein einzelnes Replikat oder mehrere Replikate von der Menge M_S zu der Menge M_A oder umgekehrt wechseln. Wichtig dabei ist, dass kein Replikat aus beiden Mengen gleichzeitig gelöscht wird, damit bei der Aktualisierung wirklich alle betroffenen Replikate geschrieben werden. Diese Anforderung wird nicht explizit aufgeführt, sondern dadurch realisiert, dass es keine Anforderung gibt, die ein Löschen ermöglicht.

Die InferenzSynchron soll in jedem Fall eine gültige Partitionierung liefern, d.h. am Ende der Inferenz sind die betroffenen Replikate in die synchron und asynchron zu aktualisierenden Replikate partitioniert:

Anforderung 2 ***InferenzSynchron: Endzustand****: Am Ende der InferenzSynchron liegt eine gültige Partitionierung der betroffenen Replikate in synchron und asynchron zu aktualisierende Replikate vor.*

Das Ergebnis der InferenzSynchron ist also eine auf Grund der Regeln ermittelte Partitionierung, die die betroffenen Replikate einschließt. Ob die aufrufende Komponente die Partitionierung verwendet, obliegt der Komponente selbst. Eine Verletzung der chronologischen Verarbeitung wird z.B. von der aufrufenden Komponente behandelt (siehe Listing 4.1 auf Seite 71). Die InferenzSynchron liefert also keinen undefinierbaren Zustand oder einen Fehlercode an die aufrufende Komponente. Für derartige Fälle werden nachfolgende Anforderungen definiert, die eine gültige Partitionierung in der Form vorsehen, dass alle Replikate synchron zu aktualisieren sind und somit die Konsistenz gewährleistet wird:

Anforderung 3 ***InferenzSynchron: Fehler****: Wenn ein Fehler während der InferenzSynchron auftritt, dann werden alle Replikate den synchron zu aktualisierenden Replikaten zugeordnet.*

Bei der Spezifikation der Protokolle für Schreib- und Lesezugriffe wurde Wert auf die Terminierung der zugehörigen Algorithmen gelegt. Damit geht einher, dass auch die zugehörigen Inferenzen terminieren. Wie auch bei den anderen Varianten der Inferenz soll hier die Reaktion auf eine erzwungene Terminierung, d.h. eine Terminierung der Inferenz durch Zeitablauf, gefordert werden:

Anforderung 4 ***InferenzSynchron: Terminierung****: Die InferenzSynchron terminiert in jedem Fall, ggf. durch Zeitablauf (Timer). Wird die InferenzSynchron durch Zeitablauf terminiert, dann werden alle Replikate den synchron zu aktualisierenden Replikaten zugeordnet.*

In Anforderung 4 kommt zum Ausdruck, dass bei Abbruch der Inferenz durch Zeitablauf keine möglicherweise ungültige Partitionierung für die Aktualisierung verwendet wird, sondern dass dann die Priorität analog zum Startzustand auf die Konsistenz gelegt wird. Während der Inferenz könnte als weiteres Problem auftreten, dass ein Replikat auf Grund einer Regel X der Menge M_S und auf Grund der Regel Y der Menge M_A zugeordnet werden soll. Wenn durch derartig widersprüchliche Regeln eine Partitionierung nicht möglich wäre, könnte kein Schreibzugriff erfolgen, obwohl die Komponenten mit Replikat möglicherweise verfügbar sind, d.h. durch widersprüchlich spezifizierte Regeln würde die Verfügbarkeit sinken. Nachfolgende Anforderung soll eine derartige Blockierung vermeiden, wobei im Zweifelsfall wiederum Priorität auf Konsistenz gelegt wird:

Anforderung 5 ***InferenzSynchron: Widersprüchliche Regeln****: Wenn bei der synchronen Aktualisierung ein Replikat durch widersprüchliche Regeln sowohl synchron als auch asynchron zu aktualisieren ist, dann wird ein derartiger Widerspruch entweder durch Regeln gelöst oder das Replikat wird synchron aktualisiert.*

Die Anforderung 5 beinhaltet, dass es Regeln zur Auflösung undefinierter Zustände gibt. Hierauf wird in Abschnitt 4.3.2 detailliert eingegangen. An dieser Stelle ist zunächst wichtig, dass grundsätzlich immer eine Partitionierung möglich ist.

Bei der InferenzAsynchron geht es darum, ob bei der asynchronen Aktualisierung derzeit ein Schreibzugriff erlaubt ist oder nicht. Derzeitiges Aktualisieren würde z.B. entgegenstehen, dass nur in bestimmten Perioden aktualisiert werden soll oder dass für ein Replikat eine tolerierbare Abweichung spezifiziert wurde und die Grenze noch nicht erreicht ist. Insbesondere in dem Fall, dass eine Grenze für zulässige Abweichungen festgelegt wurde, kann unterschieden werden, ob bei Überschreitung der Grenze alle ausgelassenen Aktualisierungen nachgeholt werden oder ob nur so viele ausgelassene Aktualisierungen nachgeholt werden, die nötig sind, um wieder innerhalb der Grenze zu liegen. Daher liefert die InferenzAsynchron (siehe Listing 4.2 auf Seite 74) drei mögliche Rückgabewerte:

1. `KeinZugriff`: Derzeit ist kein Schreibzugriff auf das Replikat R erlaubt. Damit sind auch folgende Schreibzugriffe aus der Replica Queue auf R untersagt, weil ansonsten die chronologische Verarbeitung verletzt würde. Eine Aktualisierung erfolgt ggf. später, wenn die asynchrone Aktualisierung erneut gestartet wird.
2. `EinAuftrag`: Ein Schreibzugriff auf das Replikat R ist derzeit erlaubt, aber es darf nur ein Auftrag aus der Replica Queue bearbeitet werden. Falls weitere Aufträge für R in der Replica Queue anstehen, muss erneut die InferenzAsynchron durchgeführt werden.
3. `AlleAuftraege`: Alle Schreibzugriffe auf das Replikat R sind derzeit erlaubt, d.h. alle Aufträge aus der Replica Queue, die R schreiben wollen, können ausgeführt werden.

Analog zu den Anforderungen der InferenzSynchron werden hier die Anforderungen für den Startzustand und den Endzustand formuliert, wobei auch hier Priorität auf Konsistenz gelegt wird, d.h. es wird eine zeitnahe Aktualisierung dadurch gefordert, dass derzeit alle Aufträge ausgeführt werden können:

Anforderung 6 ***InferenzAsynchron: Startzustand****: Zu Beginn der InferenzAsynchron wird die Aktualisierungsart* `AlleAuftraege` *gesetzt.*

Anforderung 7 ***InferenzAsynchron: Endzustand****: Am Ende der InferenzAsynchron liegt eine gültige Entscheidung vor. Es wird entweder der Rückgabewert* `KeinZugriff`*,* `EinAuftrag` *oder* `AlleAuftraege` *geliefert.*

Damit auch bei der InferenzAsynchron keine undefinierten Zustände verbleiben, werden Anforderungen aufgestellt, die die Reaktion auf Fehler und die Terminierung spezifizieren. Hierbei wird wiederum im Sinne der Konsistenz eine möglichst frühe Aktualisierung gefordert, d.h. ein Schreibzugriff auf das Replikat ist derzeit erlaubt. Ob die aufrufende Komponente das Ergebnis der InferenzAsynchron verwendet, obliegt analog zur InferenzSynchron der Komponente selbst. So muss auch hier die aufrufende Komponente prüfen, ob die chronologische Verarbeitung verletzt ist (siehe Listing 4.2 auf Seite 74):

Anforderung 8 ***InferenzAsynchron: Fehler****: Wenn ein Fehler während der InferenzAsynchron auftritt, dann wird die Aktualisierungsart* `AlleAuftraege` *gesetzt.*

Anforderung 9 ***InferenzAsynchron: Terminierung****: Die InferenzAsynchron terminiert in jedem Fall, ggf. durch Zeitablauf (Timer). Wenn die InferenzAsynchron durch Zeitablauf terminiert wird, dann wird die Aktualisierungsart* `AlleAuftraege` *gesetzt.*

Auch bei der asynchronen Aktualisierung könnten widersprüchlich spezifizierte Regeln auftreten. So könnte durch Regel X bestimmt sein, dass das Replikat derzeit aktualisiert werden darf, während Regel Y eine Aktualisierung derzeit nicht gestattet. Ein derartiger Widerspruch könnte ebenfalls durch Regeln gelöst werden. Standardmäßig soll der zeitnahen Aktualisierung der Vorrang gegeben werden, was durch folgende Anforderung zum Ausdruck kommt:

Anforderung 10 ***InferenzAsynchron: Widersprüchliche Regeln****: Wenn bei der asynchronen Aktualisierung für ein Replikat durch widersprüchliche Regeln unterschiedliche Aktualisierungsarten gesetzt werden, dann wird ein derartiger Widerspruch entweder durch Regeln gelöst oder die Aktualisierungsart* `AlleAuftraege` *wird gesetzt.*

Bei der InferenzLesen wird eine Menge von passenden Replikaten ermittelt, d.h. es werden Replikate bestimmt, die die geforderten Eigenschaften für den Lesezugriff erfüllen. Die Menge kann leer sein, d.h. es existiert kein Replikat, dass den geforderten Eigenschaften genügt. Welches Replikat aus den passenden Replikaten für den Lesezugriff ausgewählt wird, ist beliebig. Zu Beginn der Inferenz, bei Fehlern während der Inferenz oder bei Zeitabbruch muss davon ausgegangen werden, dass kein passendes Replikat existiert, d.h. die Menge der passenden Replikate ist leer. Dadurch sind folgende Anforderungen an die Inferenz bei Lesezugriffen motiviert:

Anforderung 11 ***InferenzLesen: Startzustand****: Zu Beginn der InferenzLesen wird gesetzt, dass die Menge der passenden Replikate leer ist.*

Anforderung 12 ***InferenzLesen: Endzustand****: Am Ende der InferenzLesen liegt ein gültiges Ergebnis vor, d.h. die Menge der passenden Replikate ist gültig bestimmt, ggf. ist die Menge leer.*

Anforderung 13 ***InferenzLesen: Fehler****: Wenn ein Fehler während der InferenzLesen auftritt, dann ist die Menge der passenden Replikate leer.*

Anforderung 14 ***InferenzLesen: Terminierung****: Wenn die InferenzLesen durch Zeitablauf terminiert wird, dann ist die Menge der passenden Replikate leer.*

Während bei der InferenzSynchron bzw. InferenzAsynchron durch widersprüchliche Regeln ein und dasselbe Fakt unterschiedlich gesetzt werden könnte, können Widersprüche bei Lesezugriffen in der Form auftreten, dass z.B. zwei Regeln formuliert sind, die unterschiedliche Konsistenzeigenschaften beinhalten und damit unterschiedlich geeignete Replikate referenzieren. Auch diese Widersprüche sollen ggf. durch Regeln behandelt werden:

Anforderung 15 ***InferenzLesen: Widersprüchliche Regeln****: Wenn bei einem Lesezugriff durch widersprüchliche Regeln unterschiedlich qualifizierte Replikate ermittelt werden, dann wird ein derartiger Widerspruch entweder durch Regeln gelöst oder es wird kein passendes Replikat geliefert.*

Abschließend sei angemerkt, dass durch diese Anforderungen an die Inferenz die Protokolle für Schreib- und Lesezugriffe neben den vorgestellten Algorithmen spezifiziert sind und damit die Koordination der Zugriffe bei der Replikationsstrategie RegRess festgelegt ist.

4.2.5. Korrektheitskriterium Letztendliche Konsistenz

Ein wichtiger Aspekt bei der Beurteilung einer Replikationsstrategie ist die Korrektheit bzw. das Korrektheitskriterium, die bzw. das der Replikationsstrategie zugrunde liegt. In Abschnitt 2.3.1 wurden die Begriffe Korrektheit und Konsistenz erläutert, die in dieser Arbeit synonym verwendet werden. Bevor in Abschnitt 4.3 die von RegRess gewährleistete Konsistenz detailliert

quantifiziert wird, folgt in diesem Abschnitt eine Diskussion des Korrektheitskriteriums „Letztendliche Konsistenz“ (siehe Definition 19 auf Seite 34), das eine Möglichkeit ist, um im Sinne der Definition 10 auf Seite 31 von eine Replikationsstrategie zu sprechen.

Letztendliche Konsistenz bedeutet, dass alle Replikate letztendlich alle Schreiboperationen empfangen und je zwei Replikate, die die gleiche Menge an Schreiboperationen empfangen haben, einen identischen Wert repräsentieren. Bei RegRess wird durch die asynchrone Aktualisierung dafür gesorgt, dass alle Replikate die Schreiboperationen in gleicher Reihenfolge empfangen. Damit die Replikate dann auch den gleichen Wert repräsentieren, müssen konfligierende Zugriffe geeignet vermieden oder behandelt werden.

Konflikte liegen dann vor, wenn erstens ein und dasselbe physische Datenobjekt, hier genau ein Replikat, von konkurrierenden Zugriffen betroffen ist oder zweitens unterschiedliche Replikate ein und desselben logischen Datenobjekts betroffen sind. Im ersten Fall wird in dieser Arbeit von Schreib-/Lesekonflikten bzw. Schreib-/Schreibkonflikten gesprochen (siehe Definition 14 auf Seite 32), im zweiten Fall von Schreib-/Lesekonflikten$_R$ bzw. Schreib-/Schreibkonflikten$_R$ (siehe Definition 15 und 16 auf Seite 33), wobei hier zur Unterscheidung der Index R gesetzt ist. Bei der folgenden Beschreibung wird auf die Probleme fokussiert, die durch die Replikation entstehen, d.h. es wird vorausgesetzt, dass ein Transaktionssystem oder ein Transaktionsmanager, ggf. in Zusammenarbeit mit den lokalen Transaktionsmanagern der Komponenten mit Replikat, die Synchronisation von Zugriffen übernimmt und insbesondere Schreib-/Schreibkonflikte vermeidet. Hierdurch kann es selbstredend zu Abbrüchen von Transaktionen kommen.

Andererseits werden bei RegRess weniger Transaktionen abgebrochen, als wenn das Korrektheitskriterium 1-Kopien-Serialisierbarkeit eingehalten werden muss. Bei RegRess wird mit abgeschwächter Serialisierbarkeit im Sinne der Epsilon-Serialisierbarkeit gearbeitet (siehe Abschnitt 2.2.4), d.h. Schreibtransaktionen werden untereinander synchronisiert und reine Lesetransaktionen werden nicht synchronisiert mit der Folge, dass das Lesen veralteter Daten erlaubt ist. Wie veraltet die Daten sein dürfen, wird durch Regeln spezifiziert, auf die in Abschnitt 4.3 detailliert eingegangen wird. Weiterhin muss gewährleistet werden, dass die Replikate ein und des selben logischen Datenobjekts zumindest gegen den gleichen Wert konvergieren, um das Korrektheitskriterium „Letztendliche Konsistenz“ zu erfüllen, das von RegRess gewährleistet werden soll. Somit sollen folgende Punkte detailliert betrachtet werden:

- Schreib-/Lesekonflikte$_R$ werden toleriert.
- Schreib-/Schreibkonflikte$_R$ werden vermieden.
- Letztendliche Konsistenz als Korrektheitskriterium wird durch chronologische Verarbeitung der Schreibzugriffe gesichert.

Durch die asynchrone Aktualisierung und den Lesezugriff auf ein beliebiges Replikat kann es zu Schreib-/Lesekonflikten$_R$ kommen. Diese Konflikte werden toleriert, wobei in Abhängigkeit der spezifizierten Regeln Aussagen zur Konsistenz möglich sind (siehe Abschnitt 4.3.3). Falls aus Sicht der Anwendung ein Lesezugriff auf ein Replikat mit speziellen Eigenschaften nötig ist, kann der Zugriff von RegRess koordiniert werden (siehe Abschnitt 4.2.3), um z.B. ein aktuelles Replikat zu lesen. Eine Vermeidung von Schreib-/Lesekonflikten auf ein und dasselbe Replikat wird vom lokalen Transaktionssystem der Komponente mit Replikat synchronisiert.

In [NH02] wird Konfliktmanagement als Zusammensetzung von Konfliktvermeidung und Konfliktbehandlung, der eine Konflikterkennung vorhergeht, beschrieben. Im Allgemeinen ist die Zusammensetzung exklusiv, d.h. entweder werden alle Konflikte durch Abbruch konfligierender Transaktionen vermieden oder Konflikte werden grundsätzlich zugelassen. Im letzten Fall müssen die Konflikte erkannt und behandelt werden. Da analog zur Epsilon-Serialisierbarkeit Schreibtransaktionen synchronisiert werden sollen, werden Schreib-/Schreibkonflikte$_R$ vermieden. Schreib-/Schreibkonflikte$_R$ treten auf, wenn z.B. ein Schreibzugriff auf ein logisches Datenobjekt O die Menge M_S^1 von Replikaten synchron aktualisieren will, eine zweiter Schreibzugriff auf dasselbe logische Datenobjekt O die Menge M_S^2 von Replikaten synchron aktualisieren will und M_S^1 und M_S^2 disjunkt sind (siehe Abschnitt 4.2.2). Angemerkt sei, dass der Konflikt durch

das Transaktionssystem vermieden würde, wenn M_S^1 und M_S^2 nicht disjunkt sind, genauso wie allgemeine Schreib-/Schreibkonflikte.

Eine Vermeidung von Schreib-/Schreibkonflikten$_R$ wird nun dadurch erreicht, dass keine zwei Schreibzugriffe „gleichzeitig“ auf ein und dasselbe logische Datenobjekt durchgeführt werden. Daher ist im Ablauf der synchronen Aktualisierung (siehe Listing 4.1 auf Seite 71) eine Sperre vor den Schreibzugriffen auf die Replikate für das jeweilige logische Datenobjekt zu beantragen, d.h. es muss eine Verwaltung für Sperren auf logische Datenobjekte ermöglicht werden. Der Nachteil eines solchen Sperrmechanismus wird am Ende dieses Abschnitts diskutiert. Weil über das Transaktionssystem allgemeine Schreib-/Schreibkonflikte, die z.B. dann auftreten, wenn durch die synchrone und die asynchrone Aktualisierung auf ein und dasselbe Replikat zugegriffen wird, und durch den Sperrmechanismus Schreib-/Schreibkonflikte$_R$ vermieden werden, erfolgt eine korrekte Synchronisation von Schreibzugriffen. Wenn diese Schreibzugriffe in eine Schreibtransaktion gemäß der Epsilon-Serialisierbarkeit eingebettet sind, dann werden diese Schreibtransaktionen dadurch synchronisiert, dass ein Transaktionsmanager die einzelnen Schreibzugriffe als Teiltransaktion innerhalb einer geschlossen-geschachtelten Transaktion behandelt (siehe Anmerkungen in Abschnitt 4.2.2).

Um das Korrektheitskriterium *„Letztendliche Konsistenz“* zu gewährleistet, muss dafür Sorge getragen werden, dass jedes Replikat jeden seiner Aktualisierungszugriffe erhält und gegen den aktuellen Wert des logischen Datenobjekts konvergiert. Bei RegRess werden alle asynchronen Aktualisierungsaufträge in der Replica Queue (siehe Abschnitt 4.2.2) gespeichert. Es wird idealisierend angenommen, dass ein Replikationsmanager, der RegRess implementiert, nicht dauerhaft ausfällt und kein Auftrag dauerhaft blockiert wird, d.h. es wird davon ausgegangen, dass jedes Replikat „seine“ Schreibzugriffe erhält. Weiterhin wird trivialerweise vorausgesetzt, dass Replikate, die die gleichen Schreibzugriffe in der gleichen Reihenfolge erhalten, auch die gleichen Werte speichern. Diese Forderung wird durch die Annahme erfüllt, dass alle lokalen Datenbanken bzw. Komponenten mit Replikat die identischen Konsistenzbedingungen lokal definiert haben (siehe Abschnitt 4.1.1), d.h. eine Schreiboperation ist grundsätzlich entweder bei allen Replikaten eines logischen Datenobjekts durchführbar oder bei keinem Replikat.

Zur Einhaltung der letztendlichen Konsistenz muss RegRess somit gewährleisten, dass Schreibzugriffe in der gleichen Reihenfolge, also chronologisch, erfolgen (Sonderfälle wie Kommutativität werden nicht gesondert betrachtet). Dazu müssen in der Replica Queue die Aufträge zumindest für jedes Replikat chronologisch gespeichert werden. Da bei einem Schreibzugriff auf ein logisches Datenobjekt eine Sperre beantragt wird und diese Sperre erst nach dem Anhängen an die Replica Queue freigegeben wird, kann kein folgender Schreibzugriff auf dasselbe logische Datenobjekt einen Vorgänger „überholen“. Also werden die asynchronen Aufträge in chronologischer Reihenfolge gespeichert. Solange asynchrone Aufträge für ein Replikat R in der Replica Queue anstehen, dürfen auch keine synchronen Zugriffe auf R durchgeführt werden. Dies wird dadurch verhindert, dass nach der Inferenz die Verletzung der chronologischen Verarbeitung geprüft wird (siehe Zeile 9 des Listings 4.1 auf Seite 71). Abschließend muss gesichert sein, dass kein Auftrag A für Replikat R aus der Replica Queue ausgeführt wird, wenn ein anderer Auftrag für R chronologisch vor A zu bearbeiten ist. Um unnötige Blockierungen zu vermeiden, können zwar aus der Replica Queue Aufträge indiziert abgerufen und bearbeitet werden, so dass eine Chronologie über alle Aufträge nicht eingehalten wird, aber für ein und dasselbe Replikat wird bei der asynchronen Aktualisierung die Chronologie gewährleistet (siehe Zeile 8 des Listings 4.2 auf Seite 74). Somit erfüllt RegRess zumindest die letztendliche Konsistenz.

Wie bereits erwähnt, können die zeitversetzten Änderungen auf Grund identischer Konsistenzbedingungen der lokalen Datenbanken grundsätzlich durchgeführt werden. Daher müssen keine Änderungen, die vorher an anderen Replikaten durchgeführt wurden, auf Grund von gescheiterten Änderungsoperationen bei anderen Replikaten des gleichen logischen Datenobjekts zurückgenommen werden, d.h. es werden nachträglich keine Transaktionen zurückgesetzt. Somit erfüllt RegRess die gegenüber der letztendlichen Konsistenz etwas schärfere Forderung der Kon-

vergenz [Len97] (vergleiche die Anmerkungen nach Definition 19 auf Seite 34 zur letztendlichen Konsistenz).

Der Nachteil der Vermeidung von Schreib-/Schreibkonflikten$_R$ über einen Sperrmechanismus und der chronologischen Verarbeitung von Schreibzugriffen je Replikat durch Verwendung einer Replica Queue ist die Abhängigkeit von diesen „Kontrolldaten“, da sowohl die Sperren als auch die Replica Queue im gesamten System eindeutig sein müssen. Daher müssen diese Kontrolldaten entweder zentral an einer Stelle verwaltet werden oder bei Verteilung konsistent sein. Im zentralen Fall ist die Replikation von dieser Stelle abhängig, d.h. die Verfügbarkeit steht und fällt mit dieser zentralen Stelle. Sofern auch der Replikationsmanager zentral angelegt ist (siehe Abschnitt 4.1.2), ist diese Abhängigkeit möglicherweise zu akzeptieren. Im verteilten Fall müssen die Kontrolldaten konsistent repliziert werden, d.h. es muss eine synchrone Replikationsstrategie gewählt werden, z.B. ein Votierungsverfahren (siehe Abschnitt 3.3.1). Eine andere Möglichkeit, die letztendliche Konsistenz bei Verzicht auf einen Sperrmechanismus und einer chronologischen Verarbeitung zu erreichen, wäre es, Konflikte zuzulassen, und diese anschließend zu erkennen und zu behandeln. Diese Möglichkeit wird im Ausblick diskutiert (siehe Kapitel 9).

4.3. Konsistenz von RegRess

Im vorangehenden Abschnitt 4.2 wurde erläutert, wie die in dieser Arbeit vorgestellte Replikationsstrategie RegRess die Zugriffe auf die Replikate koordiniert, wobei auf das Protokoll fokussiert wurde. Dabei wurde sowohl der Ablauf für Schreibzugriffe als auch für Lesezugriffe spezifiziert. Durch die Spezifikation des von RegRess verwendeten Protokolls bei Zugriffen wurde bereits sichergestellt, dass RegRess das Korrektheitskriterium *„Letztendliche Konsistenz“* gewährleistet (siehe Abschnitt 4.2.5). Es sei nochmals angemerkt, dass in dieser Arbeit die Begriffe Konsistenz und Korrektheit synonym verwendet werden.

Mit letztendlicher Konsistenz ist gemeint, dass die Replikate desselben logischen Datenobjekts den gleichen Wert repräsentieren, wenn alle Änderungen des logischen Datenobjekts zu den jeweiligen Replikaten propagiert sind. Dabei muss natürlich eine Änderung, die bei einem Replikat durchgeführt werden kann, auch an jedem anderen Replikat desselben logischen Datenobjekts grundsätzlich durchgeführt werden können. Durch die Annahme, dass alle lokalen Datenbanken bzw. Komponenten mit Replikat die identischen Konsistenzbedingungen lokal definiert haben (siehe Abschnitt 4.1.1), wird diese Forderung erfüllt. Des Weiteren wird durch die Erfüllung der lokalen Konsistenzbedingungen gewährleistet, dass die lokale Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand überführt wird (siehe Abschnitt 2.3.1), d.h. in diesem Abschnitt wird nicht die „allgemeine Konsistenz“ diskutiert, sondern die Konsistenz der Replikate desselben logischen Datenobjekts.

Zwar wird durch die letztendliche Konsistenz gewährleistet, dass alle Replikate eines logischen Datenobjekts „irgendwann“ den selben Wert repräsentieren, aber es sind keine Aussagen über das Alter der Replikate möglich, d.h. bei Lesezugriffen kann nicht gesagt werden, ob das Replikat aktuell ist bzw. wie veraltet ein Replikat ist. In Abschnitt 4.2.2 wurde das Protokoll für Schreibzugriffe der hier vorgestellten Replikationsstrategie RegRess spezifiziert, wobei der Wechsel zwischen synchroner und asynchroner Aktualisierung erläutert wurde, ohne dabei konkrete Regeln zu nennen. In diesem Abschnitt werden nun mögliche Regeln genannt, die RegRess einem Administrator zur Verfügung stellt, genauer gesagt, es werden Anforderungen aufgelistet, die RegRess bzw. die zugehörige Regelsprache (siehe Kapitel 5) erfüllen soll. Da durch die Regeln, ggf. zusammen mit den Zuständen der beteiligten Komponenten, festgelegt wird, wann und wie Replikate aktualisiert werden, ist hiermit eine genauere Bestimmung des Alters der Replikate möglich. Durch die spezifizierten Regeln wird also die gewährleistete Konsistenz von RegRess festgelegt, die in dieser Arbeit durch einen so genannten *„Konsistenzgrad“* quantifiziert wird. Beim Konsistenzgrad handelt es sich um einen probabilistischen Wert, durch den die Wahrscheinlichkeit konsistenter Zugriffe definiert wird.

Bereits in Abbildung 4.3 auf Seite 66 wurde festgelegt, dass eine Replikationseinheit mehrere Replikationsregeln besitzen kann, die jeweils für Komponenten mit Replikat gelten. Dabei wurden Regeln grob in fachlich orientierte und in technisch orientierte Regeln aufgeteilt. Diese Aufteilung wird hier aufgegriffen, indem zunächst in Abschnitt 4.3.1 fachliche Konsistenzbedingungen in Form von Anforderungen an ein regelbasiertes System und dann in Abschnitt 4.3.2 technische Konsistenzbedingungen konkretisiert werden. Durch die Spezifikation der Anforderungen an das regelbasierte System wird festgelegt, welche Regeln von einem Administrator als Konfiguration potentiell festgelegt werden können. Die konkreten Regeln bilden die Basis der von RegRess gewährleisteten Konsistenz, die in Abschnitt 4.3.3 beschrieben wird.

4.3.1. Fachliche Konsistenzbedingungen

In diesem Abschnitt werden fachliche Konsistenzbedingungen in Form von Anforderungen, die vom regelbasierten System der Replikationsstrategie RegRess zur Verfügung gestellt werden, diskutiert. Hieraus resultieren fachlich orientierte Regeln, d.h. Regeln, die eine datenzentrierte Spezifikation der Konsistenz in Abhängigkeit der Domäne erlauben. So kann z.B. anwendungsbezogenes Wissen, das besagt, dass bestimmte Replikate nicht immer den aktuellen Wert repräsentieren müssen, in der Form genutzt werden, dass für diese Replikate eine zeitversetzte Aktualisierung ausreicht, also z.B. eine asynchrone Aktualisierung in bestimmten Perioden. Demgegenüber stehen die technischen Konsistenzbedingungen (siehe Abschnitt 4.3.2), d.h. technisch orientierte Regeln, die die Reaktion auf Veränderungen der zugrunde liegenden Systemlandschaft beschreiben, wie z.B. ein Ausfall einer Komponente mit Replikat (Rechnerausfall). Diese Unterteilung kann mit der Aufteilung von Anforderungen in funktionale und nicht-funktionale Anforderungen verglichen werden.

Bevor konkrete Anforderungen für fachliche Konsistenzbedingungen aufgestellt werden, folgen zunächst Anforderungen, die für alle Regeln relevant sind. In Abbildung 4.3 auf Seite 66 wurde die Struktur von Regeln als UML-Klassendiagramm illustriert. Diese Struktur soll vollständigkeitshalber als Anforderung spezifiziert werden, wobei bedacht werden muss, dass Replikationseinheiten identifiziert werden können (siehe Definition 35 auf Seite 65):

Anforderung 16 ***Struktur von Regeln**: Regeln werden Replikationseinheiten zugeordnet und betreffen genau ein „Zielsystem“, d.h. eine Replikationseinheit besitzt mehrere Regeln, die jeweils für eine Komponente mit Replikat gelten. Bei einem Schreib- oder Lesezugriff auf ein logisches Datenobjekt können ggf. eine oder mehrere Replikationseinheiten als Initiator dieses Schreib- oder Lesezugriffs identifiziert werden.*

Bei der Konfiguration von RegRess werden somit Replikationseinheiten definiert, für die Regeln spezifiziert werden. Die Replikationseinheiten müssen bei Schreib- bzw. Lesezugriffen auf logische Datenobjekte identifizierbar sein, entweder, indem sie als Parameter mitgegeben werden, oder, indem sie von der zugreifenden Komponente (Client) oder von dem zuzugreifenden Objekt bestimmt werden. Wenn keine Regeln für einen Zugriff vorliegen, z.B. weil keine entsprechende Replikationseinheit vorliegt, dann gelten die Startzustände der jeweiligen Inferenzart (siehe Anforderungen 1, 6 und 11 in Abschnitt 4.2.4).

Grundsätzlich ist es möglich, dass für einen Zugriff mehr als eine Replikationseinheit identifiziert werden kann, z.B. weil für die zugreifende Komponente und für eine Menge von Objekten, zu denen das zuzugreifende Objekt gehört, Regeln spezifiziert wurden. In derartigen Fällen werden alle Regeln der identifizierten Replikationseinheiten bei der Inferenz herangezogen, was durch nachfolgende Anforderung zum Ausdruck gebracht wird:

Anforderung 17 ***Aktive Regeln bei einer Inferenz**: Bei der Inferenz werden aktive Regeln berücksichtigt. Die aktiven Regeln sind diejenigen, die zu den für diesen Zugriff identifizierten Replikationseinheiten gehören.*

Zwar kann eine Regel aktiv sein, weil sie zu der Replikationseinheit gehört, für die ein Zugriff durchgeführt wird, aber trotzdem unberücksichtigt bleiben, weil sie derzeit nicht gültig ist. Bereits im UML-Klassendiagramm (siehe Abbildung 4.3 auf Seite 66) wurden Gültigkeitszeiten für Regeln attributiert, die in der folgenden Anforderung berücksichtigt werden:

Anforderung 18 ***Gültigkeitszeitraum von Regeln***: *Für jede Regel kann ein Zeitraum definiert werden, in dem die Regel gültig ist, d.h. in dem die Regel bei der Inferenz ausgewertet wird. Der Zeitraum kann sich periodisch wiederholen, z.B. täglich von 10:00 Uhr bis 15:00 Uhr. Wenn kein Zeitraum angegeben ist, dann ist die Regel immer gültig.*

In Abbildung 4.4 auf Seite 69 wurde die Partitionierung der betroffenen Replikate in synchron und asynchron zu aktualisierende Replikate gezeigt. Diese Partitionierung beruht darauf, dass Replikate entweder grundsätzlich synchron oder grundsätzlich asynchron aktualisiert werden oder wechselfähig in ihrer Aktualisierungsart sind. Daher können diese Zustände je Replikationseinheit den Replikaten über Regeln zugeordnet werden:

Anforderung 19 ***Zustand eines Replikats***: *Für die Replikate einer Replikationseinheit kann ein Zustand gesetzt werden, der angibt, wie die Replikate zu aktualisieren sind bzw. ob sie wechselfähig sind. Somit sind mögliche Zustände:* `synchron`, `asynchron` *oder* `wechselfähig`. *Diese Zustände können je Replikationseinheit unterschiedlich festgelegt werden. Für eine Replikationseinheit können mehrere Zustandsregeln spezifiziert werden, z.B. für unterschiedliche Zeiträume werden unterschiedliche Aktualisierungsarten festgelegt. Bei Widersprüchen wird der Zustand* `synchron` *gesetzt bzw. bei Widerspruch zwischen* `wechselfähig` *und* `asynchron` *wird der Zustand* `wechselfähig` *gesetzt.*

Die Anforderung 19 spielt nur bei der InferenzSynchron (siehe Abschnitt 4.2.4) eine Rolle, also bei der Partitionierung in synchron und asynchron zu aktualisierende Replikate. Da hier die Voreinstellung die synchrone Aktualisierung ist, wird bei wechselfähigen Replikaten zunächst die synchrone Aktualisierung angenommen. Erst dann, wenn weitere Regeln einen Wechsel erlauben, wie z.B. eine periodische Aktualisierung (fachliche Konsistenzbedingung) oder ein Ausfall einer Komponente mit Replikat (technische Konsistenzbedingung), wird das entsprechende Replikat asynchron aktualisiert. Angemerkt sei, dass es bei Problemen beim Zugriff auf synchron zu aktualisierende Replikate zum Abbruch des Zugriffs kommt, während bei Problemen bei der asynchronen Aktualisierung der Zugriff zu einem späteren Zeitpunkt wiederholt werden kann.

Fachliches Wissen des Anwendungsbereichs erlaubt unter Umständen, dass Replikate veralten dürfen. Häufig ist dabei interessant, in irgendeiner Form Grenzen für die Veralterung festzulegen. Hierfür wurden in Abschnitt 3.2.3 verschiedene abgeschwächte Konsistenzbegriffe aus der entsprechenden Literatur vorgestellt, die an dieser Stelle aufgegriffen werden sollen. Um solche Grenzen der Veralterung einzuhalten, wäre es nach [ABGMA88] nötig, „gleichzeitig“ zu der Transaktion, die die Überschreitung der Grenze bewirkt, die ausgelassenen Änderungen zumindest in der Form nachzuholen, dass die Grenze wieder eingehalten wird. Zwar ist dieses „gleichzeitige“ Nachholen von ausgelassenen Änderungen im Rahmen einer verteilten ACID-Transaktion (siehe Abschnitt 2.2.1) möglich, bedeutet aber erhöhten Aufwand und insbesondere die Gefahr von Transaktionsabbrüchen.

Wenn das „gleichzeitige“ Nachholen von ausgelassenen Änderungen bei Überschreitung der Grenze nicht angewendet wird, dann müssen die entsprechenden Replikate ungültig gesetzt werden, damit die spezifizierte Konsistenz gewährleistet bleibt. Weil z.B. bei dem Abstandsmaß *„Wertdifferenz“* eine lokale Komponente mit veraltetem Replikat nicht alleine entscheiden kann, ob die Grenze erreicht bzw. überschritten ist, kann das entsprechende Replikat nicht lokal ungültig gesetzt werden, ohne sich mit einem aktuellen Replikat abzustimmen. In [ABGMA88] wird daher zusätzlich eine Kommunikation von so genannten *„Nullnachrichten“* eingeführt, mit der die Erreichbarkeit von Komponenten überprüft wird. Bei Überschreiten einer bestimmten

Verzögerung (delay) wird von einem Ausfall ausgegangen, der zusätzlich bei einem Zugriff auf das veraltete Replikat angezeigt wird, sodass der Client dann über die Gültigkeit der gelesenen Daten selbst entscheiden kann.

Bei der aktuellen Version von RegRess soll auf jeglichen Aufwand verzichtet werden, das Überschreiten von Grenzen für die Veralterung der Replikate zu verhindern oder kenntlich zu machen. Entsprechende Erweiterungen werden im Ausblick diskutiert (siehe Kapitel 9). Die nachfolgenden Anforderungen, mit denen die Veralterung von Replikaten ermöglicht wird und eine Grenze für das Alter festgelegt werden kann, dienen der asynchronen Aktualisierung zur Bestimmung, ob die ausgelassenen Änderungen eines Replikats derzeit nachgeholt werden sollen oder nicht, d.h. die InferenzAsynchron liefert erst bei Überschreitung der Grenze, dass der Schreibzugriff durchzuführen ist. Wenn nun die Aktualisierung des veralteten Replikats scheitert und anschließend Lesezugriffe auf das Replikat folgen, dann liegt eine Verletzung der Altersgrenze vor. Wie bereits erwähnt, soll diese Verletzung toleriert werden, weil ansonsten die entsprechenden Replikate ungültig gesetzt werden müssten.

In [ABGMA88] wurden so genannte Kohärenzbedingungen definiert, mit denen der zulässige Abstand eines Replikats gegenüber einem aktuellen Replikat spezifiziert werden kann (siehe Definition 22 auf Seite 37). Dabei wird zwischen einer Zeitdifferenz, einem Versionsabstand und einer Wertdifferenz unterschieden. Derartige Kohärenzbedingungen sollen auch bei der hier vorgestellten Replikationsstrategie RegRess berücksichtigt werden:

Anforderung 20 ***Kohärenzbedingungen***: *In Form eines zeitlichen Abstands, eines Versionsabstands und/oder einer Wertdifferenz können Kohärenzbedingungen spezifiziert werden, mit der die Veralterung von Replikaten gegenüber einem aktuellen Replikat des selben logischen Objekts festgelegt werden kann. Liegen für ein wechselfähiges Replikat Kohärenzbedingungen vor, wird bei der InferenzSynchron das Replikat den asynchron zu aktualisierenden Replikaten zugeordnet. Wenn eine Kohärenzbedingung bei der InferenzAsynchron verletzt ist, d.h. wenn eine zulässige Grenze für das Alter überschritten ist, dann werden ausgelassene Aktualisierungen nachgeholt.*

Bei Snapshots [AL80] wird kein Abstandsmaß spezifiziert, sondern es werden Aktualisierungszeitpunkte festgelegt (siehe Definition 23 auf Seite 37), wobei die zeitversetzte Aktualisierung entweder periodisch oder ereignisgesteuert erfolgt. Eine ereignisorientierte Aktualisierung wird an dieser Stelle nicht betrachtet, weil hierfür keine Inferenz nötig ist, sondern eine entsprechende Funktionalität des Replikationsmanagers. Somit verbleibt die Forderung nach einer periodischen Aktualisierung:

Anforderung 21 ***Periodische Aktualisierung***: *Für das Nachholen von ausgelassenen Aktualisierungen können periodische Zeitpunkte festgelegt werden. Wechselfähige Replikate, für die eine periodische Aktualisierung spezifiziert ist, werden bei der InferenzSynchron den asynchron zu aktualisierenden Replikaten zugeordnet.*

Wenn Replikate, für die Kohärenzbedingungen oder eine periodische Aktualisierung festgelegt ist, zwingend synchron zu aktualisieren sind (siehe Anforderung 19 auf Seite 86), dann liegt ein Konflikt vor. Die Reaktion auf Konflikte ist durch Anforderung 5 auf Seite 80 geregelt. Neben diesen abgeschwächten Konsistenzbegriffen werden in Abschnitt 3.2.3 Session-Garantien (siehe Definition 21 auf Seite 36) vorgestellt. Hierauf kann bei RegRess verzichtet werden, weil derartige Garantien durch geeignete Regeln abgebildet werden können, z.B. kann für Lesezugriffe eine bestimmte Konsistenz gefordert werden (siehe unten).

Weiterhin werden in Abschnitt 3.2.3 abgeschwächte Konsistenzbegriffe vorgestellt, die die „Datenfrische“ bzw. die „probabilistische Konsistenz“ betreffen. Während z.B. bei den Kohärenzbedingungen bestimmte Replikate aktualisiert werden müssen, nämlich die veralteten Replikate, die die zulässige Grenze überschritten haben, können bei der „Datenfrische“ oder der „probabilistischen Konsistenz“ beliebige Aktualisierungen nachgeholt werden, um wieder in einen

tolerierbaren Bereich zu gelangen. Beispielsweise wird die Frische einer replizierten Datenbank (siehe Definition 26 auf Seite 38) dadurch verbessert, dass irgendwelche ausgelassene Aktualisierungen nachgeholt werden, obwohl ein bestimmtes Replikat beliebig veralten kann.

Eine Analogie bei RegRess ist die Anzahl an Aufträgen in der Replica Queue. Je mehr Aufträge in der Replica Queue anstehen, d.h. je größer die Anzahl an ausgelassenen Aktualisierungen, desto größer die Inkonsistenz bzw. die Frische der Datenbank. Die Überschreitung einer bestimmten Anzahl an Aufträgen kann zum Anlass für das Nachholen der Aktualisierungen genommen werden. Bei RegRess wird die gewährleistete Konsistenz in Form eines Konsistenzgrades gemessen (siehe Abschnitt 4.3.3), auf den folgende Forderung zielt:

Anforderung 22 ***Konsistenzgradbedingung:*** *Es kann eine Grenze für den Konsistenzgrad festgelegt werden. Wird die Grenze unterschritten, dann werden ausgelassene Aktualisierungen nachgeholt.*

Sowohl bei den Kohärenzbedingungen, bei der periodischen Aktualisierung als auch bei der Konsistenzgradbedingung kann festgelegt werden, ob alle Aufträge oder nur ein Auftrag aus der Replica Queue bearbeitet werden. Die Konsistenzgradbedingung hat jedoch keinen Einfluss auf die InferenzSynchron, weil sie keine Bedeutung für einzelne Replikate hat, sondern für die gesamte Datenbank. Hierüber wird ausschließlich der Zeitpunkt für die asynchrone Aktualisierung gesteuert.

Während mit den oben genannten fachlichen Konsistenzbedingungen die Aktualisierung von Replikaten gesteuert wird, ob bei der synchronen oder asynchronen Aktualisierung, kann bei RegRess auch die Menge der Replikate für einen Lesezugriff auf Grund anwendungsbezogenen Wissens beeinflusst werden. Analog zur Replikationsstrategie FAS (siehe Abschnitt 3.4.2), bei der eine Grenze für die Datenfrische der Replikate für den Lesezugriff vorgegeben werden kann, ist bei RegRess eine Spezifikation der tolerierbaren Konsistenz bzw. Inkonsistenz bei Lesezugriffen möglich. Die tolerierbare Inkonsistenz, die ein Replikat aufweisen darf, bezieht sich dabei auf den Abstand zu einem aktuellen Replikat des selben logischen Objekts, d.h. es handelt sich um Kohärenzbedingungen für Lesezugriffe (vergleiche Anforderung 20), die in folgender Anforderung zum Ausdruck gebracht sind:

Anforderung 23 ***Lesen veralteter Daten:*** *Lesezugriffe auf veraltete Daten werden toleriert. Für das Alter der zu lesenden Daten können Kohärenzbedingungen festgelegt werden, d.h. es kann eine Grenze in Form eines zeitlichen Abstands, eines Versionsabstands und/oder einer Wertdifferenz des zu lesenden Replikats gegenüber einem aktuellen Replikat des selben logischen Objekts festgelegt werden.*

In Abschnitt 4.2.3 ist die Koordination von Lesezugriffen bei RegRess spezifiziert. Dabei wurde diskutiert, dass auch das Lesen eines beliebigen Replikats, insbesondere eines lokalen Replikats, erlaubt ist. In diesem Fall ist beim Lesen keine Einschränkung hinsichtlich des Alters des Replikats möglich. Es kann bestenfalls abgeschätzt werden, wie veraltet das Replikat ist, wofür die festgelegten Bedingungen für Aktualisierungen herangezogen werden müssen. Wenn der Lesezugriff über einen Replikationsmanager erfolgt, der RegRess implementiert, dann können auch Kohärenzbedingungen für Lesezugriffe berücksichtigt werden.

4.3.2. Technische Konsistenzbedingungen

In diesem Abschnitt werden technische Konsistenzbedingungen in Form von Anforderungen festgelegt, die RegRess die Reaktion auf Zustandsänderungen des zugrunde liegenden verteilten Systems erlaubt. Mit den in Abschnitt 4.3.1 vorgestellten fachlichen Konsistenzbedingungen kann Anwendungswissen in der Form berücksichtigt werden, dass im Wesentlichen das Alter der Replikate direkt beeinflusst wird. Die technischen Konsistenzbedingungen zielen z.B. auf

Verfügbarkeit und Performance. Sie beeinflussen das Alter von Replikaten nur indirekt, nämlich dann, wenn auf Grund technischer Umstände eine asynchrone Aktualisierung erlaubt wird.

Aus Sicht eines Replikationsmanagers, der einen Schreib- oder Lesezugriff auf ein Replikat durchführen möchte, ist die Konsequenz eines Rechnerausfalls oder einer Netzpartitionierung gleich, nämlich auf das Replikat kann nicht zugegriffen werden, es ist nicht verfügbar. Bei einem Lesezugriff kann ggf. auf ein anderes Replikat zugegriffen werden und bei der asynchronen Aktualisierung kann der Schreibzugriff später nachgeholt werden. Wenn der Schreibzugriff auf ein Replikat allerdings bei der synchronen Aktualisierung innerhalb der synchronisierten Transaktion fehlschlägt, dann muss im Allgemeinen die Transaktion abgebrochen werden. Bei RegRess kann die Transaktion ggf. wiederholt werden, wobei dann das synchron zu aktualisierende Replikat den asynchron zu aktualisierenden Replikaten zugeordnet wird:

Anforderung 24 ***Verfügbarkeit****: Ein Replikat, das bei einer synchronen Aktualisierung nicht verfügbar ist, d.h. auf dem kein Schreibzugriff durchgeführt werden kann, wird asynchron aktualisiert, wenn es sich um ein wechselfähiges Replikat handelt.*

Die synchrone Aktualisierung erfolgt in einer synchronisierten Transaktion, d.h. die jeweiligen lokalen Teiltransaktionen müssen sich über den Ausgang der Transaktion abstimmen (siehe Abschnitt 2.2.3). Diese Koordination ist mit Aufwand verbunden. Insbesondere richtet sich die Verarbeitungsgeschwindigkeit im Allgemeinen nach dem langsamsten Teilnehmer der verteilten Transaktion. Wenn nun durch den langsamsten Teilnehmer die Transaktion zu stark verzögert wird, kann eine asynchrone Aktualisierung des entsprechenden Replikats sinnvoll sein. Hierdurch ist die folgende Anforderung motiviert:

Anforderung 25 ***Performance bei Schreibzugriffen****: Wenn die Performance einer Komponente mit Replikat einen Schwellwert überschreitet, dann kann bei der synchronen Aktualisierung ein wechselfähiges Replikat dieser Komponente asynchron aktualisiert werden, bzw. dann kann bei der asynchronen Aktualisierung ein Replikat dieser Komponente später aktualisiert werden.*

Die Berücksichtigung der Performance erfolgt auch bei Lesezugriffen. Bei RegRess kann ein Client eine bestimmte Performance für den Lesezugriff fordern, wobei zusätzlich Konsistenzbedingungen angegeben werden können. Damit ist es z.B. möglich, zwar nicht das aktuellste Replikat zu erhalten, aber die Daten schnell zur Verfügung zu haben. Die geforderte Performance wird dabei allerdings nicht zugesichert, sondern es werden Replikate für den Lesezugriff bestimmt, die bei vorhergehenden Lesezugriffen die Kriterien erfüllt haben. Diese Möglichkeit wird durch folgende Anforderung gegeben:

Anforderung 26 ***Performance bei Lesezugriffen****: Bei einem Lesezugriff kann eine „performante“ Komponente mit Replikat für den Lesezugriff auf ein Replikat ausgewählt werden. Ggf. kann zusätzlich eine Konsistenzbedingung berücksichtigt werden.*

Ein anderer Aspekt, der zu unterschiedlicher Reaktion bei Zugriffen führen kann, ist die Datengröße. Wenn z.B. ein logisches Objekt und damit die zugehörigen Replikate eine vorgegebene Datengröße überschreiten, dann kann bei der synchronen Aktualisierung eine andere Partitionierung besser geeignet sein als bei eher kleinen Objekten. Ähnliche Reaktionen sind bei der asynchronen Aktualisierung, bei der z.B. größere Objekte später bearbeitet werden, oder bei Lesezugriffen, bei denen z.B. in Abhängigkeit der Datengröße unterschiedliche Replikate gewählt werden, denkbar. Somit wird nachfolgende Anforderung allgemein formuliert:

Anforderung 27 ***Datengröße****: Bei der Definition von Regeln soll die Dateigröße berücksichtigt werden können.*

Bereits in den ersten Versionen von RegRess (siehe z.B. [NHHT03]) wurde einbezogen, dass eine synchrone Aktualisierung gefordert werden kann, wenn zu viele Schreib-/Schreibkonflikte$_R$ oder Schreib-/Lesekonflikte$_R$ auftreten. Bei der hier vorgestellten Version von RegRess werden Schreib-/Schreibkonflikte$_R$ vermieden (siehe Abschnitt 4.2.2), sodass sich die nachfolgende Anforderung auf Schreib-/Lesekonflikte$_R$ bezieht:

Anforderung 28 ***Konfliktbedingung***: *Wenn die Anzahl der Schreib-/Lesekonflikte$_R$ eine vorgegebene Grenze überschreitet, dann wird ein wechselfähiges Replikat synchron aktualisiert.*

Bei der Anforderung 28 ist zu beachten, dass die Anzahl der Schreib-/Lesekonflikte$_R$ gezählt werden muss. Insbesondere wenn ein Zugriff auf ein lokales Replikat unter Umgehung des Replikationsmanagers erlaubt ist, können Schreib-/Lesekonflikte$_R$ nur festgestellt werden, wenn sich die Komponenten mit Replikat und der Replikationsmanager austauschen. Außerdem darf ein Replikat nur dann synchron aktualisiert werden, wenn kein Auftrag für dieses Replikat in der Replica Queue ansteht, weil ansonsten die Chronologie verletzt würde. In diesem Fall wird die (Teil-)Transaktion abgebrochen (siehe Zeile 9 des Listings 4.1 auf Seite 71).

In Abschnitt 4.2.4 wird die Inferenz bei Schreib- und Lesezugriffen vorgestellt, wobei insbesondere ein eindeutiges Ergebnis der Inferenz verlangt wird. In den Anforderungen 5, 10 und 15 wird die Reaktion auf widersprüchliche Regeln derart festgelegt, dass entweder ein Standardwert geliefert wird oder dass der Widerspruch mittels Regeln gelöst wird. Widersprüchliche Regeln können bei fachlichen und/oder technischen Konsistenzbedingungen auftreten, die durch folgende Anforderung behandelbar werden:

Anforderung 29 ***Regelung von Widersprüchen***: *Widersprüchliche Regeln, d.h. Regeln mit konträren Aktionen, können durch Regeln behandelt werden.*

Auch mit der Anforderung 29 wird die Konsistenz von RegRess beeinflusst, weil z.B. durch eine Widerspruchsregel entweder die stärkere oder schwächere Konsistenzbedingung gewählt wird. Die Behandlung von Widersprüchen wird in Abschnitt 5.3 erläutert. Eine Art der Behandlung von widersprüchlichen Regeln ist die Verwendung von Prioritäten, wodurch folgende Anforderung motiviert ist

Anforderung 30 ***Vergabe von Prioritäten für Regeln***: *Um widersprüchliche Regeln mittels Prisorisierung behandeln zu können, soll die Vergabe von Prioritäten für Regeln in Form von natürlichen Zahlen möglich sein, wobei gilt: Je höher der Wert, desto höher die Priorität. Wenn keine Priorität explizit vergeben wird, dann wird der Prioritätswert null angenommen.*

Eine tabellarische Übersicht über alle Anforderungen an das regelbasierte System bzw. die Regelsprache befindet sich in Anhang B.

4.3.3. Konsistenzgrad von RegRess

Das Wesen einer Replikationsstrategie wird durch das verwendete Korrektheitskriterium bzw., hier synonym, durch die gewährleistete Konsistenz bestimmt. Nach Definition 10 auf Seite 31 koordiniert eine Replikationsstrategie die Zugriffe auf die Replikate unter Verwendung eines bestimmten Korrektheitskriteriums. In Abschnitt 4.2.5 wird erläutert, dass die in dieser Dissertation vorgestellte Replikationsstrategie RegRess das Korrektheitskriterium *„Letztendliche Konsistenz"* erfüllt. Das Lesen veralteter Daten wird toleriert, d.h. Schreib-/Lesekonflikte$_R$ (siehe Definition 15 auf Seite 33) können auftreten, und Schreib-/Schreibkonflikte$_R$ werden durch Sperrmechanismen verhindert. Durch die chronologische Verarbeitung von Änderungen wird somit erreicht, dass jedes Replikat eines logischen Objekts, das alle Änderungen empfangen hat, letztendlich den gleichen Wert speichert.

Die letztendliche Konsistenz wird bei RegRess durch die Protokolle für Schreib- und Lesezugriffe gesichert, die in Abschnitt 4.2 spezifiziert sind. Es werden temporäre Inkonsistenzen erlaubt, die aber nur in so weit quantifizierbar sind, dass sie nach Abschluss aller Propagierungen aufgelöst werden. Genauere Aussagen zu der gewährleisteten Konsistenz, d.h. über das Alter der Replikate, können über die fachlichen und technischen Konsistenzbedingungen erfolgen, die in den Abschnitten 4.3.1 und 4.3.2 festgehalten sind. Wenn bei den Konsistenzbedingungen z.B. ausschließlich Kohärenzbedingungen (siehe Anforderung 20 auf Seite 87) formuliert werden, dann ist das Alter der Replikate durch die entsprechenden Abstandsmaße für die Zeitdifferenz, dem Versionsabstand bzw. der Wertdifferenz begrenzt. In [YV02] wird für diese drei Dimensionen ein Abstandsvektor definiert.

Neben den Kohärenzbedingungen gibt es bei RegRess weitere Bedingungen, mit denen die Konsistenz beeinflusst wird. Insbesondere die technischen Konsistenzbedingungen wie z.B. die Anforderung für die Performance (siehe Anforderung 25 auf Seite 89) können eine Veralterung der Replikate bewirken, die auf den Zustand der beteiligten Systeme basieren und damit schwerer vorhersagbar sind. Eine Grenze für das Alter kann dann dadurch festgelegt werden, dass entsprechende Regeln spezifiziert werden, wie z.B. Widerspruchsregeln, die die Kohärenzbedingungen gegenüber technischen Konsistenzbedingungen priorisieren. Ohne eine derartige Priorisierung können Replikate theoretisch beliebig veralten, wobei unter „normalen Umständen“, d.h. z.B. keine endlose Netzpartitionierung, die asynchrone Aktualisierung ausgelassene Aktualisierungen nachholt und somit letztendlich für einen konsistenten Zustand sorgt.

In Abschnitt 3.2.3 wurden verschiedene abgeschwächte Konsistenzbegriffe vorgestellt, die entweder auf einzelne Replikate im Sinne von Abstandsmaßen oder auf die gesamte replizierte Datenbank abstellen. Im letzten Fall werden entweder die Abstände der einzelnen Replikate aufaddiert und gemittelt oder eine Quote berechnet. Wie bereits erwähnt, kann dadurch die gewährleistete Konsistenz bzw. das Korrektheitskriterium einer Replikationsstrategie genauer quantifiziert werden. Außerdem hilft die genauere Bestimmung der Konsistenz dabei, einen optimalen Kompromiss hinsichtlich der Zielkonflikte Konsistenz, Verfügbarkeit und Performance zu finden (siehe Abschnitt 3.1).

Auch bei RegRess liegt die Priorität nicht zwingend auf der Konsistenz, sondern mittels Regeln kann eine geeignete Abstimmung hinsichtlich der Replikationsziele erreicht werden (siehe Abschnitt 4.4). Um eine für RegRess geeignete Metrik für Konsistenz zu erhalten, wird in diesem Abschnitt der abgeschwächte Konsistenzbegriff *„Konsistenzgrad“* eingeführt. Der Konsistenzgrad ist ein normierter, reeller Wert im Bereich [0..1], wobei der Wert 1 Konsistenz meint und kleiner werdende Werte abnehmende Konsistenz widerspiegeln. Die Ermittlung des Konsistenzgrades erfolgt entweder durch eine Messung am realen System oder durch eine Prognose, mit der eine Vorhersage des Konsistenzverhaltens von RegRess erreicht wird. Die Prognose basiert entweder auf eine Simulation oder wird mittels stochastischem Modell berechnet. Somit ist der restliche Abschnitt wie folgt gegliedert:

- Gemessener Konsistenzgrad
- Simulierter Konsistenzgrad
- Berechneter Konsistenzgrad

Gemessener Konsistenzgrad

Unabhängig davon, welche konkreten Konsistenzbedingungen bei RegRess formuliert sind, lässt sich zumindest zur Laufzeit das Alter der Replikate einfach mittels der Replica Queue ermitteln: Bei der Zeit- bzw. Wertdifferenz wird ein Replikat R mit dem letzten Auftrag der Replica Queue, das R ändert, verglichen (bei der Zeitdifferenz muss ein Zeitstempel für R mitgeführt werden). Der Versionsabstand kann ausschließlich über die Anzahl Aufträge eines Replikats in der Replica Queue errechnet werden, d.h. es wird kein Zugriff auf das lokale Replikat oder das Führen von Protokolldaten benötigt. Der Versionsabstand ist hier wie folgt definiert (zur Notation siehe Abschnitt 4.1.2):

Definition 38 ***Versionsabstand***: *Die Versionsnummer $v(R_k^o, t)$ eines Replikats R_k^o, das auf der k-ten Komponente mit Replikat lokalisiert ist und zum o-ten logischen Objekt gehört, zum Zeitpunkt t ist die Anzahl der Schreibzugriffe auf dieses Replikat R_k^o bis einschließlich zum Zeitpunkt t. Der Versionsabstand $\delta(R_k^o, t)$ eines Replikats R_k^o zum Zeitpunkt t ist die Differenz der Versionsnummer eines aktuellen Replikats des o-ten logischen Objekts zum Replikat R_k^o:*

$$\delta(R_k^o, t) = v(R_a^o, t) - v(R_k^o, t)$$

mit R_a^o ist zum Zeitpunkt t ein aktuelles Replikat.

Die Versionsnummer ist ein Zähler für die Schreibzugriffe, die auf ein Replikat durchgeführt werden, d.h. es handelt sich um eine natürliche Zahl. Weil bei RegRess an jedes Replikat alle Änderungen propagiert werden, hat ein aktuelles Replikat des logischen Objekts O die höchste Versionsnummer aller Replikate, die zu O gehören. Damit ist der Versionsabstand eine natürliche Zahl inklusive der Null, also $\delta \in \mathbb{N}_0$. Aktuelle Replikate haben den Versionsabstand $\delta = 0$ und veraltete Replikate haben einen positiven Versionsabstand $\delta > 0$. Der Versionsabstand eines veralteten Replikats kann bei RegRess der Replica Queue entnommen werden, denn er ist die Anzahl an Aufträgen in der Replica Queue, die dieses Replikat schreiben. Sei $a(R_k^o, t)$ die Anzahl Aufträge in der Replica Queue für Replikat R_k^o zum Zeitpunkt t, dann gilt:

$$\delta(R_k^o, t) = a(R_k^o, t) \tag{4.4}$$

Mit Hilfe des Versionsabstands wird nun ein normierter Konsistenzbegriff definiert, nämlich der Konsistenzgrad eines Replikats. Für die Normierung wird eine Hyperbelfunktion genutzt, die den Versionsabstand beinhaltet:

Definition 39 ***Konsistenzgrad eines Replikats***: *Der Konsistenzgrad $KG_{Rep}(R_k^o, t)$ eines Replikats R_k^o zum Zeitpunkt t ist definiert als:*

$$KG_{Rep}(R_k^o, t) = \frac{1}{1 + \delta(R_k^o, t)}$$

Der Konsistenzgrad eines Replikats KG_{Rep} liegt somit im Bereich [0..1]. Ein aktuelles Replikat ist konsistent und hat den Konsistenzgrad 1. Je älter ein Replikat, desto größer wird der Versionsabstand und desto kleiner wird der Konsistenzgrad. Die Verwendung einer Hyperbelfunktion zur Normierung bietet den Vorteil, dass bei großen Versionsabständen eine weitere Vergrößerung des Versionsabstandes den Konsistenzgrad $KG_{Rep}(R_k^o, t)$ weniger verändert als bei kleinen Versionsabständen. Dies kann derart ausgelegt werden, dass z.B. eine Vergrößerung des Versionsabstandes von eins auf zwei schwerer wiegt als eine Vergrößerung von 100 auf 101. Der Nachteil des Konsistenzgrades $KG_{Rep}(R_k^o, t)$ liegt darin, dass der Wert schwerer zu interpretieren ist, als z.B. der Versionsabstand $\delta(R_k^o, t)$.

Anstelle des Versionsabstands könnte in der Definition 39 auch eine andere Abstandsfunktion verwendet werden, die z.B. die Zeitdifferenz und/oder die Wertdifferenz berücksichtigt. Auch eine eindimensionale Abstandsfunktion aus einem dreidimensionalen Konsistenzvektor, der Zeitdifferenz, Versionsabstand und Wertdifferenz beinhaltet (siehe [YV02]), könnte verwendet werden. Zusätzlich könnte ein Gewichtsfaktor zur Abstandsfunktion eingeführt werden, um die Hyperbel zu „strecken“ und damit das Fallen der Konsistenzgrad-Funktion zu beeinflussen. Wie bereits erwähnt, ist bei RegRess die Ermittlung des Versionsabstands über die Replica Queue besonders einfach. Mit der Gleichung 4.4 lässt sich der Konsistenzgrad eines veralteten Replikats wie folgt berechnen:

$$KG_{Rep}(R_k^o, t) = \frac{1}{1 + a(R_k^o, t)} \tag{4.5}$$

Analog zu der Frische einer replizierten Datenbank (siehe Definition 26 auf Seite 38), die mittels der Frische der Replikate berechnet wird, kann ein Konsistenzgrad einer replizierten Datenbank auf Basis der Konsistenzgrade der beteiligten Replikate definiert werden. Laut Annahme (siehe Abschnitt 4.1.1) wird von einer voll-replizierten Datenbank mit dem Replikationsgrad n ausgegangen, d.h. es existieren n Komponenten mit Replikat. Wenn nun m logische Objekte vorliegen, dann gibt es insgesamt $n \cdot m$ Replikate, die in folgender Definition aufaddiert werden:

Definition 40 ***Konsistenzgrad einer voll-replizierten Datenbank****: Der Konsistenzgrad $KG_{DB}(t)$ einer voll-replizierten Datenbank zum Zeitpunkt t ist definiert als:*

$$KG_{DB}(t) = \frac{1}{n \cdot m} \sum_{k=1}^{n} \sum_{o=1}^{m} KG_{Rep}(R_k^o, t)$$

mit n = Replikationsgrad (= Anzahl Komponenten), m = Anzahl logischer Objekte

Wie oben erwähnt, ist der Konsistenzgrad schwerer zu interpretieren als der Versionsabstand. Daher soll hier auch ein Versionsabstand für eine voll-replizierte Datenbank angegeben werden, der analog zum Konsistenzgrad durch Mittelung berechnet wird:

Definition 41 ***Versionsabstand einer voll-replizierten Datenbank****: Der Versionsabstand $V_{DB}(t)$ einer voll-replizierten Datenbank zum Zeitpunkt t ist definiert als:*

$$V_{DB}(t) = \frac{1}{n \cdot m} \sum_{k=1}^{n} \sum_{o=1}^{m} \delta(R_k^o, t)$$

mit n = Replikationsgrad (= Anzahl Komponenten), m = Anzahl logischer Objekte

Um die Frische einer replizierten Datenbank über einen Zeitraum zu berechnen, wird in [CGM00] die Frischefunktion über den Zeitraum integriert. Auch der Konsistenzgrad eines Zeitraums, ob für ein Replikat oder für die voll-replizierte Datenbank, kann durch Integration über den gewünschten Zeitraum definiert werden. Bei einem gemessenen Konsistenzgrad werden jedoch eher l Messungen in dem Zeitraum t_1 bis t_l durchgeführt, die gemittelt werden:

$$KG_{DB} = \frac{1}{n \cdot m \cdot l} \sum_{k=1}^{n} \sum_{o=1}^{m} \sum_{i=1}^{l} KG_{Rep}(R_k^o, t_i), \text{ im Zeitraum } t_1 \text{ bis } t_l \tag{4.6}$$

bzw. analog für den Versionsabstand:

$$V_{DB} = \frac{1}{n \cdot m \cdot l} \sum_{k=1}^{n} \sum_{o=1}^{m} \sum_{i=1}^{l} \delta(R_k^o, t), \text{ im Zeitraum } t_1 \text{ bis } t_l \tag{4.7}$$

In Gleichung 4.6 werden zunächst mit dem rechten Summenzeichen die l gemessenen Konsistenzgrade eines Replikats R_k^o aufaddiert. Mit dem Faktor $\frac{1}{l}$ erhält man den mittleren Konsistenzgrad des Replikats. Durch das mittlere Summenzeichen werden die mittleren Konsistenzgrade aller Replikate einer lokalen Datenbank bzw. einer Komponente mit Replikat aufaddiert. Mit dem Faktor $\frac{1}{m}$ erhält man dann den mittleren Konsistenzgrad einer Komponente mit Replikat. Diese werden durch das erste Summenzeichen für jede Komponente mit Replikat aufaddiert, sodass man mit dem Faktor $\frac{1}{n}$ den mittleren Konsistenzgrad der voll-replizierten Datenbank im Zeitraum t_1 bis t_l erhält.

Um den Konsistenzgrad einer voll-replizierten Datenbank zu berechnen, müssen nicht die Konsistenzgrade aller Replikate berechnet werden, weil der Konsistenzgrad aller aktuellen Replikate gleich 1 ist und der Konsistenzgrad der veralteten Replikate sich über die Replica Queue bestimmen lässt (siehe Gleichung 4.5). Sei $z_v(t)$ die Anzahl veralteter Replikate zum Zeitpunkt t, d.h.

es gibt $z_v(t)$ unterschiedliche Replikate zum Zeitpunkt t, für die Aufträge in der Replica Queue stehen, dann berechnet sich zum Zeitpunkt t die Anzahl $z_a(t)$ der aktuellen Replikate bei einem Replikationsgrad von n und m logischen Objekten wie folgt:

$$z_a(t) = n \cdot m - z_v(t) \tag{4.8}$$

Weiterhin sei $a_i(t)$, $i = 1, 2, ..z_v(t)$ die Anzahl der Aufträge in der Replica Queue eines veralteten Replikats R_k^o, $k = 1, 2, .., n$, $o = 1, 2, .., m$. Wenn $a_i(t)$ die Aufträge für R_k^o und $a_j(t)$ die Aufträge für R_l^p zählt und $i \neq j$ ist, dann gilt: $k \neq l \vee o \neq p$. Die $a_i(t)$ geben also zum Zeitpunkt t den Versionsabstand für paarweise verschiedene, veraltete Replikate an (siehe Gleichung 4.4). Außerdem decken die $a_i(t)$ die gesamten Aufträge ab, d.h. wenn $x(t)$ die Anzahl aller Aufträge in der Replica Queue zum Zeitpunkt t ist, dann gilt:

$$x(t) = \sum_{i=1}^{z_v(t)} a_i(t) \tag{4.9}$$

Allgemein kann der Konsistenzgrad eines Replikats R_k^o, $k = 1, 2, .., n$, $o = 1, 2, .., m$ zum Zeitpunkt t wie folgt ausgedrückt werden, wenn $a_i(t)$, $i = 1, 2, .., z_v(t)$ die Anzahl der Aufträge in der Replica Queue für veraltete Replikate ist (siehe Gleichung 4.5):

$$KG_{Rep}(R_k^o, t) = \begin{cases} \frac{1}{1 + a_i(t)} & R_k^o \text{ zum Zeitpunkt } t \text{ veraltet} \\ 1 & R_k^o \text{ zum Zeitpunkt } t \text{ aktuell} \end{cases} \tag{4.10}$$

Von den $n \cdot m$ Replikaten sind zum Zeitpunkt t genau $z_a(t)$ Replikate aktuell und $z_v(t)$ Replikate veraltet. Daher gilt für die Summe der Konsistenzgrade aller Replikate:

$$\sum_{k=1}^{n} \sum_{o=1}^{m} KG_{Rep}(R_k^o, t) = z_a(t) + \sum_{i=1}^{z_v(t)} \frac{1}{1 + a_i(t)} \tag{4.11}$$

Der Konsistenzgrad einer voll-replizierten Datenbank (siehe Definition 40) kann nun mit der Gleichung 4.11 allein durch die Anzahl der Aufträge für verschiedene, veraltete Replikate berechnet werden:

$$\begin{aligned} KG_{DB}(t) &= \frac{1}{n \cdot m} \sum_{k=1}^{n} \sum_{o=1}^{m} KG_{Rep}(R_k^o, t) \\ &= \frac{1}{n \cdot m} (z_a(t) + \sum_{i=1}^{z_v(t)} \frac{1}{1 + a_i(t)}) \\ &= \frac{1}{n \cdot m} (n \cdot m - z_v(t) + \sum_{i=1}^{z_v(t)} \frac{1}{1 + a_i(t)}) \\ &= 1 - \frac{1}{n \cdot m} (z_v(t) - \sum_{i=1}^{z_v(t)} \frac{1}{1 + a_i(t)}) \\ &= 1 - \frac{1}{n \cdot m} \sum_{i=1}^{z_v(t)} (1 - \frac{1}{1 + a_i(t)}) \\ &= 1 - \frac{1}{n \cdot m} \sum_{i=1}^{z_v(t)} \frac{a_i(t)}{1 + a_i(t)} \end{aligned} \tag{4.12}$$

Der Ausdruck $a_i(t)/(1 + a_i(t))$ kann als „Inkonsistenzgrad“ aufgefasst werden, d.h. umgekehrt zum Konsistenzgrad ist ein Replikat umso inkonsistenter, je näher der Wert bei 1 liegt. Diese Inkonsistenzen werden in der Gleichung 4.12 aufaddiert und über alle $n \cdot m$ Replikate gemittelt. Der so ermittelte Inkonsistenzanteil wird dann vom Konsistenzwert 1 subtrahiert, um den Konsistenzgrad der voll-replizierten Datenbank zu erhalten. Analog zur Gleichung 4.6 kann über die „Inkonsistenzen“ ein gemittelter Konsistenzgrad für einen Zeitraum berechnet werden.

Der Vorteil der einfachen Berechnung des Konsistenzgrades $KG_{DB}(t)$ über die Anzahl der Aufträge in der Replica Queue, insbesondere dann, wenn im Verhältnis zur Gesamtanzahl $n \cdot m$ an Replikaten wenig veraltete Replikate $z_v(t)$ vorhanden sind, zeigt auch die Schwäche eines derart berechneten Konsistenzgrades. Wenn $z_v(t) \ll n \cdot m$ gilt, dann ist $KG_{DB}(t) \approx 1$, d.h. der Konsistenzgrad liegt nahe bei 1. Dies muss nun nicht daran liegen, dass Inkonsistenzen selten auftreten oder kaum bemerkt werden, sondern dass in der verteilten Datenbank viele Objekte gespeichert sind, z.B. auch „Datenleichen“, auf die selten zugegriffen wird. Falls nun auf eine Teilmenge der Replikate zugegriffen wird und genau diese Replikate häufiger veralten, dann erfolgen mehr Zugriffe auf inkonsistente Daten als es der Konsistenzgrad $KG_{DB}(t)$ angibt. Abhilfe schafft dann der abgeschwächte Konsistenzbegriff „Bemerkte Frische“ (siehe Abschnitt 3.2.3).

Die bemerkte Frische (siehe Definition 27 auf Seite 39) ist definiert als das Verhältnis von Zugriffen, die ein aktuelles Replikat betreffen, zu den gesamten Zugriffen, ausgedrückt in Prozent. Bei den Zugriffen wird nicht das Alter der zugegriffenen Replikate berücksichtigt. In der folgenden Definition wird das Alter der Replikate, auf die zugegriffen wird, in Form des Konsistenzgrades für Replikate, also KG_{Rep}, einbezogen. Weil bei RegRess Schreib-/Schreibkonflikte$_R$ verhindert werden, stellt der folgende Konsistenzgrad ausschließlich auf Lesezugriffe ab:

Definition 42 ***Konsistenzgrad von Lesezugriffen****: Ein Lesezugriff l_i greift zum Zeitpunkt t_i auf das Replikat R_i zu. Dann ist der Konsistenzgrad KG_{Lesen} von z_l Lesezugriffen definiert als:*

$$KG_{Lesen} = \frac{1}{z_l} \sum_{i=1}^{z_l} KG_{Rep}(R_i, t_i)$$

Beim Konsistenzgrad KG_{Lesen} handelt es sich ebenfalls um einen normierten Wert, weil die z_l normierten Konsistenzgrade der zugegriffenen Replikate gemittelt werden. Die Konsistenzgrade der zugegriffenen Replikate können gemäß Gleichung 4.10 berechnet werden, d.h. auch beim Konsistenzgrad KG_{Lesen} können bei veralteten Replikaten die Anzahl an Aufträgen in der Replica Queue genutzt werden. Wenn in einem Zeitraum die Lesezugriffe gleichverteilt über alle Replikate stattfinden, dann werden die Konsistenzgrade aller Replikate gemäß des Konsistenzgrades KG_{DB} einmal oder mehrfach aufaddiert, d.h. es gilt:

(4.13) $KG_{Lesen} \approx KG_{DB}$, bei gleichverteilten Lesezugriffen in einem festen Zeitraum

Auch bei gleichverteilten Lesezugriffen sind die Konsistenzgrade im Allgemeinen nur ungefähr gleich, weil die einzelnen Konsistenzgrade der Replikate unter Umständen zu unterschiedlichen Zeitpunkten ermittelt werden. Wenn die Lesezugriffe stark ungleich verteilt sind, dann können die beiden Konsistenzgrade erheblich voneinander abweichen, wobei der Konsistenzgrad KG_{Lesen} eher der „Außenwirkung“, d.h. der Sicht der Clients, entsprechen dürfte. Der Nachteil des Konsistenzgrades KG_{Lesen} ist es, dass er schwieriger zu berechnen ist als der Konsistenzgrad KG_{DB}, weil alle Lesezugriffe, insbesondere lokale Lesezugriffe, zentral protokolliert und ausgewertet werden müssen.

Die gemessenen Konsistenzgrade, ob KG_{DB} oder KG_{Lesen} bieten den Vorteil, dass sie das Konsistenzverhalten der hier vorgestellten Replikationsstrategie RegRess genau widerspiegeln und dass sie mehr oder weniger einfach zu ermitteln sind. Der Nachteil besteht darin, dass die Konsistenz erst zur Laufzeit berechnet werden kann. Wenn z.B. nachträglich Regeln für die synchrone Aktualisierung geändert werden, kann vorab keine genaue Aussage zur gewährleisteten Konsistenz gemacht werden. Es sind bestenfalls Tendenzen zu nennen. Die nachfolgenden

Konsistenzgrade dienen der Bestimmung der Konsistenz im Voraus, d.h. nachdem die Regeln festgelegt sind.

Simulierter Konsistenzgrad

Um vorab die gewährleistete Konsistenz einer Replikationsstrategie zu bestimmen, muss betrachtet werden, wie bei der Replikationsstrategie die Zugriffe koordiniert werden. Dabei gibt es grundsätzlich zwei Vorgehensweisen, nämlich die Konsistenz analytisch oder durch Simulation zu bestimmen. In diesem Paragraphen wird zunächst auf einen Konsistenzgrad eingegangen, der auf Basis eines Simulationsmodells ermittelt wird. Da die Simulation auch zur Evaluation dient, wird hierauf detailliert im Kapitel 7 eingegangen.

Allgemein gesagt wird ein Simulationsmodell benötigt, dass die Systemlandschaft, in der die Replikationsstrategie eingesetzt wird, möglichst ähnlich abbildet. Es muss ein verteiltes System simuliert werden, dessen lokale Rechner Komponenten mit Replikaten beherbergen (siehe Abbildung 4.1 auf Seite 61). Auf diesen Komponenten erfolgen Schreib- und Lesezugriffe, die durch die simulierte Replikationsstrategie koordiniert werden. Während eines Simulationslaufs kann dann die Konsistenz gemessen werden, d.h. der simulierte Konsistenzgrad entspricht dem gemessenen Konsistenzgrad, nur das in diesem Fall am Simulationsmodell gemessen wird:

Definition 43 ***Konsistenzgrad durch Simulation****: Der Konsistenzgrad KG_{Sim} wird durch Simulation ermittelt. In der Simulation wird der Konsistenzgrad gemessen, d.h.*

$$KG_{Sim} = KG_{DB}$$

wobei KG_{DB} der gemessene Konsistenzgrad der Replikationsstrategie des Simulationsmodells ist.

Damit der Konsistenzgrad KG_{Sim} durch Simulation ermittelt werden kann, muss zunächst einmalig ein entsprechendes Simulationsmodell entwickelt werden, wobei die betrachtete Replikationsstrategie implementiert werden muss. Änderungen der Replikationsstrategie müssen auch im Simulationsmodell angepasst werden. Bei RegRess ergeben sich Änderungen vor allem durch andere Regeln, d.h. das Simulationsmodell muss einen entsprechenden Regelinterpreter beinhalten (siehe Kapitel 7). Dann allerdings kann das neue Konsistenzverhalten von RegRess durch die veränderten Regeln einfach durch einen erneuten Simulationslauf bestimmt werden. Der Vorteil liegt also in der einfachen Handhabung nach dem einmaligen Erstellen des Simulationsmodells. Der Nachteil liegt darin, dass die Ergebnisse stark von der Affinität des Simulationsmodells zur realen Systemlandschaft abhängen.

Berechneter Konsistenzgrad

Bei einer analytischen Vorhersage der gewährleisteten Konsistenz von asynchronen Replikationsstrategien müssen im Allgemeinen die asynchronen Aktualisierungen betrachtet werden, d.h. es wird die Frage untersucht, wann die verspäteten Aktualisierungen nachgeholt werden. Einerseits können Abstandsmaße, andererseits Quoten oder Wahrscheinlichkeiten für die Konsistenz verwendet werden (siehe Abschnitt 3.2.3). Wie bereits erwähnt, kann bei RegRess die gewährleistete Konsistenz dann mit Abstandsmaßen bestimmt werden, wenn ausschließlich Kohärenzbedingungen als Regeln spezifiziert werden. Weil aber auch andere Regeln festgelegt werden können, wird in diesem Paragraph ein Konsistenzgrad definiert, der ein wahrscheinlichkeitstheoretisches Maß für das Konsistenzverhalten von RegRess angibt. Hierfür könnte grundsätzlich auch die *„probabilistische Frische“* (siehe Definition 29 auf Seite 39) verwendet werden, aber um die speziellen Merkmale von RegRess zu berücksichtigen, wird ein *„probabilistischer Konsistenzgrad“* definiert, der auf Markov-Ketten [SW04b] basiert.

Bei der Definition des probabilistischen Konsistenzgrades wird analog zum vorhergehenden Paragraphen verfahren, d.h. es werden zunächst Begriffe auf Ebene der Replikate definiert, die in die übergeordneten Begriffe einfließen. Der Konsistenzgrad $KG_{Rep}(R_k^o, t)$ (siehe Definition 39 auf Seite 92) basiert auf dem Versionsabstand, den ein Replikat R_k^o, zum Zeitpunkt t hat. Für

ein wahrscheinlichkeitstheoretisches Maß (probabilistisches Maß) wird die Wahrscheinlichkeit genutzt, mit der ein Replikat einen bestimmten Versionsabstand hat. Angemerkt sei, dass anstelle eines Versionsabstands auch andere Abstandsmaße verwendet werden könnten, sich der Versionsabstand aber bei RegRess anbietet (siehe oben). Somit folgt die Definition einer Versionsabstandswahrscheinlichkeit:

Definition 44 ***Versionsabstandswahrscheinlichkeit****:* *Die Versionsabstandswahrscheinlichkeit* $P(Versionsabstand\ des\ Replikats\ R_k^o = i) = p_i(R_k^o) \in [0..1]$, $i = 0, 1, 2, ..$ *ist die Wahrscheinlichkeit dafür, dass ein Replikat* R_k^o, $k = 1, 2, ..n$, $o = 1, 2, ..m$ *einen Versionsabstand* i *hat. Es gilt:*

$$\sum_{i=0}^{\infty} p_i(R_k^o) = 1$$

Bevor erläutert wird, wie die Versionsabstandswahrscheinlichkeiten eines Replikats mit Markov-Ketten bestimmt werden, folgen einige Definitionen, die diese Wahrscheinlichkeiten verwenden. Zunächst wird eine diskrete Zufallsvariable [Fis88] analog zum Versionsabstand (siehe Definition 38 auf Seite 92) definiert:

Definition 45 ***Zufallsvariable für den Versionsabstand****:* *Die Abbildung* V_{ZV} *ist eine Zufallsvariable für den Versionsabstand und ist wie folgt definiert:*

$$V_{ZV}(Versionsabstand = i) = V_{ZV}(i) = i$$

Weil auch hier analog zum gemessenen Konsistenzgrad eine Normierung des Versionsabstandes verwendet werden soll (siehe oben), wird eine Zufallsvariable definiert, die entsprechend dem gemessenen Konsistenzgrad KG_{Rep} (siehe Definition 39 auf Seite 92) eine Hyperbelfunktion für die Normierung nutzt:

Definition 46 ***Zufallsvariable für den normierten Versionsabstand****:* *Die Abbildung* KG_{ZV} *ist eine Zufallsvariable für den normierten Versionsabstand und ist wie folgt definiert:*

$$KG_{ZV}(Versionsabstand = i) = KG_{ZV}(i) = \frac{1}{1+i}$$

Die Versionsabstandswahrscheinlichkeiten bilden eine Wahrscheinlichkeitsverteilung der diskreten Zufallsvariablen V_{ZV} bzw. KG_{ZV}. Der Erwartungswert (auch Mittelwert, siehe z.B. [Fis88]) einer Zufallsvariablen ist der Wert, der als Mittelwert eines oftmals wiederholten Experiments erwartet wird. Bei diskreten Zufallsvariablen berechnet sich der Erwartungswert als Skalarprodukt aus Wahrscheinlichkeitsverteilung einer Zufallsvariablen mit der Zufallsvariablen. Der erwartete Versionsabstand eines Replikats ist demzufolge der Versionsabstand eines Replikats, der im Mittel erwartet wird, und wie folgt definiert ist:

Definition 47 ***Erwarteter Versionsabstand eines Replikats****:* *Seien* $p_i(R_k^o)$, $i = 0, 1, 2, ..$ *die Versionsabstandswahrscheinlichkeiten eines Replikats* R_k^o *und* $V_{ZV}(i)$ *eine Zufallsvariable für den Versionsabstand. Dann ist der erwartete Versionsabstand* $V_{ErwRep}(R_k^o)$ *des Replikats* R_k^o *definiert als:*

$$V_{ErwRep}(R_k^o) = \sum_{i=0}^{\infty} p_i(R_k^o) \cdot V_{ZV}(i) = \sum_{i=0}^{\infty} p_i(R_k^o) \cdot i$$

Analog zum erwarteten Versionsabstand eines Replikats wird der erwartete Konsistenzgrad eines Replikats definiert, der dem Konsistenzgrad eines Replikats entspricht, der im Mittel erwartet wird:

Definition 48 ***Erwarteter Konsistenzgrad eines Replikats****: Seien $p_i(R_k^o)$, $i = 0, 1, 2, ..$ die Versionsabstandswahrscheinlichkeiten eines Replikats R_k^o und $KG_{ZV}(i)$ eine Zufallsvariable für den Versionsabstand. Dann ist der erwartete Konsistenzgrad $KG_{ErwRep}(R_k^o)$ des Replikats R_k^o definiert als:*

$$KG_{ErwRep}(R_k^o) = \sum_{i=0}^{\infty} p_i(R_k^o) \cdot KG_{ZV}(i) = \sum_{i=0}^{\infty} \frac{p_i(R_k^o)}{1+i}$$

In der Definition 47 bzw. Definition 48 werden im Allgemeinen nicht unendlich viele Versionsabstandswahrscheinlichkeiten $p_i(R_k^o)$ benötigt. Die Versionsabstandswahrscheinlichkeiten $p_i(R_k^o)$ konvergieren „normalerweise“ gegen null, da ansonsten die Replikate divergieren würden. Bei der Replikationsstrategie RegRess wird z.B. der mögliche höchste Versionsabstand im Allgemeinen durch Replikationsregeln festgelegt. Es ist auch möglich, bei der praktischen Berechnung des erwarteten Versionsabstandes $V_{ErwRep}(R_k^o)$ bzw. des erwarteten Konsistenzgrades $KG_{ErwRep}(R_k^o)$ die Summe auf eine bestimmte Anzahl Summanden zu beschränken, wobei eine entsprechende Fehlertoleranz berücksichtigt wird.

Wie bei den gemessenen Konsistenzgraden kann nun ein erwarteter Versionsabstand bzw. ein erwarteter Konsistenzgrad der voll-replizierten Datenbank berechnet werden, indem über die erwarteten Versionsabstände bzw. erwarteten Konsistenzgrade aller Replikate gemittelt wird. Weil bei RegRess jedoch Replikate, die ähnlich behandelt werden, in Replikationseinheiten zusammengefasst sind (siehe Definition 35 auf Seite 65) und sich damit die Berechnung vereinfacht (siehe unten), folgen zunächst die Definitionen des erwarteten Versionsabstandes und des erwarteten Konsistenzgrades einer Replikationseinheit:

Definition 49 ***Erwarteter Versionsabstand einer Replikationseinheit:*** *Sei a_i, $i = 1, 2, ..$ die Anzahl Replikate einer Replikationseinheit RE_i und R_j das j-te Replikat von RE_i. Dann ist der erwartete Versionsabstand $V_{ErwRE}(RE_i)$ definiert als:*

$$V_{ErwRE}(RE_i) = \frac{1}{a_i} \sum_{j=1}^{a_i} V_{ErwRep}(R_j)$$

Definition 50 ***Erwarteter Konsistenzgrad einer Replikationseinheit****: Sei a_i, $i = 1, 2, ..$ die Anzahl Replikate einer Replikationseinheit RE_i und R_j das j-te Replikat von RE_i. Dann ist der erwartete Konsistenzgrad $KG_{ErwRE}(RE_i)$ definiert als:*

$$KG_{ErwRE}(RE_i) = \frac{1}{a_i} \sum_{j=1}^{a_i} KG_{ErwRep}(R_j)$$

In der Definition 49 bzw. Definition 50 werden die erwarteten Versionsabstände bzw. die erwarteten Konsistenzgrade aller Replikate einer Replikationseinheit addiert. Eine Vereinfachung ist dann möglich, wenn diese erwarteten Versionsabstände bzw. erwarteten Konsistenzgrade gleich sind. Bei RegRess werden die Regeln je Replikationseinheit spezifiziert, sodass alle Replikate einer Replikationseinheit gleich behandelt werden. Wenn dann noch die Schreib- und Leseraten innerhalb dieser Gruppe „gleich“ sind, dann sind die Versionsabstandswahrscheinlichkeiten der Replikate einer Replikationseinheit (siehe unten) und damit die erwarteten Versionsabstände bzw. die erwarteten Konsistenzgrade gleich. Somit kann vereinfacht der erwartete Versionsabstand bzw. der erwartete Konsistenzgrad einer Replikationseinheit gleich dem erwarteten Versionsabstand bzw. dem erwarteten Konsistenzgrad eines beliebigen Replikats der Replikationseinheit gesetzt werden:

$$V_{ErwRE}(RE_i) = V_{ErwRep}(R_j) \tag{4.14}$$

$$\text{mit } R_j \in RE_i \text{ und } R_j, R_k \in RE_i \Rightarrow V_{ErwRep}(R_j) = V_{ErwRep}(R_k)$$

(4.15) $$KG_{ErwRE}(RE_i) = KG_{ErwRep}(R_j)$$

$$\text{mit } R_j \in RE_i \text{ und } R_j, R_k \in RE_i \Rightarrow KG_{ErwRep}(R_j) = KG_{ErwRep}(R_k)$$

Der erwartete Versionsabstand bzw. der erwartete Konsistenzgrad einer voll-replizierten Datenbank, der hier aus Analogie zur *„probabilistischen Frische“* (siehe Definition 29 auf Seite 39) als probabilistischer Versionsabstand bzw. probabilistischer Konsistenzgrad bezeichnet wird, ergibt sich nun durch Mittelung der erwarteten Versionsabstände bzw. der erwarteten Konsistenzgrade der zugehörigen Replikationseinheiten. Diese Mittelung erfolgt gewichtet nach der Anzahl an Replikaten, die zu einer Replikationseinheit gehören. Zusätzlich kann beim probabilistischen Versionsabstand bzw. beim probabilistischen Konsistenzgrad jeder Replikationseinheit ein Gewicht zugeordnet werden, mit dem z.B. unterschiedliches Zugriffsverhalten abgedeckt werden kann. Hierdurch können Aspekte berücksichtigt werden, die bei der Definition des Konsistenzgrades von Lesezugriffen KG_{Lesen} diskutiert wurden (siehe Definition 42 auf Seite 95). Damit dienen folgende Definitionen zur Abschätzung des Konsistenzverhaltens der Replikationsstrategie RegRess:

Definition 51 ***Probabilistischer Versionsabstand****: Für eine voll-replizierte Datenbank seien z_r Replikationseinheiten RE_i, $i = 1, 2, .., z_r$ spezifiziert. Sei a_i, $i = 1, 2, .., z_r$ die Anzahl Replikate der Replikationseinheit RE_i und sei g_i, $i = 1, 2, .., z_r$ ein Gewichtsfaktor der Replikationseinheit RE_i. Dann ist der probabilistische Versionsabstand V_{Prob} definiert als:*

$$V_{Prob} = \frac{1}{\sum_{i=1}^{z_r} g_i \cdot a_i} \sum_{i=1}^{z_r} g_i \cdot a_i \cdot V_{ErwRE}(RE_i)$$

Definition 52 ***Probabilistischer Konsistenzgrad****: Für eine voll-replizierte Datenbank seien z_r Replikationseinheiten RE_i, $i = 1, 2, .., z_r$ spezifiziert. Sei a_i, $i = 1, 2, .., z_r$ die Anzahl Replikate der Replikationseinheit RE_i und sei g_i, $i = 1, 2, .., z_r$ ein Gewichtsfaktor der Replikationseinheit RE_i. Dann ist der probabilistische Konsistenzgrad KG_{Prob} definiert als:*

$$KG_{Prob} = \frac{1}{\sum_{i=1}^{z_r} g_i \cdot a_i} \sum_{i=1}^{z_r} g_i \cdot a_i \cdot KG_{ErwRE}(RE_i)$$

Insbesondere wenn sich die vereinfachte Rechnung für die erwarteten Versionsabstände bzw. für die erwarteten Konsistenzgrade der Replikationseinheiten gemäß Gleichung 4.14 bzw. Gleichung 4.15 nutzen lässt, kann der probabilistische Versionsabstand V_{Prob} bzw. der probabilistische Konsistenzgrad KG_{Prob} durch z_r Versionsabstände bzw. durch z_r Konsistenzgrade repräsentativer Replikate berechnet werden. Angemerkt sei weiterhin, dass auch die Zuordnung eines Replikats zu mehreren Replikationseinheiten kein Problem darstellt, weil eine Mittelung erfolgt. Abschließend soll ein kleines Beispiel den Einfluss der Gewichtung nach Anzahl Replikaten und Gewichtsfaktor zeigen. Es sei angenommen, dass eine voll-replizierte Datenbank mit zwei Replikationseinheiten vorliegt: RE_1 beinhaltet sechs Replikate, $KG_{ErwRE}(RE_1) = 0,8$; RE_2 beinhaltet vier Replikate, $KG_{ErwRE}(RE_2) = 0,3$. Die Gewichtsfaktoren g_1 und g_2 werden gleich 1 gesetzt, d.h. keine expliziten Gewichte:

$$KG_{Prob} = \frac{1}{6+4}(6 \cdot 0,8 + 4 \cdot 0,3) = 0,6$$

Das einfache Mittel der erwarteten Konsistenzgrade der beiden Replikationseinheiten ergibt: $(0,8 + 0,3)/2 = 0,55$. Durch Gewichtung der Replikationseinheiten mit ihrer Größe, sprich Anzahl an Replikaten, ergibt sich eine Erhöhung des probabilistischen Konsistenzgrades KG_{Prob} auf $0,6$, weil die Replikationseinheit RE_1, die einen höheren Konsistenzgrad als Replikationseinheit RE_2 aufweist, mehr Replikate beinhaltet und somit überdurchschnittlich gewichtet wird.

Eine weitere Erhöhung des probabilistischen Konsistenzgrades KG_{Prob} ergibt eine Gewichtung mit den Faktoren $g_1 = 7$ und $g_2 = 3$ auf $0,6\overline{8}$, die z.B. dadurch begründet ist, dass 70% der Zugriffe auf Replikatseinheit RE_1 und 30% auf RE_2 entfallen:

$$KG_{Prob} = \frac{1}{7 \cdot 6 + 3 \cdot 4}(7 \cdot 6 \cdot 0,8 + 3 \cdot 4 \cdot 0,3) = 0,6\overline{8}$$

Um den erwarteten Versionsabstand eines Replikats V_{ErwRep} bzw. den erwarteten Konsistenzgrad eines Replikats KG_{ErwRep} berechnen zu können, der in die Berechnung des erwarteten Versionsabstandes bzw. des erwarteten Konsistenzgrades einer Replikationseinheit KG_{ErwRE} einfließt, werden die Versionsabstandswahrscheinlichkeiten $p_i(R_k^o)$ (siehe Definition 44 auf Seite 97) benötigt. Ein Sonderfall bildet eine Replikationseinheiten RE_j, deren Replikate jederzeit synchron aktualisiert werden und somit immer aktuell sind. In diesem Fall ist die Versionsabstandswahrscheinlichkeit $p_0(R_k^o) = 1$ und $p_i(R_k^o) = 0, i \geq 1$. Damit ist $V_{ErwRep}(R) = 0$ und $KG_{ErwRep}(R) = 1, R \in RE_j$ und damit $V_{ErwRE}(RE_j) = 0$, $KG_{ErwRE}(RE_j) = 1$. Dieser erwartete Versionsabstand bzw. erwartete Konsistenzgrad der Replikationseinheit RE_j muss natürlich bei der Berechnung des probabilistischen Versionsabstandes V_{Prob} bzw. des probabilistischen Konsistenzgrades KG_{Prob} berücksichtigt werden.

Wenn ein Replikat veralten kann, dann sind die Versionsabstandswahrscheinlichkeiten des Replikats größer als null, zumindest bis zu einem Versionsabstand von n, d.h. im Allgemeinen gilt: $p_i(R_k^o) > 0, 0 \leq i \leq n$. Bei der Replikationsstrategie RegRess hängt die Veralterung von Replikaten einerseits von den spezifizierten Regeln und andererseits von Systemkennzahlen ab, d.h. von Schreibraten und Bearbeitungsraten. Letzteres wird häufig durch so genannte Geburts- und Todesprozesse der Stochastik modelliert (auch Irrfahrt mit bzw. ohne Pause; engl.: random walk, birth death processes; siehe z.B. [Tri01]). Weil bei RegRess zusätzlich die Regeln ein unterschiedliches Konsistenzverhalten für verschiedene Replikate ermöglicht, kann kein einheitliches Modell für alle Replikate aufgestellt werden. Somit gibt es keine einheitliche Lösung für alle Replikate, z.B. in Form einer parametrisierten Formel.

Daher wird im Folgenden ein allgemeines, stochastisches Modell in Form einer zeitstetigen Markov-Kette (auch Markov-Prozess [SW04b]) vorgestellt, mittels dem die Versionsabstandswahrscheinlichkeiten $p_i(R_k^o)$ des Replikats R_k^o bestimmt werden können. Je nach spezifizierten Regeln, die das Replikat R_k^o betreffen, und den Schreib- und Bearbeitungsraten muss diese zeitstetige Markov-Kette auf das konkrete Replikat R_k^o angepasst werden, um dann die stationäre Verteilung [Tri01] der Markov-Kette zu berechnen. Die stationäre Verteilung der Markov-Kette des Replikats R_k^o stellen im Wesentlichen die Versionsabstandswahrscheinlichkeiten $p_i(R_k^o)$ des Replikats dar.

Die Idee der Verwendung einer Markov-Kette besteht darin, dass ein Zustand Z_i der Markov-Kette einem Versionsabstand von i entspricht. Bei RegRess ist es so, dass zu einem Zeitpunkt t zwei Modi der Verarbeitung unterschieden werden müssen:

1. Eine Abarbeitung von Aufträgen des Replikats R_k^o aus der Replica Queue ist zum Zeitpunkt t nicht aktiv: Weil keine Aufträge des Replikats R_k^o bearbeitet werden, kann der Versionsabstand zum Zeitpunkt t nicht dekrementiert werden, d.h. es gibt keinen Übergang vom Zustand Z_i zum „Vorgängerzustand“ Z_{i-1}. Wenn zum Zeitpunkt t ein Schreibzugriff erfolgt, sodass ein weiterer Auftrag des Replikats R_k^o in die Replica Queue geschrieben wird und sich damit der Versionsabstand erhöht, dann gibt es einen Übergang vom Zustand Z_i zum „Nachfolgezustand“ Z_{i+1}.
2. Eine Abarbeitung von Aufträgen des Replikats R_k^o aus der Replica Queue ist zum Zeitpunkt t aktiv: Weil nun Aufträge des Replikats R_k^o bearbeitet werden, kann der Versionsabstand zum Zeitpunkt t dekrementiert werden, d.h. es gibt einen Übergang vom Zustand Z_i zum „Vorgängerzustand“ Z_{i-1}. Ein Schreibzugriff erhöht wie unter 1.) den Versionsabstand und bedingt einen entsprechenden Zustandsübergang.

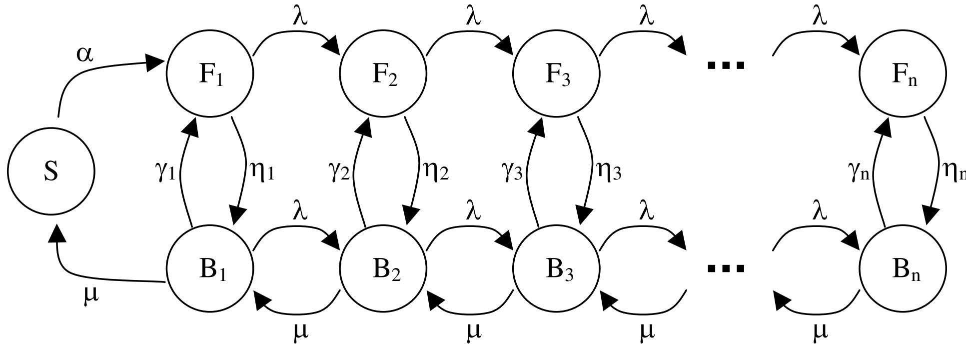

Abbildung 4.5.: Versionsabstandswahrscheinlichkeiten mittels zeitstetiger Markov-Kette

Weil es somit zu einem Zeitpunkt t unterschiedliche Zustandsübergänge gibt, spricht man von einer inhomogenen Markov-Kette [Fis88]. Inhomogene Markov-Ketten lassen sich im Allgemeinen schwieriger lösen als homogene Markov-Ketten, bei denen die Zustandsübergänge zu jedem Zeitpunkt t gleich sind. Nach [SW04b] lassen sich inhomogene Markov-Ketten in homogene Markov-Ketten überführen, indem die Zeitkomponente in die Zustände kodiert wird. In dem hier vorliegenden Fall bedeutet das, dass je Versionsabstand zwei Zustände für die unterschiedlichen Modi unterschieden werden.

In Abbildung 4.5 ist die allgemeine zeitstetige Markov-Kette, die als Vorlage für die Konstruktion einer konkreten zeitstetigen Markov-Kette für ein bestimmtes Replikats dient, als Übergangsgraph dargestellt. Nachfolgend werden sowohl die Zustände als auch die Übergangsraten (exponentielle Wahrscheinlichkeitsdichtefunktion [Fis88]) beschrieben, wobei die Raten in gleichen Zeiteinheiten dimensioniert sind:

S Der Zustand S bildet den Startzustand. Das Replikat R_k^o ist aktuell, d.h. der Versionsabstand ist null. Vom Startzustand S kann ausschließlich zum Zustand F_1 übergegangen werden, d.h. der Übergangsgraph ist so modelliert, dass bei einem Versionsabstand von null die asynchrone Aktualisierung nicht aktiv ist, also Modus 1 ist aktiv (siehe oben). Falls benötigt, kann auch ein entsprechender Übergang mit der Übergangsrate α vom Zustand S zu B_1 modelliert werden, d.h. Modus 2 ist aktiv (siehe oben).

F_i Die Zustände F_i, $i = 1, 2, 3, .., n$ bedeuten einen Versionsabstand von i, wobei Modus 1 aktiv ist (siehe oben), d.h. ein Zustand F_i hat keinen Übergang zu seinem „Vorgängerzustand“ F_{i-1}.

B_i Die Zustände B_i, $i = 1, 2, 3, .., n$ bedeuten ebenfalls einen Versionsabstand von i, wobei Modus 2 aktiv ist (siehe oben), d.h. ein Zustand B_i mit $i > 1$ hat einen Übergang zu seinem „Vorgängerzustand“ B_{i-1} und der Zustand B_1 hat einen Übergang zum Startzustand S.

λ Die Übergangsrate λ ist die Schreibrate auf das Replikat R_k^o. Damit wird der Übergang von einem Versionsabstand i zum Versionsabstand $i + 1$ modelliert, d.h. vom Zustand F_i zum Zustand F_{i+1} bzw. vom Zustand B_i zum Zustand B_{i+1}. Für den Übergang vom Versionsabstand 0 zum Versionsabstand 1, d.h. vom Zustand S zum Zustand F_1, gilt die gesonderte Übergangsrate α.

μ Die Übergangsrate μ ist die Bearbeitungsrate bei der asynchronen Aktualisierung, d.h. die Bearbeitungsrate, mit der Aufträge des Replikats R_k^o aus der Replica Queue bearbeitet werden. Damit wird der Übergang von einem Versionsabstand i zum Versionsabstand $i-1$ modelliert, d.h. vom Zustand B_1 zum Zustand S bzw. vom Zustand B_i zum Zustand B_{i-1} für $i > 1$.

α Die Übergangsrate α modelliert den Übergang vom Versionsabstand 0 zum Versionsabstand 1, d.h. vom Zustand S zum Zustand F_1 (ggf. zu B_1, siehe oben). Wenn ein Replikat grundsätzlich asynchron aktualisiert wird, dann entspricht die Übergangsrate α der Schreibrate λ (siehe oben). Wenn ein Replikat nur nach Eintritt einer bestimmten Bedingung in einer Replikationsregel asynchron aktualisiert wird, dann gilt: $\alpha = c \cdot \lambda$ mit $0 < c < 1$. c gibt die Wahrscheinlichkeit für das Eintreten der Bedingung(en) an, die zur asynchronen Aktualisierung des Replikats R_k^o führen.
Beispiel: Wenn eine Replikationsregel spezifiziert ist, die die asynchrone Aktualisierung des Replikats R_k^o erlaubt, wenn die Komponente K_k nicht verfügbar ist, und die Komponente K_k eine Verfügbarkeit von $0,9$ hat, dann gilt: Nicht-Verfügbarkeit tritt mit der Wahrscheinlichkeit $0,1$ ein. Damit gilt: $\alpha = 0,1 \cdot \lambda$.

η_i Die Übergangsrate η_i modelliert einen Übergang vom Zustand F_i zum Zustand B_i, d.h. der Modus wird von Modus 1 zu Modus 2 gewechselt (siehe oben). Damit wird modelliert, dass die Bearbeitung der Replica Queue begonnen wird, also die asynchrone Aktualisierung des Replikats R_k^o beginnt. Weil die Übergangsrate vom Versionsabstand abhängen kann, kann es je Zustand F_i unterschiedliche Übergangsraten geben (siehe unten).

γ_i Die Übergangsrate γ_i modelliert einen Übergang vom Zustand B_i zum Zustand F_i, d.h. der Modus wird von Modus 2 zu Modus 1 gewechselt (siehe oben). Damit wird modelliert, dass während der Bearbeitung der Replica Queue die asynchrone Aktualisierung des Replikats R_k^o abgebrochen wird. Weil die Übergangsrate vom Versionsabstand abhängen kann, kann es je Zustand B_i unterschiedliche Übergangsraten geben (siehe unten).

Bevor auf die Anzahl der benötigten Zustände für ein konkretes Replikat R_k^o und die Übergangsraten, insbesondere die Übergangsraten η_i und γ_i, eingegangen wird, sollen die Versionsabstandswahrscheinlichkeiten $p_i(R_k^o)$ bestimmt werden. Die in Abbildung 4.5 gezeigte Markov-Kette ist irreduzibel und aperiodisch, sodass es genau eine *„stationäre Verteilung“* gibt [Tri01]. Die stationäre Verteilung einer Markov-Kette gibt die Wahrscheinlichkeiten dafür an, sich in den entsprechenden Zuständen der Markov-Kette zu befinden. Für die Berechnung der stationären Verteilung sei auf die entsprechende Literatur verwiesen (siehe z.B. [SW04b]). Liegt die stationäre Verteilung vor, dann können die hier benötigten Versionsabstandswahrscheinlichkeiten $p_i(R_k^o)$ berechnet werden. Seien s^*, f_i^*, b_i^*, $i = 1, 2, 3, .., n$ die Wahrscheinlichkeiten der stationären Verteilung sich im Zustand S, F_i bzw. B_i zu befinden, dann gilt:

$$p_i(R_k^o) = \begin{cases} s^* & i = 0 \\ f_i^* + b_i^* & 1 \leq i \leq n \\ 0 & i > n \end{cases} \tag{4.16}$$

In Abbildung 4.6 ist als Beispiel eine zeitstetige Markov-Kette abgebildet, die auf folgenden Daten beruht: Für alle Replikate gilt eine Kohärenzbedingung, die einen Versionsabstand von maximal vier erlaubt. Wenn der Versionsabstand gleich vier ist, werden synchrone Schreibzugriffe abgewiesen. Die asynchrone Aktualisierung beginnt bei einem Versionsabstand von vier. Die zugehörigen Replikationsregeln sind im Listing 7.3 auf Seite 202 dargestellt. Somit enthält die Markov-Kette einen Startzustand und jeweils vier Zustände F_i bzw. B_i mit $1 \leq i \leq 4$.

Weiterhin wird angenommen, dass in der gegebenen Zeiteinheit fünf Schreibzugriffe erfolgen, d.h. $\lambda = 5$. Weil die Replikate grundsätzlich asynchron aktualisiert werden, gilt: $\alpha = \lambda = 5$. Die Bearbeitungsrate für Aufträge aus der Replica Queue liegt um den Faktor 20 höher als die Schreibrate, sodass $\mu = 100$ gilt. Weil ein Versionsabstand kleiner als vier toleriert wird, wird nur im Zustand F_4 die asynchrone Aktualisierung begonnen. Deswegen existiert nur ein Übergang vom Zustand F_4 zu B_4, wobei angenommen wird, dass das Umschalten von Modus 1 nach Modus 2 (siehe oben) zehn mal schneller erfolgt, als die Bearbeitung von Aufträgen der

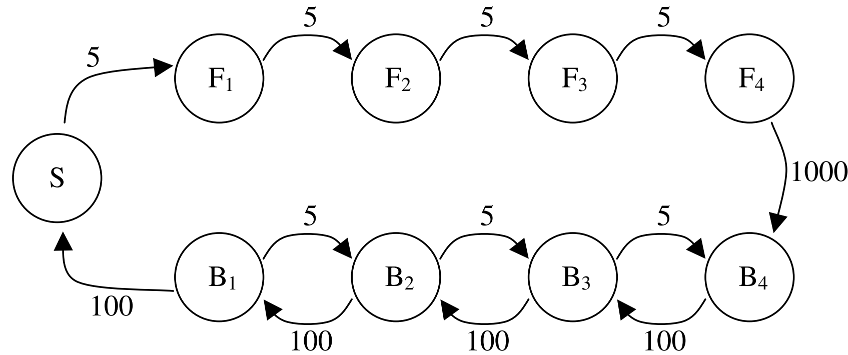

Abbildung 4.6.: Zeitstetige Markov-Kette für Versionsabstand von 4

Replica Queue, d.h. $\eta_1 = \eta_2 = \eta_3 = 0$ und $\eta_4 = 1000$. Der Einfachheit halber wird angenommen, dass keine Transaktionsfehler, die z.B. durch Nicht-Verfügbarkeit bedingt sein könnten, während der asynchronen Aktualisierung auftreten. Damit gilt: $\gamma_i = 0$ für $1 \leq i \leq 4$.

Für die Berechnung der stationären Verteilung wurde das Werkzeug PRISM verwendet, eine Open Source Software zur Analyse stochastischer Modelle (www.prismmodelchecker.org). Grundsätzlich kann jedes Werkzeug genutzt werden, dass zeitstetige Markov-Ketten (Markov-Prozesse) auswerten kann. Folgende Versionsabstandswahrscheinlichkeiten wurden gemäß Gleichung 4.16 ermittelt:

p_0	$0,242$
p_1	$0,248$
p_2	$0,248$
p_3	$0,248$
p_4	$0,014$

Mit den Versionsabstandswahrscheinlichkeiten kann der erwartete Versionsabstand bzw. der erwartete Konsistenzgrad eines Replikats berechnet werden. Weil alle Replikate die gleichen Erwartungswerte haben, entsprechen diese Erwartungswerte den Erwartungswerten der replizierten Datenbank:

$$V_{ErwRep}(R_k^o) = V_{Prob} = 1,488 \tag{4.17}$$

$$KG_{ErwRep}(R_k^o) = KG_{Prob} = 0,511 \tag{4.18}$$

Weil die Zustände der hier vorgestellten Markov-Kette die Versionsabstände widerspiegeln, lassen sich Replikate, die auf Grund von Replikationsregeln mit Versionsabstandsbedingungen veralten dürfen, besonders einfach modellieren. Bei anderen Replikationsregeln ist die Konstruktion der Markov-Kette weitaus schwieriger. Wenn einem Replikat z.B. ein Zeitverzug gestattet ist, dann muss zunächst die Anzahl Zustände bestimmt bzw. abgeschätzt werden, weil nicht exakt gesagt werden kann, wie viele Versionen mittlerweile geschrieben wurden. Damit lässt sich auch nicht exakt bestimmen, bei welchem Zustand F_i die asynchrone Aktualisierung beginnt, d.h. für die Übergangsraten η_i muss eine Abschätzung vorgenommen werden, z.B. in Form einer Wahrscheinlichkeitsverteilung. Im Wesentlichen gilt gleiches für die Übergangsraten γ_i, wobei hier eher auf technische Konsistenzbedingungen eingegangen werden muss.

In dieser Dissertation liegt der Fokus auf den simulierten Konsistenzgrad, weil die Replikationsstrategie RegRess bzw. der Replikationsmanager KARMA, der RegRess implementiert, mittels Simulation evaluiert wird (siehe Abschnitt 7.3). Neben dem Konsistenzgrad bietet die Simulation auch weitere Kennzahlen zur Analyse. Daher wird hier der probabilistische Versionsabstand

bzw. der probabilistische Konsistenzgrad, insbesondere die Konstruktion der entsprechenden Markov-Ketten, nicht weiter vertieft. Hierfür sei auf die einschlägige Literatur verwiesen, z.B. [Fis88, Tri01, SW04b].

4.4. Trade-Off der Replikationsziele

In Abbildung 3.1 auf Seite 28 sind die Ziele der Replikation, also Verfügbarkeit, Performance und Konsistenz, als Zielkonflikte dargestellt. Ein Zielkonflikt liegt deshalb vor, weil es sich um konkurrierende Ziele [Dör03] handelt (auch konträre oder gegenläufige Ziele genannt). Ziele sind genau dann konkurrierend, wenn die Verbesserung des einen Ziels eine Verschlechterung eines anderen Ziels nach sich zieht. Mit *„Trade-Off“* ist hier das Abwägen von konkurrierenden Zielen gemeint, d.h. es soll aus verschiedenen Lösungsalternativen ein optimaler Kompromiss hinsichtlich der vorgegebenen Ziele ausgewählt werden. Hierbei handelt es sich um ein Entscheidungsproblem, das in der Entscheidungstheorie durch *„rationales Entscheiden“* [EW02] gelöst wird.

Bei RegRess handelt es sich um eine Replikationsstrategie, bei der die Koordination der Schreib- und Lesezugriffe durch Inferenz von Regeln erfolgt (siehe Abschnitt 4.2). In den Regeln können neben Konsistenzbedingungen auch Reaktionen auf Nicht-Verfügbarkeit und/oder Performanceeigenschaften, wie z.B. Antwortzeiten, berücksichtigt werden. Somit kann mit verschiedenen Regelmengen ein unterschiedliches Verhalten von RegRess hinsichtlich der Ziele Verfügbarkeit, Performance und Konsistenz erreicht werden. Die Bestimmung einer aus Sicht des Anwenders optimalen Regelmenge zur Lösung des Zielkonflikts soll in diesem Abschnitt diskutiert werden.

In Abschnitt 4.4.1 wird zunächst auf rationales Entscheiden bei der Replikation eingegangen. Anschließend werden in Abschnitt 4.4.2 die Replikationsziele Verfügbarkeit, Performance und Konsistenz speziell der Replikationsstrategie RegRess betrachtet, d.h. es wird gezeigt, wie in Abhängigkeit von Regeln die entsprechenden Kennzahlen ermittelt bzw. berechnet werden. Ein rationales Entscheiden an Hand dieser Kennzahlen erfolgt mittels einer multiattributiven Wertfunktion, die in Abschnitt 4.4.3 betrachtet wird.

4.4.1. Rationales Entscheiden bei Replikation

Um ein Entscheidungsproblem zu lösen, müssen Ziele und Alternativen vorliegen. Dabei wird jede Alternative hinsichtlich der Ziele gemessen bzw. bewertet. Bei mehreren Zielen, die konkurrierend sind, wird eine Wertfunktion benötigt, die den Vergleich der Alternativen bezüglich der Zielerreichung erlaubt. In [EW02] wird die Gesamtheit der Ziele als Zielsystem bezeichnet, wobei die wichtigsten Anforderungen an ein Zielsystem wie folgt genannt werden:

- Vollständigkeit: Alle wesentlichen Aspekte, die die Entscheidung beeinflussen, müssen in den Zielen berücksichtigt sein.
- Redundanzfreiheit: Ziele sollten sich in ihrer Bedeutung nicht überschneiden.
- Messbarkeit: Die Zielerreichung sollte möglichst treffend und eindeutig messbar sein.

In Abschnitt 3.1 wurden Ziele der Replikation diskutiert. Häufig werden in der Literatur als wichtige Ziele die Verfügbarkeit, Performance und Konsistenz (siehe z.B. [Has97]) genannt, die auch in dieser Dissertation verwendet werden. Im Allgemeinen sind bei Replikationsstrategien unterschiedliche Mengen von Replikaten bei Schreib- bzw. Lesezugriffen betroffen. Bei der Replikationsstrategie ROWA (siehe Abschnitt 3.3.1) erfolgt beispielsweise ein Schreibzugriff auf alle Replikate des logischen Objekts, sodass die Performance, z.B. in Form von Antwortzeit, gegenüber einem Lesezugriff, bei dem nur ein Replikat des logischen Objekts gelesen werden muss, im Allgemeinen höher ist. Daraus folgt, dass die Ziele Verfügbarkeit, Performance und Konsistenz hinsichtlich Schreib- und Lesezugriffen unterschieden werden sollten.

Andererseits wird für das Zielsystem möglichst Redundanzfreiheit gefordert. Insbesondere bei den synchronen Replikationsstrategien werden aber Schreib- und Lesezugriffe derart aufeinander abgestimmt, dass die 1-Kopien-Serialisierbarkeit erreicht wird. Bei ROWA bedeutet das, dass das Ziel, hohe Konsistenz bei Schreibzugriffen, sich mit dem Ziel, kurze Antwortzeiten bei Lesezugriffen, überschneidet. Ein weiteres Beispiel ist es, dass bei den asynchronen Replikationsstrategien die abgeschwächte Konsistenz (siehe Abschnitt 3.2.3) sowohl für Schreib- als auch für Lesezugriffe gilt. In dieser Dissertation wird daher bei den Zielen nicht nach der Zugriffsart unterschieden, sondern es werden Schreibzugriffe angenommen.

Die Messbarkeit als weitere Anforderung an das Zielsystem ist für Verfügbarkeit und Performance durch verschiedene Metriken in der Literatur gegeben (siehe Abschnitt 4.4.2). Konsistenz ist dagegen im Allgemeinen schwerer messbar. Weil hier aber unter Konsistenz das Verhältnis der Replikate eines logischen Objekts zueinander verstanden werden soll, können entsprechende Metriken für abgeschwächte Konsistenz in replizierten Datenbanken verwendet werden (siehe Abschnitt 3.2.3). Dabei kommen eher die Metriken in Betracht, die die Gesamtheit der Replikate bewerten, wie z.B. die Frische einer replizierten Datenbank (siehe Definition 26 auf Seite 38), die bemerkte Frische (siehe Definition 27 auf Seite 39) oder der in dieser Dissertation verwendete Konsistenzgrad (siehe Abschnitt 4.3.3).

Die Alternativen, die beim rationalen Entscheiden bei Replikation in Erwägung kommen, können entweder aus unterschiedlichen Replikationsstrategien bestehen, die verglichen werden sollen, und/oder aus unterschiedlichen Konfigurationen einer Replikationsstrategie, um eine optimale Voreinstellung zu bestimmen. Bei der Replikationsstrategie Weigthed Voting (siehe Abschnitt 3.3.1) könnten z.B. unterschiedliche Quorengröße und Gewichte untersucht werden. Bei der hier vorgestellten Replikationsstrategie RegRess wird das Verhalten durch die spezifizierten Regeln bestimmt. Somit können bei RegRess unterschiedliche Regelmengen untersucht werden, um für ein Anwendungsszenario die optimale Regelmenge auszuwählen.

Zur Bestimmung der optimalen Alternative ist eine Bewertung der Zielerreichung nötig. Weil bei der Replikation hinsichtlich mehrerer Ziele optimiert werden muss, spricht man von einem mehrdimensionalen Optimierungsproblem. Daher wird eine so genannte Wertfunktion für mehrere Attribute, eine multiattributive Wertfunktion [EW02], benötigt. Hierauf wird in Abschnitt 4.4.3 eingegangen.

In dieser Dissertation wird angenommen, dass die Konsequenzen bei jeder Alternative feststehen, d.h. es handelt sich hier um eine *„Entscheidung unter Sicherheit“* [Dör03]. Damit ist gemeint, dass dann, wenn die „Umwelteinflüsse“ bekannt sind, die Zielerreichung einer Alternative eindeutig ermittelt werden kann. Wenn also z.B. die Verfügbarkeit einer jeden Komponente mit Replikat bekannt ist, dann kann auch die Verfügbarkeit beim Einsatz einer Replikationsstrategie ermittelt werden. Würde beispielsweise die Verfügbarkeit einer Komponente mit Replikat auf Grund irgendwelcher Umwelteinflüsse schwanken, so müsste eine Wahrscheinlichkeitsdichtefunktion angesetzt werden und man spricht von *„Entscheidung unter Unsicherheit“*.

4.4.2. Verfügbarkeit, Performance und Konsistenz bei RegRess

In diesem Abschnitt werden Metriken für die Ziele der Replikation vorgestellt sowie die Bestimmung der Kennzahlen in den jeweiligen Maßeinheiten erläutert. Dabei gelten die folgenden Beschreibungen grundsätzlich für alle Replikationsstrategien, jedoch soll hier auf die in dieser Dissertation entwickelte Replikationsstrategie RegRess fokussiert werden. Dadurch können unterschiedliche Regelmengen, die die Konfiguration für RegRess bilden, verglichen werden. Daher wird nachfolgend gezeigt, wie an Hand von Regeln die Verfügbarkeit, Performance und Konsistenz ermittelt werden kann, wobei auf Schreibzugriffe abgestellt wird (siehe Abschnitt 4.4.1).

Für die Konsistenz soll an dieser Stelle die Metrik Konsistenzgrad verwendet werden, die ausführlich in Abschnitt 4.3.3 erläutert wird. Wie beschrieben, kann der Konsistenzgrad gemessen, durch Simulation bestimmt oder analytisch errechnet werden. Das Gleiche gilt für Verfügbarkeit und Performance. Der Simulator F4SR (siehe Abschnitt 7.3.1) bietet verschiedene Auswer-

tungen, die nach einem Simulationslauf z.B. in Diagrammen angezeigt werden. In Abbildung 7.13 auf Seite 204 wird das Verhältnis von erfolgreichen Schreibzugriffen zu abgewiesenen Schreibzugriffen dargestellt, das für die Verfügbarkeit von Schreibzugriffen verwendet werden kann. Allerdings sollte bei abgewiesenen Schreibzugriffen zwischen Ausfall und anderer Fehler unterschieden werden (siehe unten). Die Antwortzeiten, die als Metrik für die Performance genutzt werden können, sind in Abbildung 7.13 auf Seite 204 dargestellt. Das KARMA-Plugin (siehe Abschnitt 7.3.3) zeigt zusätzlich den Konsistenzgrad an. Deswegen wird in diesem Abschnitt nachfolgend auf die analytische Bestimmung der Verfügbarkeit und Performance abgestellt.

Verfügbarkeit von Schreibzugriffen bei RegRess

Zunächst wird Verfügbarkeit allgemein definiert und eine Metrik inklusive Berechnungsvorschrift angegeben:

Definition 53 (nach [Bör03]) ***Verfuegbarkeit**: Verfügbarkeit, genauer die Punkt-Verfügbarkeit, ist als Wahrscheinlichkeit definiert, dass ein System bei gegebenen Arbeitsbedingungen und zu einem bestimmten Zeitpunkt die geforderte Funktion ausführt. Die Verfügbarkeit wird wie folgt berechnet:*

$$V = \frac{MTTF}{MTTF + MTTR}$$

mit MTTF (mean time to failure): Zeitmittelwert zwischen zwei Fehlern, MTTR (mean time to repair): Zeitmittelwert für die Reparaturzeit.

In Definition 53 wird allgemein von einer Funktion (auch Dienst) gesprochen. Hier handelt es sich um einen Schreibzugriff. Dabei wird die Verfügbarkeit aus Sicht eines Clients bewertet (siehe Abbildung 4.1), d.h. es wird ein Schreibzugriff auf ein logisches Objekt betrachtet, den ein Client an den Replikationsmanager übergibt. Der Dienst, also der Schreibzugriff, gilt als erbracht, wenn die Spezifikation erfüllt ist. Hiermit ist gemeint, dass der Replikationsmanager die betroffenen Replikate des logischen Objekts „erreichen“ konnte. Transaktionsfehler, z.B. auf Grund von Verletzung der Integritätsbedingungen (siehe Abschnitt 2.3.2), betreffen nicht die Verfügbarkeit. Somit werden an dieser Stelle so genannte Rechnerausfälle [Bör03] betrachtet bzw. Ausfall der Komponente mit Replikat.

Bei RegRess erfolgt die Koordination eines Schreibzugriffs dadurch, dass die Replikate in synchron und asynchron zu aktualisierende Replikate partitioniert werden (siehe Abschnitt 4.2.2). Aus Sicht des Clients betreffen die asynchron zu aktualisierenden Replikate die Verfügbarkeit nur insofern, als dass der Replikationsmanager, insbesondere die Replica Queue, verfügbar ist, während die synchron zu aktualisierenden Replikate, genauer gesagt deren Komponenten mit Replikat, verfügbar sein müssen. Die Partitionierung wird wiederum durch Regeln vorgegeben, sodass sich bei RegRess je logischem Objekt bzw. je Replikationseinheit eine unterschiedliche Verfügbarkeit ergeben kann.

Der Einfachheit halber wird nachfolgend angenommen, das der Replikationsmanager immer verfügbar ist, also $V_{Replikationsmanager} = 1$. Weiterhin sei V_k die Verfügbarkeit der k-ten Komponente mit Replikat. Außerdem wird vorausgesetzt, dass Ausfälle der Komponenten mit Replikat unabhängig voneinander sind. Damit kann z.B. die Verfügbarkeit für einen Schreibzugriff bei der Replikationsstrategie ROWA (siehe Abschnitt 3.3.1) durch Multiplikation der jeweiligen Verfügbarkeit einer Komponente mit Replikat ausgedrückt werden [Fis88]:

$$V_{ROWA} = \prod_{k=1}^{n} V_k \tag{4.19}$$

Im Folgenden wird analog zum Konsistenzgrad, bei dem mittels des Konsistenzgrades eines Replikats der Konsistenzgrad einer replizierten Datenbank errechnet wird, die Verfügbarkeit der replizierten Datenbank definiert. Somit wird die Verfügbarkeit eines Replikats aus Sicht eines

Clients benötigt. Dabei wird unterschieden, ob ein Replikat R_k^o synchron oder asynchron aktualisiert wird. Wenn R_k^o synchron aktualisiert wird, dann wird für die Verfügbarkeit des Replikats die Verfügbarkeit der Komponente mit Replikat, also V_k angesetzt. Wenn das Replikat asynchron aktualisiert wird, dann wird die Verfügbarkeit nicht benötigt bzw. sie wird auf eins gesetzt, d.h. aus Sicht eines Clients ist ein asynchron zu aktualisierendes Replikat immer verfügbar.

Bei RegRess kann durch Regeln festgelegt werden, dass ein Replikat grundsätzlich synchron oder grundsätzlich asynchron aktualisiert wird. Ein Replikat, das synchron aktualisiert werden soll und wechselfähig ist, zählt aus Sicht der Verfügbarkeit zu den asynchronen Replikaten, weil es bei Nicht-Verfügbarkeit asynchron aktualisiert wird. Daneben kann es aber auch Replikate geben, die in Abhängigkeit einer Bedingung entweder synchron oder asynchron aktualisiert werden, z.B. in Abhängigkeit der Antwortzeit oder eines Versionsabstandes. Für derartige Replikate soll die Verfügbarkeit zwischen V_k und 1 liegen, wobei mit der Wahrscheinlichkeit für die unterschiedlichen Aktualisierungsarten gewichtet werden soll. Dadurch ist folgende Definition motiviert:

Definition 54 ***Wahrscheinlichkeit für die synchrone Aktualisierung eines Replikats:*** *$q(R_k^o)$ ist die Wahrscheinlichkeit dafür, dass das Replikat R_k^o synchron aktualisiert wird. Für ein grundsätzlich synchron zu aktualisierendes Replikat gilt: $q(R_k^o) = 1$. Für ein wechselfähiges oder grundsätzlich asynchron zu aktualisierendes Replikat gilt: $q(R_k^o) = 0$.*

Wenn die Aktualisierungsart des Replikats R_k^o von Bedingungen abhängt, dann muss $q(R_k^o)$ in Definition 54 entsprechend bestimmt werden. Wenn z.B. die Bedingung die Antwortzeit im Sinne einer Überlastung der Komponente mit Replikat beinhaltet, dann ist die Wahrscheinlichkeit für eine Überlastung der Komponente mit Replikat für $q(R_k^o)$ anzusetzen. Mit $q(R_k^o)$ kann nun die Verfügbarkeit eines Replikats definiert werden:

Definition 55 ***Verfügbarkeit eines Replikats:*** *Die Verfügbarkeit $V(R_k^o)$ eines Replikats R_k^o ergibt sich als gewichtetes Mittel aus der Verfügbarkeit V_k der Komponente K_k, die das Replikat R_k^o speichert, und dem Wert 1, wobei das Gewicht $q(R_k^o)$ der Wahrscheinlichkeit für synchrone Aktualisierung entspricht:*

$$V(R_k^o) = q(R_k^o) \cdot V_k + (1 - q(R_k^o)) \cdot 1 = 1 + q(R_k^o) \cdot V_k - q(R_k^o)$$

Zur Verdeutlichung der Definition 55 sei folgende Ungleichung angegeben:

$$0 \leq V_k \leq V(R_k^o) \leq 1 \tag{4.20}$$

Für die beiden Spezialfälle, d.h. das Replikat R_k^o wird grundsätzlich synchron bzw. asynchron aktualisiert, gilt:

$$V(R_k^o) = \begin{cases} V_k & q(R_k^o) = 1, \text{ synchrone Aktualisierung} \\ 1 & q(R_k^o) = 0, \text{ asynchrone Aktualisierung} \end{cases} \tag{4.21}$$

Mit der Verfügbarkeit der einzelnen Replikate eines logischen Objekts wird nun die Verfügbarkeit eines Schreibzugriffs auf das logische Objekt definiert, wobei Unabhängigkeit der Wahrscheinlichkeiten angenommen wird, wie oben vorausgesetzt:

Definition 56 ***Verfügbarkeit eines logischen Objekts:*** *Die Verfügbarkeit $V(O^o)$ eines logischen Objekts O^o berechnet sich durch Multiplikation der Verfügbarkeiten $V(R_k^o)$ der Replikate des logischen Objekts::*

$$V(O^o) = \prod_{k=1}^{n} V(R_k^o)$$

In Definition 56 werden also analog zur Replikationsstrategie ROWA die Verfügbarkeiten der betroffenen Replikate multipliziert (siehe Gleichung 4.19 auf Seite 106). Weil hier einige Faktoren aber den Wert 1 annehmen können, gilt mit Gleichung 4.19 und 4.20:

$$V(O^o) = \prod_{k=1}^{n} V(R_k^o) \geq \prod_{k=1}^{n} V_k = V_{ROWA} \tag{4.22}$$

Auf Basis der Verfügbarkeiten der logischen Objekte folgt nun die Definition für die Verfügbarkeit der replizierten Datenbank:

Definition 57 ***Verfügbarkeit einer replizierten Datenbank****: Die Verfügbarkeit $V(DB)$ einer replizierten Datenbank berechnet sich durch Mittelung der Verfügbarkeiten $V(O^o)$ der m logischen Objekte:*

$$V(DB) = \frac{1}{m} \sum_{o=1}^{m} V(O^o)$$

Um die Verfügbarkeit $V(DB)$ einer replizierten Datenbank gemäß Definition 57 zu berechnen, müssen nicht zwingend alle logischen Objekte einzeln ermittelt werden, sondern es kann analog zum Konsistenzgrad KG_{Prob} über Replikationseinheiten vereinfacht werden, ggf. können auch hier Gewichte verwendet werden. Wenn durch die Regeln nur die beiden Spezialfälle auftreten (siehe Gleichung 4.21 auf Seite 107), vereinfacht sich die Berechnung ebenfalls erheblich, weil dann nur die Komponenten berücksichtigt werden müssen, die stets synchron aktualisiert werden.

Performance von Schreibzugriffen bei RegRess

Als Metrik für die Performance wird im Allgemeinen Antwortzeit, Verweildauer, Durchsatz, Jobs pro Zeiteinheit oder ähnliches gewählt. In dieser Dissertation wird die Antwortzeit als Metrik für die Performance verwendet, die hier wie folgt definiert ist:

Definition 58 (nach [MA02]) ***Antwortzeit****: Die Antwortzeit ist das Zeitintervall vom Senden einer Nachricht bis zum Empfang einer Antwort. Die Antwortzeit besteht aus der Übertragungszeit, falls eine Netzkommunikation erforderlich ist, aus der Bearbeitungszeit und aus der Wartezeit.*

In Definition 58 ist die Antwortzeit für einen Dienstaufruf, hier für einen Schreibzugriff, definiert. Für mehrere Schreibzugriffe variiert die Antwortzeit im Allgemeinen auch dann, wenn die gleichen Komponenten mit Replikat involviert sind. Daher werden im Allgemeinen mittlere Antwortzeiten angeben. In analoger Weise zur Verfügbarkeit (siehe oben) wird anschließend die mittlere Antwortzeit für ein Replikat, für ein logisches Objekt und für die replizierte Datenbank definiert. Dabei sei angenommen, dass A_k die mittlere Antwortzeit der k-ten Komponente mit Replikat ist und dass A_{RQ} die Antwortzeit der Replica Queue ist, d.h. die Zeit, die benötigt wird, um einen Auftrag in die Replica Queue zu schreiben. Im Allgemeinen ist A_{RQ} kleiner als A_k mit $k = 1, 2, ..., n$, weil es sich bei den Zugriffen auf die Komponenten mit Replikat um entfernte Zugriffe handelt.

Auch bei der mittleren Antwortzeit eines Replikats muss unterschieden werden, ob das Replikat synchron oder asynchron aktualisiert wird bzw. ob es Bedingungen gibt, die einen zeitweisen Wechsel der Aktualisierungsart bewirken. Daher wird auch hier die Wahrscheinlichkeit für die synchrone Aktualisierung eines Replikats benötigt (siehe Definition 54 auf Seite 107), um die mittlere Antwortzeit eines Replikats zu definieren:

Definition 59 ***Mittlere Antwortzeit eines Replikats***: *Die mittlere Antwortzeit $A(R_k^o)$ eines Replikats R_k^o ergibt sich als gewichtetes Mittel aus der mittleren Antwortzeit A_k der Komponente K_k, die das Replikat R_k^o speichert, und der Antwortzeit A_{RQ} der Replica Queue, wobei das Gewicht $q(R_k^o)$ der Wahrscheinlichkeit für synchrone Aktualisierung entspricht:*

$$A(R_k^o) = q(R_k^o) \cdot A_k + (1 - q(R_k^o)) \cdot A_{RQ}$$

Für die beiden Spezialfälle, d.h. das Replikat R_k^o wird grundsätzlich synchron bzw. asynchron aktualisiert, gilt:

$$A(R_k^o) = \begin{cases} A_k & q(R_k^o) = 1, \text{ synchrone Aktualisierung} \\ A_{RQ} & q(R_k^o) = 0, \text{ asynchrone Aktualisierung} \end{cases} \tag{4.23}$$

Mit den mittleren Antwortzeiten der einzelnen Replikate eines logischen Objekts wird nun die mittlere Antwortzeit eines Schreibzugriffs auf das logische Objekt definiert. Dabei sei angenommen, dass alle Schreibzugriffe auf die Replikate sowie das Anhängen der Aufträge in die Replica Queue seriell nacheinander ausgeführt werden:

Definition 60 ***Mittlere Antwortzeit eines logischen Objekts***: *Die mittlere Antwortzeit $A(O^o)$ eines logischen Objekts O^o berechnet sich bei serieller Ausführung der Teiloperationen durch Addition der mittleren Antwortzeiten $A(R_k^o)$ der Replikate des logischen Objekts:*

$$A(O^o) = \sum_{k=1}^{n} A(R_k^o)$$

Bei der Berechnung der mittleren Antwortzeit $A(O^o)$ in Definition 60 ist es wichtig, dass die Teiloperationen, d.h. insbesondere die Schreibzugriffe auf die Replikate, seriell nacheinander durchgeführt werden. Ein Replikationsmanager kann eine kleinere Antwortzeit erreichen, wenn die Schreibzugriffe auf die Replikate parallel durchgeführt werden. In diesem Fall ist in der Definition 60 anstatt der Summe das Maximum der mittleren Antwortzeiten der Replikate des logischen Objekts zu bilden. Mit den mittleren Antwortzeiten der logischen Objekte folgt nun die Definition für die mittlere Antwortzeit der replizierten Datenbank:

Definition 61 ***Mittlere Antwortzeit einer replizierten Datenbank***: *Die mittlere Antwortzeit $A(DB)$ einer replizierten Datenbank berechnet sich durch Mittelung der mittleren Antwortzeiten $A(O^o)$ der m logischen Objekte:*

$$A(DB) = \frac{1}{m} \sum_{o=1}^{m} A(O^o)$$

Auch hier kann wie bei der Verfügbarkeit der replizierten Datenbank die Berechnung der mittleren Antwortzeit der replizierten Datenbank vereinfacht werden, wenn die Berechnung über Replikationseinheiten erfolgt.

4.4.3. Multiattributive Wertfunktion

Damit die Ergebnisse, die eine Alternative hinsichtlich der gegebenen Ziele erreicht, bewertet werden können, müssen die Präferenzen des Entscheiders bekannt sein. In der Entscheidungstheorie werden die Ziele, wie auch in diesem Abschnitt, als Attribute bezeichnet [EW02]. Bei der Replikation, insbesondere bei der Replikationsstrategie RegRess, müssen also für eine Alternative, z.B. für eine bestimmte Regelmenge, die Attribute Verfügbarkeit, Performance und Konsistenz bewertet werden, d.h. es wird eine Funktion benötigt, die die Attribute auf eine Zahl abbildet.

Gemäß Abschnitt 4.4.2 wird Verfügbarkeit als Wahrscheinlichkeit (Bereich [0..1], dimensionslos) gemessen, die Performance als Antwortzeit in Millisekunden und die Konsistenz als Konsistenzgrad, wobei es sich um einen normierten Wert handelt (Bereich [0..1], dimensionslos). Ein gleichzeitiger Vergleich der drei Attribute ist dadurch erschwert, dass unterschiedliche Dimensionen beteiligt sind, z.B. Wahrscheinlichkeiten und Millisekunden. Zwar haben Verfügbarkeit und Konsistenz, genauer gesagt der Konsistenzgrad, den gleichen Wertebereich, aber im Allgemeinen wird der Wertebereich unterschiedlich abgedeckt. So liegen die Verfügbarkeiten heutiger Systeme im Allgemeinen nahe bei 1.

Demzufolge wird eine so genannte *„Wertfunktion“* [EW02] benötigt, die die Attributwerte einer Alternative auf eine reelle Zahl abbildet. Dabei wird im Fall von mehreren Attributen von einer *„multiattributiven Wertfunktion“* gesprochen. Die multiattributive Wertfunktion muss die Präferenzen des Entscheiders widerspiegeln, insbesondere muss Transitivität gelten.

Um die multiattributive Wertfunktion zu bestimmen, wird häufig in zwei Schritten vorgegangen, wie z.B. in [RLPT97], wo die Einbeziehung von Nutzerinteressen bei der Dienstvermittlung unter CORBA diskutiert wird. Zunächst werden jeweils Wertfunktionen für ein Attribut bestimmt. Die einzelnen Wertfunktionen werden zu einer multiattributiven Wertfunktion verknüpft.

Eine Wertfunktion für ein Attribut kann z.B. nach der *„Direct-Rating-Methode“*, nach der *„Methode gleicher Wertdifferenzen“* oder nach der *„Halbierungsmethode“* ermittelt werden (siehe z.B. [EW02]). Eine Wertfunktion ist wegen der Transitivität eine monotone Funktion, die bei den genannten Methoden den Wertebereich [0..1] hat. Weil sie die Präferenzen des Entscheiders berücksichtigt, kann sie unterschiedlichste Ausprägungen haben. Bei der Antwortzeit ist die Wertfunktion z.B. monoton fallend, d.h. mit wachsender Antwortzeit liefert die Wertfunktion kleinere Werte. Die Wertfunktion kann dabei z.B. eine Hyperbelform haben, wodurch bei kleinen Antwortzeiten eine Änderung stärker ins Gewicht fällt als bei großen Antwortzeiten.

Die Verknüpfung der Wertfunktionen einzelner Attribute zur multiattributiven Wertfunktion erfolgt häufig durch gewichtete Addition. Die Ermittlung der Gewichte erfolgt z.B. nach dem *„Trade-Off-Verfahren“*, nach dem *„Swing-Verfahren“* oder nach dem *„Direct-Ratio-Verfahren“* (siehe z.B. [EW02]). Auch hier spielen natürlich die Präferenzen des Entscheiders eine gewichtige Rolle.

Für den Vergleich von Replikationsstrategien und/oder Konfigurationen von Replikationsstrategien, z.B. für den Vergleich unterschiedlicher Regelmengen der hier vorgestellten Replikationsstrategie RegRess, bedeutet das, dass präferenzbasierte Wertfunktionen für Verfügbarkeit, Performance und Konsistenz und entsprechende Gewichte ermittelt werden müssen. Dabei fließen neben den Präferenzen des Entscheiders vor allem Systemkennzahlen wie Antwortzeiten, Schreib-/Leseraten etc. ein. Eine Entscheidung für eine Alternative hängt somit von der konkreten Systemlandschaft ab. Daher wird hier auf eine tiefere Diskussion der multiattributiven Wertfunktion in dieser Dissertation verzichtet.

5. Regelsprache RRML

Die Koordination der Zugriffe auf die Replikate erfolgt bei der Replikationsstrategie RegRess durch Inferenz von Regeln. Während in Kapitel 4 das Protokoll von RegRess spezifiziert wurde, wird in diesem Kapitel die Regelsprache RRML (**R**eplication **R**ule **M**arkup **L**anguage) vorgestellt, die die Formulierung von Regeln in einer eigens entwickelten Regelsprache erlaubt. Bei der RRML handelt es sich um eine Auszeichnungssprache auf Basis von XML (Extensible Markup Language, [MBK00]). Als syntaktische Vorlage für die RRML dienten die XML-basierten Regelsprachen RuleML (Rule Markup Language, [Bol01]) und XRML (eXtensible Rule Markup Language, [LS03]), die die Beschreibung von allgemeinen Regeln ermöglichen. Eine Beschreibung dieser beiden allgemeinen Regelsprachen sowie Grundlagen zu Regeln und regelbasierten Systemen sind im Anhang A dargestellt.

Eine erste Version der RRML wurde im Rahmen dieses Dissertationsvorhabens in einer von mir betreuten Diplomarbeit von Jan Stefan Addicks entwickelt [Add05]. In der Diplomarbeit wurde neben der Regelsprache auch ein Regelinterpreter (Inferenzmaschine, Rule Interpreter) und ein Regeleditor entwickelt, so dass folgende für diese Arbeit relevanten Konzepte und Tools entstanden sind:

Regelsprache RRML 1.0: Die Regelsprache RRML in der Version 1.0 gestattet die Formulierung von Regeln für die Partitionierung der Replikate in synchron und asynchron zu aktualisierende Replikate. Die RRML 1.0 betrifft somit in dieser Arbeit den Teil der Inferenz, der bei der synchronen Aktualisierung herangezogen wird, der InferenzSynchron (siehe Abschnitt 4.2.4). In der RRML 1.0 können Regeln für Komponenten mit Replikat, für logische Objekte und für Replikate vergeben werden, aber Replikationseinheiten wurden noch nicht berücksichtigt.

Regelinterpreter RIM: Der Regelinterpreter RIM (Rule Inferencing Machine) als zentrale Komponente des regelbasierten Systems ist eine Inferenzmaschine für die RRML, d.h. eine Software-Komponente, die von einem Replikationsmanager genutzt wird und die Inferenz von Regeln, die in RRML formuliert sind, durchführt. Hierauf wird in Abschnitt 6.2 detailliert eingegangen.

Regeleditor rulEdit: Der Regeleditor rulEdit (siehe Abschnitt 7.2) erlaubt die Verwaltung von Regeln, die in RRML formuliert sind, und der zugehörigen Mapping-Dateien mittels einer grafischen Benutzeroberfläche. Weiterhin können Simulationsläufe für den semantischen Test der Regeln durch Aufruf der Inferenz-Komponente RIM gestartet werden.

In der hier vorgestellten Version 2.0 wird die RRML gegenüber der Version 1.0 um die Inferenz für die asynchrone Aktualisierung und für die Lesezugriffe (InferenzAsynchron und InferenzLesen, siehe Abschnitt 4.2.4) erweitert. Weiterhin wird bei der InferenzSynchron die Möglichkeit der Regelspezifikation für Replikationseinheiten ergänzt sowie Änderungen hinsichtlich der Protokollunterstützung (siehe Abschnitt 4.2.2) und der fachlichen Konsistenzbedingungen (siehe Abschnitt 4.3.1) eingebracht. Insbesondere die Behandlung widersprüchlicher Regeln wurde verfeinert. Soweit zum Verständnis nötig, werden die entsprechenden Konzepte und Entwürfe der RRML 1.0 übernommen, ohne jeweils auf die zugehörige Diplomarbeit [Add05] zu verweisen.

Der Aufbau dieses Kapitels gliedert sich wie folgt: Zunächst wird in Abschnitt 5.1 die Zielsetzung der RRML erläutert. Dabei wird die in diesem Kapitel verwendete Darstellung für Regeln, die so genannte *„ON-IF-THEN-Darstellung“*, aufgezeigt. Der Abschnitt 5.2 beschäftigt sich mit Anforderungen an die RRML, die nicht-funktionaler Natur sind und die über die in Kapitel 4

beschriebenen Anforderungen für Konsistenzbedingungen und Protokollspezifikationen hinausgehen. Im Abschnitt 5.3 wird die RRML entworfen, d.h. die Anforderungen an die RRML werden in Regeln umgesetzt, wobei wegen der Verständlichkeit die ON-IF-THEN-Darstellung genutzt wird. Die Transformation der ON-IF-THEN-Regeln in das Zielformat XML wird in Abschnitt 5.4 dargelegt. Abschließend erfolgt in Abschnitt 5.5 eine Analyse und Bewertung der RRML, wobei z.B. Terminierungsfragen während der Inferenz der Regeln diskutiert werden.

5.1. Zielsetzung, Regeldarstellung und Notation

Das primäre Ziel der RRML (Replication Rule Markup Language) ist es, Regeln für die Replikationsstrategie RegRess formulieren zu können. Mittels Inferenz dieser Regeln wird bei jedem Zugriff auf ein logisches Objekt die Menge der betroffenen Replikate und die Art der Aktualisierung bestimmt. Somit wird durch die Regeln die von RegRess gewährleistete Konsistenz bestimmt. Daher müssen in der RRML solche Regeln formuliert werden können, die die spezifizierten Konsistenzbedingungen von RegRess abdecken (siehe Abschnitt 4.3).

Weiterhin sollen Regeln einfach formuliert werden können und ihre Verwaltung soll möglichst unkompliziert sein. Daher werden in Abschnitt 5.2 Anforderungen an die RRML aufgestellt, die über die Abdeckung der Konsistenzbedingungen hinausgehen. Es muss auch die Möglichkeit bestehen, die Regeln zwischen Replikationsmanagern, z.B. bei einer Verteilung, zu transferieren und sie zu persistieren. Zusätzlich ist eine einfache Erweiterbarkeit der RRML hilfreich, um zukünftige Anforderungen erfüllen zu können. Daher wurde die RRML auf Basis der XML [MBK00] entwickelt.

Der Nachteil einer Repräsentation der Regeln in XML ist die Länge auf Grund der benötigten Marken (engl. Tag) und die kompliziertere Lesbarkeit beispielsweise gegenüber einfachen Wenn-Dann-Regeln. Daher werden Regeln in nachfolgenden Beispielen und im konzeptionellen Teil (siehe Abschnitt 5.3) als so genannte *„Reaktionsregeln“* (siehe Anhang A.1) dargestellt, bevor sie im Abschnitt 5.4 in XML-Syntax transformiert werden. Reaktionsregeln (reaction rules) werden auch als ECA-Regeln (Event-Condition-Action-Regeln) bezeichnet, die beispielsweise im Bereich der aktiven Datenbanken Anwendung finden [WC95]:

EVENT	*ein beobachtbares Ereignis tritt ein*
CONDITION	*ein Prädikat wird evaluiert, nachdem das Ereignis eingetreten ist*
ACTION	*eine Reihe von Aktionen werden ausgeführt, falls das Prädikat erfüllt ist*

Bei Reaktionsregeln muss also zunächst ein beobachtbares Ereignis eingetreten sein, auf das mit bestimmten Aktionen reagiert werden soll. Diese Aktionen werden aber nur dann ausgeführt, wenn spezifizierte Bedingungen erfüllt sind. Eine alternative Bezeichnung für *„EVENT-CONDITION-ACTION“* ist *„ON-IF-THEN“*, wobei die drei Teile der Reaktionsregel sich entsprechen. Ein Beispiel für eine Reaktionsregel als ON-IF-THEN-Regel ist ein Spamfilter, der in Email-Programmen zu finden ist:

ON	*Email im Posteingang*
IF	*Absender ist xyz@spam.com*
THEN	*verschiebe Email nach Papierkorb*

Da schon bei der Entwicklung der RRML, Version 1.0, die ON-IF-THEN-Darstellung gewählt wurde, soll sie auch hier nachfolgend verwendet werden. Der dreiteilige Aufbau der Reaktionsregeln bietet sich für die Spezifikation von Regeln an, mit denen bei der Replikation die Zugriffe koordiniert werden sollen. Im Ereignisteil sind Zugriffe auf logische oder physische Objekte die beobachtbaren Ereignisse. Ob derartige Ereignisse zu Aktionen führen sollen, kann im Bedingungsteil an bestimmte Prädikate geknüpft werden. Mögliche Aktionen sind dann die Art

des Zugriffs bzw. die betroffenen Replikate festzulegen. Ein Beispiel ist ein Schreibzugriff eines Clients auf ein logisches Datenobjekt, der an einen Replikationsmanager delegiert wird (siehe Abbildung 4.1 auf Seite 61). Wenn dann z.B. der aktuelle Wochentag ein Samstag oder Sonntag ist, dann wird ein spezielles Replikat asynchron aktualisiert. Dieses Beispiel sieht als ON-IF-THEN-Regel wie folgt aus, wobei für das Ereignis und die Aktion einfacher Fließtext verwendet wurde:

ON *Objekt O soll geschrieben werden*
IF *wochentag = Samstag OR wochentag = Sonntag*
THEN *aktualisiere Replikat R asynchron*

An dieser Stelle wird noch kurz auf verwendete Abkürzungen innerhalb der Regeln eingegangen. In Abschnitt 4.1.2 wurde bereits die in dieser Arbeit verwendete Notation an Hand einer Systemlandschaft erläutert. Neben den dort genannten Objekten, Replikaten und Komponenten mit Replikat wird hier der Buchstabe „E“ für eine Replikationseinheit (siehe Definition 35 auf Seite 65) eingeführt, wobei ein tiefergestellter Index eine laufende Nummerierung angibt. Weiterhin soll die Möglichkeit bestehen, Default-Regeln zu formulieren, die alle logischen Objekte betreffen. Hierfür steht die Abkürzung „D“. Damit können folgende Abkürzungen in den Regeln auftauchen:

D Default-Regeln, die für alle logischen Objekte gelten
E_i Replikationseinheit, i = 1,..,s; s Anzahl Replikationseinheiten
O^o o-tes logische Datenobjekt; o = 1,..,m; m Anzahl logischer Datenobjekte
K_k k-te Komponente mit Replikat; k = 1,..,n; n Anzahl Komponenten mit Replikat
R_k^o Physisches Datenobjekt bzw. Replikat; o,k wie oben

Eine Replikationseinheit ist entweder eine Komponente, die einen Zugriff auf ein logisches Objekt durchführt, oder eine Menge von Objekten. Damit könnte z.B. ein logisches Objekt als einelementige Menge einer Replikationseinheit dargestellt werden. Aus Kompatibilitätsgründen zur Version 1.0 der RRML sollen aber auch in der Version 2.0 Regeln für Objekte und Komponenten formuliert werden können. Um die Objekte und Komponenten sowie die Replikationseinheiten mit ihren Elementen identifizieren zu können, müssen entsprechende Zuordnungstabellen (Mapping-Daten) vorhanden sein, die in Abschnitt 5.4 beschrieben werden.

Nachfolgend werden beispielhaft zwei Regeln in der RRML gezeigt, die als Reaktionsregeln in der ON-IF-THEN-Darstellung unter Verwendung der oben genannten Abkürzungen notiert sind:

(1) ON *update(D)* IF *true* THEN *async_update(K_3)*
(2) ON *update(E_3)* IF *hour > 18* THEN *async_update(O^2)*

Mit *„update()“* ist in beiden Regeln ein Schreibzugriff gemeint und mit *„async_update()“* die Zuordnung der entsprechenden Replikate zur asynchronen Aktualisierung. Bei der ersten Regel handelt es sich wegen des Parameters D um eine Default-Regel. Weil hier keine explizite Bedingung gesetzt ist, wird mit dieser Regel formuliert, dass alle Replikate der Komponente K_3 den asynchron zu aktualisierenden Replikaten zugeordnet werden.

In der zweiten Regel ist das beobachtbare Ereignis ein Schreibzugriff auf die Replikationseinheit E_3, genauer gesagt, auf ein logisches Datenobjekt, das zur Replikationseinheit E_3 gehört. Wenn dann die aktuelle Stunde größer als 18 ist, d.h. es ist mindestens 19 Uhr, dann werden alle Replikate des logischen Objekts O^2 den asynchron zu aktualisierenden Replikaten zugeordnet. Alle möglichen Ereignisse, Bedingungen und Aktionen, die in Regeln der RRML 2.0 verwendet werden, sowie deren Semantik werden in Abschnitt 5.3 spezifiziert.

5.2. Anforderungen an die RRML

Die Anforderungen, die an die RRML (Replication Rule Markup Language) gestellt werden, ergeben sich, wie bereits erwähnt, in erster Linie durch die Konsistenzbedingungen und den Protokollspezifikationen (siehe Kapitel 4) der hier vorgestellten Replikationsstrategie RegRess. Es muss mittels der RRML möglich sein, solche Regeln zu formulieren, die diese Anforderungen bzgl. der Konsistenzbedingungen unter Einhaltung der spezifizierten Protokolle abdecken, z.B. ein Abstandsmaß, mit dem das „Alter“ von Replikaten gesteuert wird.

In diesem Abschnitt werden weitere Anforderungen aufgestellt, die die Ausdrucksmächtigkeit der RRML erhöhen. Dabei werden Vereinfachungen und Speicherungsaspekte betrachtet, die bereits in der Version 1.0 der RRML realisiert wurden und die aus Kompatibilitätsgründen auch in Version 2.0 einbezogen werden. Ein tabellarischer Überblick aller Anforderungen befindet sich in Anhang B.

Um die Regelsprache zu vereinfachen und die Anzahl der Regeln zu reduzieren, wird gefordert, dass nicht für jedes logische Objekt eine Regel formuliert werden muss. Wenn keine Regel bei einer Inferenz zum Tragen kommt, dann gilt der Startzustand der Inferenz (siehe Abschnitt 4.2.4). Bei der InferenzSynchron ist der Startzustand z.B. die synchrone Aktualisierung aller Replikate, um Priorität auf die Konsistenz der betroffenen Replikate zu legen. Diese Vereinfachung motiviert folgende Anforderung:

Anforderung 31 ***Weglassen von expliziten Replikationsregeln***: *Es muss nicht individuell für jedes Replikat eines logischen Objekts eine Regel explizit formuliert werden.*

Wenn keine expliziten Regeln angegeben sind, dann gilt, wie oben erwähnt, der Startzustand der Inferenz, der im Abschnitt 4.2.4 spezifiziert ist. Um diese Voreinstellung allgemein zu ändern bzw. zu verfeinern, soll die Möglichkeit bestehen, Default-Regeln zu formulieren, die für alle logischen Objekte gelten:

Anforderung 32 ***Default-Regeln***: *Regeln, die für alle logischen Objekte gelten, können als Default-Regeln formuliert werden.*

Die Default-Regeln aus Anforderung 32 stellen im Grunde eine Vereinfachung dar, weil andernfalls die entsprechenden Aktionen für jede Replikationseinheit formuliert werden müssten. Ähnlich ist eine Vereinfachung für Regeln wünschenswert, die alle Replikate einer Komponente betreffen, d.h. es wird nicht die initiierende Komponente betrachtet, sondern die „Zielkomponente“, die die betroffenen Replikate speichert:

Anforderung 33 ***Regeln für alle Replikate einer Komponente***: *Es sollen Regeln formuliert werden können, die für alle Replikate einer Komponente gelten.*

Damit Regeln einfacher identifiziert werden können und ihre Auswirkungen leicht verständlich sind, werden optionale Attribute nachfolgend gefordert, mit denen den Regeln ein Titel zugeordnet werden kann und eine Beschreibung informeller Art hinterlegt werden kann:

Anforderung 34 ***Beschreibungsmöglichkeit von Regeln***: *Es soll eine Möglichkeit geboten werden, Regeln in natürlicher Sprache zu beschreiben und mit einem Titel oder einem Bezeichner versehen zu können, wenn dieses benötigt oder gewünscht wird.*

Analog zu XRML [KL03] soll eine logische Separation von zusammengehörenden Regeln möglich sein, d.h. es soll eine Möglichkeit der Gruppierung von Regeln bestehen. Auch für Gruppen sollen optionale Attribute für Titel und Beschreibung vorgesehen werden:

Anforderung 35 ***Gruppierung von Regeln***: *Regeln sollen sich gruppieren lassen, um logische Einheiten zu bilden. In diesem Zusammenhang ist es sinnvoll, zusätzliche Attribute zur Beschreibung und Bezeichnung der Gruppe einzuführen.*

Zur Verbesserung der Übersicht innerhalb des RRML-Dokuments und zur Vereinfachung des Parsens eines RRML-Dokuments soll die Möglichkeit geschaffen werden, Regeln auszulagern und unter bestimmten Bedingungen wieder einzubinden:

Anforderung 36 ***Auslagerung von Regeln:*** *Regeln sollen in separate Dokumente ausgelagert werden können. Als Resultat von Regeln sollen solche externen Dokumente während der Inferenz wieder eingebunden werden können.*

Die Struktur von Regeln wurde so modelliert, dass jede Regel für eine Komponente mit Replikat gilt (siehe Abbildung 4.3 auf Seite 66). Wenn nun alle n Replikate eines logischen Objekts auf die gleiche Weise aktualisiert werden sollen, müssten n Regeln für die einzelnen Replikate formuliert werden. Eine Reduzierung der Anzahl Regeln stellt nachfolgende Anforderung dar, die eine einfache Definition der Aktualisierungsart von logischen Objekten fordert, die dann intern auf die betroffenen Replikate bzw. Komponenten mit Replikat umgesetzt werden müssen:

Anforderung 37 ***Allgemeine Aktualisierungsart*** *Es muss für logische Objekte bzw. Replikationseinheiten einfach definierbar sein, dass deren Replikate nur auf eine Art aktualisiert werden sollen.*

Die Anforderung 37 zielt auf eine Vereinfachung der Anforderung 19, in der je Replikationseinheit ein Zustand gesetzt werden kann, mit der die jeweiligen Replikate zu aktualisieren sind. Wie bereits erwähnt, dienen die Anforderungen in diesem Abschnitt der Erhöhung der Ausdrucksmächtigkeit der RRML. Sie beeinflussen nicht die Arbeitsweise bzw. das Protokoll der Replikationsstrategie RegRess, sondern vereinfachen die Verwaltung der Regeln.

5.3. Konzeptioneller Entwurf der RRML

Der konzeptionelle Entwurf dient dazu, den Sprachumfang der RRML (Replication Rule Markup Language) festzulegen. Dabei müssen die Anforderungen an die RRML (siehe Anhang B) erfüllt werden. Mittels der RRML können Regeln spezifiziert werden, die bei der Replikationsstrategie RegRess die Koordination der Zugriffe beeinflussen, genauer gesagt, mit denen die Menge der Replikate für die Zugriffe bestimmt werden. Die Regeln werden in diesem Abschnitt in der ON-IF-THEN-Darstellung (siehe Abschnitt 5.1) präsentiert, bevor sie im Abschnitt 5.4 in XML transformiert werden.

Bei der Vorstellung des konzeptionellen Entwurfs wird zunächst in Abschnitt 5.3.1 die formale Spezifikation der Regeln vorgenommen, d.h. der Sprachumfang wird in der so genannten Backus-Naur-Form (BNF, [Sch01b]) präsentiert. Anschließend folgt eine detaillierte Erläuterung der möglichen Regeln, wobei in Abschnitt 5.3.2 die Regeln für die synchrone Aktualisierung, in Abschnitt 5.3.3 die Regeln für die asynchrone Aktualisierung und in Abschnitt 5.3.4 die Regeln für Lesezugriffe vorgestellt werden. Die Behandlung von widersprüchlichen Regeln mit den zugehörigen Widerspruchsregeln wird in Abschnitt 5.3.5 diskutiert. Weil für alle Regeln der Bedingungsteil gleich ist, wird hierauf in Abschnitt 5.3.6 eingegangen. Weil damit der Regelaufbau komplettiert ist, befinden sich in Abschnitt 5.3.6 auch einige Beispiele. Abschließend folgt in Abschnitt 5.3.7 die Verifikation der Anforderungen.

5.3.1. Formale Spezifikation der Regeln

In diesem Abschnitt wird der syntaktische Aufbau der Regeln formal spezifiziert, d.h. die Grammatik der RRML wird definiert. In der RRML werden Reaktionsregeln (siehe Anhang A.1) formuliert, die hier in der ON-IF-THEN-Form dargestellt werden. Eine derartige Reaktionsregel hat allgemein folgenden Aufbau:

ON *Ereignisteil: ereignis(Parameter)*
IF *Bedingungsteil: prädikate()*
THEN *Aktionsteil: aktion(Parameter)*

Der Ereignisteil besteht aus einem Ereignis, auf das reagiert werden soll, sofern zusätzlich die angegebenen Bedingungen erfüllt sind. Weil es bei der RRML „gleiche“ Ereignisarten gibt, z.B. das synchrone Aktualisieren, wird das eigentliche Ereignis über einen Parameter identifiziert. Ein Beispiel für „ereignis(Parameter)“ ist „update(E_3)“, d.h. für die Ereignisart „update“ (synchrone Aktualisierung) wird mittels des Parameters E_3 das folgende Ereignis identifiziert: Eine synchrone Aktualisierung eines logischen Objekts, das zur Replikationseinheit E_3 gehört.

Der Bedingungsteil ist ein boolscher Ausdruck. Wenn die Bedingung erfüllt ist, also der Wahrheitswert „wahr“ geliefert wird, dann wird für das eingetretene Ereignis die angegebene Aktion ausgeführt. Einzelne Prädikate können durch die logischen Operatoren „und“ und „oder“ verknüpft werden. Ein Ausdruck kann auch Methoden enthalten, z.B. wird bei $30 < datasize()$ die Datengröße des logischen Datenobjekts ermittelt, bevor der Ausdruck berechnet wird.

Im Aktionsteil wird die Aktion beschrieben, die bei eingetretenem Ereignis und erfüllter Bedingung ausgeführt wird. Die Aktion kann selbst wiederum ein Ereignis auslösen. Daher werden hier die Aktionen syntaktisch wie Ereignisse spezifiziert, d.h. in Form einer Aktionsart und einem Parameter, wobei die Aktionsarten auch Ereignisarten sind. Für ein Ereignis, das auf Grund einer derartigen Aktion ausgelöst wurde, kann optional eine Regel formuliert sein. Somit gleichen Aktionen syntaktisch den Ereignissen, haben aber eine andere semantische Bedeutung. Neben dem Auslösen eines Ereignisses werden in der Aktion weitere Operationen ausgeführt, z.B. die Zuordnung von Replikaten zu den asynchron zu aktualisierenden Replikaten bei der synchronen Aktualisierung. Die Semantik der Ereignisse und Aktionen wird in den Abschnitten 5.3.2 bis 5.3.5 erläutert, die Bedingungen sind im Abschnitt 5.3.6 beschrieben.

An dieser Stelle wird die Grammatik der RRML beschrieben. Eine Grammatik [Sal79, Sch01b] ist ein 4-Tupel $G = (V, \Sigma, P, S)$, das folgende Bedingungen erfüllt: V ist eine endliche Menge von Nichtterminalen. Σ ist eine endliche Menge, das Terminalalphabet. P ist die endliche Menge der Produktionen. $S \in V$ ist die Startvariable. In Listing 5.1 sind die Produktionen in der so genannten Backus-Naur-Form (BNF, [Sch01b]) dargestellt. Die Nichtterminale V sind in eckigen Klammern und das Terminalalphabet als Text gegeben. Die Startvariable S ist <Regel>. Da es sich um eine kontextfreie Grammatik handelt, ist die RRML eine kontextfreie Sprache.

Im Listing 5.1 ist der syntaktische Aufbau von Regeln der RRML dargestellt. Dabei erfolgt zunächst eine Klassifizierung der Regeln entsprechend der Inferenzart: InferenzSynchron, InferenzAsynchron und InferenzLesen (siehe Abschnitt 4.2.4). Zusätzlich können Regeln formuliert werden, die bei der Behandlung von widersprüchlichen Regeln (siehe Abschnitt 5.3.5) berücksichtigt werden, bei der auch mittels Inferenz ein Ergebnis ermittelt wird. Die Regeln der vier Regelarten haben den gleichen Aufbau, jedoch unterschiedliche Ereignisse und Aktionen. Der Bedingungsteil gleicht sich für alle Regeln. Der Ereignisteil bzw. der Aktionsteil besteht aus einer Ereignisart (oder einfach Ereignis) bzw. aus einer Aktionsart (oder einfach Aktion) und einem Parameter, der geklammert wird. Der Parameter ist für alle Regeln gleich und wurde in Abschnitt 5.1 als Abkürzung vorgestellt.

Weil ein Ereignis entweder extern ausgelöst wird, z.B. ein Schreibzugriff (Aktualisierung) auf ein logisches Objekt, der von einem Client initiiert wird, oder intern ausgelöst wird, z.B. durch eine Aktion, die eine bestimmte Aktualisierungsart für Replikate vorgibt, wird im Listing 5.1 zwischen externen und internen Ereignisarten unterschieden. Jede Aktion, bis auf die Aktionsart `use_ruleset`, löst ein Ereignis gleicher Art mit gleichem Parameter aus, sodass sich der syntaktische Aufbau von Aktionen und Ereignissen in diesem Fall gleicht, jedoch eine unterschiedliche Semantik aufweist. Auf die Semantik wird in den Abschnitten 5.3.2 bis 5.3.6 eingegangen, die zusätzlich eine detaillierte Beschreibung der Konstrukte enthalten sowie die Aktivitäten der Aktionen erläutern.

Listing 5.1: Backus-Naur-Form der Regeln in RRML

```
<Regel> ::= <RegelSynchron>|<RegelAsynchron>|<RegelLesen>|
            <RegelWiderspruch>

<RegelSynchron> ::= ON <EreignisSynchron>(<Parameter>)
                    IF <Bedingung>
                    THEN <AktionSynchron>

<EreignisSynchron> ::= <ExternesES>|<InternesES>
<ExternesES> ::= update
<InternesES> ::= sync_update|asyn_update|
                 changeableTrue|changeableFalse
<AktionSynchron> ::= <InternesES>(<Parameter>)|
                     use_ruleset(<FileId>)

<RegelAsynchron> ::= ON <EreignisAsynchron>(<Parameter>)
                     IF <Bedingung>
                     THEN <AktionAsynchron>
<EreignisAsynchron> ::= <ExternesEA>|<InternesEA>
<ExternesEA> ::= write
<InternesEA> ::= later|one|all
<AktionAsynchron> ::= <InternesEA>(<Parameter>)|
                      use_ruleset(<FileId>)

<RegelLesen> ::= ON <EreignisLesen>(<Parameter>)
                 IF <Bedingung>
                 THEN <AktionLesen>
<EreignisLesen> ::= read
<AktionLesen> ::= getConsistency(<ParameterKonsistenz>)|
                  getPerformance(<ParameterPerformance>)|
                  use_ruleset(<FileId>)

<RegelWiderspruch> ::= ON <EreignisWiderspruch>(<Parameter>)
                       IF <Bedingung>
                       THEN <AktionWiderspruch>
<EreignisWiderspruch> ::= conflict
<AktionWiderspruch> ::= set(<ParameterWiderspruch>)

<Parameter> ::= $D|E_i|O^o|K_k|R_k^o$

<ParameterKonsistenz>  ::= V|T|A
<ParameterPerformance> ::= R|C
<ParameterWiderspruch> ::= CGP|FGP|PCG|PFG

<FileID> ::= "äPlattformabhngiger␣Dateiname"

<Bedingung> ::= true|false|<Praedikat>|¬(<Bedingung>)|
                (<Bedingung> ∨ <Bedingung>)|
                (<Bedingung> ∧ <Bedingung>)

<Praedikat> ::= <Funktion> <Operator> <Wert>
<Funktion> ::= day|month|year|weekday|hour|minute|
               init_node|target_node|object|
               diff_version|diff_time|diff_value|period|consistency|
               datasize|conflicts|connection_throughput|responsetime
<Operator> ::= =|≠|>|≥|<|≤
<Wert> ::= "reelle␣Zahl"
```

Ein Regelinterpreter (siehe Abschnitt 6.2) zieht bei der Inferenz diese Regeln heran, um die Art der Koordination bei Schreib- und Lesezugriffen gemäß den Protokollen aus Abschnitt 4.2 zu bestimmen. Der Ablauf der Inferenz gleicht sich für die Inferenzarten InferenzSynchron, InferenzAsynchron und InferenzLesen. Zunächst werden die externen Ereignisse bestimmt, die zu Beginn der Inferenz als eingetreten gesetzt werden. Anschließend werden die Regeln ermittelt, deren Ereignisse mit den eingetretenen Ereignissen übereinstimmen. Bei passenden Regeln wird der Bedingungsteil geprüft. Wenn die Bedingung erfüllt ist, dann kann die Aktion ausgeführt werden, es sei denn, ein Widerspruch zu einer bereits ausgeführten Regel wird erkannt. Bei einem Widerspruch wird eine Widerspruchsbehandlung aktiviert. Wenn die Aktion ausgeführt wird und ein internes Ereignis auslöst, dann muss wieder geprüft werden, ob es weitere passende Regeln gibt (siehe auch Listing 7.2 auf Seite 191).

5.3.2. Regeln für die synchrone Aktualisierung

Nachdem in Abschnitt 5.3.1 die Syntax der Regeln definiert wurde, wird in diesem Abschnitt die Semantik zunächst der Regeln spezifiziert, die bei der synchronen Aktualisierung formuliert werden können. Da die Bedingungen für alle Inferenzarten gemeinsam in Abschnitt 5.3.6 beschrieben werden, erfolgt hier eine Erläuterung folgender Aspekte:

- Ergebnis der Inferenz bei der synchronen Aktualisierung
- Ereignisse der RRML bei der synchronen Aktualisierung
- Aktivitäten einer Aktion der RRML bei der synchronen Aktualisierung
- Aktionen der RRML bei der synchronen Aktualisierung
- Startereignisse der RRML bei der synchronen Aktualisierung

Ergebnis der Inferenz bei der synchronen Aktualisierung

Bei der synchronen Aktualisierung ist das Ziel der Inferenz (InferenzSynchron) die Partitionierung der betroffenen Replikate in synchron und asynchron zu aktualisierende Replikate (siehe Abschnitt 4.2.4). Daher wird in einer Aktion neben dem Auslösen eines Ereignisses i.a. auch eine Zuordnung von Replikaten zu den Mengen M_S (synchron zu aktualisierende Replikate) und M_A (asynchron zu aktualisierende Replikate) durchgeführt. Wie die Mengen M_S und M_A in einer Implementierung realisiert werden, hängt von der verwendeten Inferenzmaschine ab (siehe Kapitel 7). Zum Abschluss der InferenzSynchron muss dieses Ergebnis dem Aufrufer der InferenzSynchron geliefert werden, nämlich der Methode `schreibzugriffSynchron` (siehe Listing 4.1 auf Seite 71).

Hier soll angenommen werden, dass ein Ergebnisobjekt für die zu aktualisierenden Replikate des logischen Objekts O^o geführt wird, das im Folgenden als `ErgebnisSynchron` bezeichnet wird. Da es sich laut Annahme um eine vollreplizierte Datenbank mit Replikationsgrad n handelt (siehe Abschnitt 4.1.1), müssen n Replikate aktualisiert werden, für die Informationen in `ErgebnisSynchron` gehalten werden müssen. Daher sind zwei der nachfolgenden Attribute des Objekts `ErgebnisSynchron` n-dimensional, d.h. es können entsprechende Werte für die n Replikate gespeichert werden. `ErgebnisSynchron` beinhaltet somit folgende Attribute:

- `aktualisierungsart`: n-dimensionales Attribut, wobei jedes Element den Wert `synchron` oder `asynchron` annehmen kann. Es wird festgehalten, ob das Replikat R_k^o, $k = 1, 2, ..n$ synchron bzw. asynchron aktualisiert werden soll. Die n Werte des Attributs müssen mit `synchron` initialisiert werden (siehe Anforderung 1 auf Seite 79).
- `wechselfaehig`: n-dimensionales Attribut, wobei jedes Element den Wert `true` oder `false` annehmen kann. Es wird festgehalten, ob das Replikat R_k^o, $k = 1, 2, ..n$ wechselfähig ist oder nicht. Die n Werte des Attributs werden mit `false` initialisiert.
- `regeln`: Speichert eine Liste der Regeln, die bei der Inferenz berücksichtigt wurden. Hierüber können ggf. Widersprüche in den Regeln erkannt werden.

Die Partitionierung der n Replikate in die Mengen M_S und M_A erfolgt direkt an Hand des Attributs `aktualisierungsart`. Wenn während des synchronen Schreibzugriffs ein Fehler bei einem wechselfähigen Replikat auftritt, dann wird dieses Replikat aus der Menge M_S entnommen und in die Menge M_A aufgenommen, sodass dieses Replikat bei einem wiederholten Schreibzugriff zu den asynchron zu aktualisierenden Replikaten zählt (siehe Listing 4.1 auf Seite 71). Daher benötigt die Methode `schreibzugriffSynchron` für jedes der n Replikate auch das Attribut `wechselfaehig`. Das Attribut `regeln` hingegen dient ausschließlich der Erkennung von widersprüchlichen Regeln.

Ereignisse der RRML bei der synchronen Aktualisierung

In Listing 5.1 auf Seite 117 ist durch das Nichtterminal `<EreignisSynchron>` definiert, auf welche Ereignisse bei der synchronen Aktualisierung reagiert wird. Diese Ereignisse werden im Folgenden erläutert:

- `update(<Parameter>)`: Eine Aktualisierung, d.h. ein Schreibzugriff eines Clients, wird an den Replikationsmanager gestellt. Hierbei handelt es sich um ein externes Ereignis, dass durch einen Schreibzugriff eines Clients auf ein logisches Objekt initiiert wird. Somit werden hierüber die Startereignisse gebildet (siehe unten).

  `<Parameter>`:

  D Default-Ereignis. Mittels des Default-Parameters können Regeln für alle Objekte und Replikationseinheiten formuliert werden.

  E_i i-te Replikationseinheit. Weil es sich um ein Startereignis handelt, sind die Replikationseinheiten betroffen, die entweder das logische Objekt, auf das vom Client zugegriffen wird, oder die initiierende Komponente beinhalten. Bei einem Schreibzugriff können mehrere Replikationseinheiten betroffen sein, wenn z.B. ein logisches Objekt in mehreren Replikationseinheiten auftaucht oder wenn die initiierende Komponente oder Komponenten mit Replikat zu Replikationseinheiten zusammengefasst sind (siehe unten, Startereignisse).

  O^o Das logische Objekt, auf das vom Client zugegriffen werden soll.

  K_k Die initiierende Komponente. Hierbei muss es sich nicht um eine Komponente mit Replikat handeln. Es soll gelten: $k = 1, .., n$ Komponente mit Replikat (n = Replikationsgrad), $k = n + 1, .., m$ Komponente ohne Replikat

  R_k^o Derzeit nicht für die Ereignisart `update` benötigt, weil ein Client kein spezielles Replikat schreibt, sondern alle n Replikate eines logischen Objekts.

- `sync_update(<Parameter>)`: Eine synchrone Aktualisierungsanforderung für die in `<Parameter>` übergebene Einheit wurde als Ereignis ausgelöst. Hierbei handelt es sich um ein intern ausgelöstes Ereignis, d.h. das Ereignis wurde als Aktion einer Regel ausgelöst.

  `<Parameter>`:

  D wie beim Ereignis `update()`.

  E_i wie beim Ereignis `update()`, jedoch kann eine beliebige Replikationseinheit betroffen sein.

  O^o wie beim Ereignis `update()`, jedoch kann ein beliebiges logisches Objekt betroffen sein.

  K_k Eine Komponente mit Replikat. Im Gegensatz zu `update()` sind hier nur die „Zielsysteme“ gemeint, d.h. die Komponenten, deren Replikate geschrieben werden sollen.

  R_k^o Ein bestimmtes Replikat, für das eine Regel formuliert wird. Dies ist die feinste Granularitätsstufe.

- `async_update(<Parameter>)`: Analog zum Ereignis `sync_update()` wurde eine asynchrone Aktualisierungsanforderung für die in `<Parameter>` übergebene Einheit als Ereignis ausgelöst, wobei es sich ebenfalls um ein intern ausgelöstes Ereignis handelt.

  `<Parameter>`: wie beim Ereignis `sync_update()`.

- `changeableTrue(<Parameter>)`: Das eingetretene Ereignis ist eine Änderung des Attributs `wechselfaehig` auf `true`, wobei das intern ausgelöste Ereignis für die in `<Parameter>` übergebene Einheit eingetroffen ist.

  `<Parameter>`: wie beim Ereignis `sync_update()`.

- `changeableFalse(<Parameter>)`: Analoges Ereignis zu `changeableTrue()`, wobei hier das Attribut `wechselfaehig` auf `false` gesetzt wurde.

  `<Parameter>`: wie beim Ereignis `sync_update()`.

Wie bereits erwähnt, wird der Bedingungsteil der Regeln in Abschnitt 5.3.6 gemeinsam für alle Regelarten erläutert, sodass nachfolgend zunächst auf die Aktionen bei der synchronen Aktualisierung eingegangen wird.

Aktivitäten einer Aktion der RRML bei der synchronen Aktualisierung

Bevor auf die Aktionen der RRML bei der synchronen Aktualisierung detailliert eingegangen wird, sollen zunächst die Aktivitäten einer Aktionsausführung beschrieben werden. Die Beschreibung erfolgt in der in Abschnitt 4.2.2 verwendeten Pseudo-Sprache, insbesondere weil die Bearbeitung einer Aktion einem Methodenablauf gleicht. Vorausgesetzt wird, dass ein Ergebnisobjekt `ErgebnisSynchron` mit den oben genannten Attributen existiert, auf das einfachheitshalber global zugegriffen werden kann. Das Ziel einer Aktion ist die Manipulation der Attribute, um die n Replikate zu partitionieren (siehe oben), und ein Ereignis gleichen Namens zu erzeugen, für das möglicherweise Regeln spezifiziert sind. Eine Ausnahme bildet die Aktion `use_ruleset`, mit der Regeln aus einer Datei nachgeladen werden.

Von einer Aktion sind nicht grundsätzlich alle n Replikate betroffen. Die Ermittlung der von der ausgeführten Aktion betroffenen Replikate hängt einerseits vom Parameter der Aktion ab und andererseits vom logischen Objekt, das durch den Schreibzugriff des Clients geändert werden soll. Es wird angenommen, dass O^z das zugegriffene, logische Objekt sei. Dann werden in Abhängigkeit des Parameters der Aktion die folgenden Replikate ermittelt, deren zugehörige Attribute in `ErgebnisSynchron` ggf. manipuliert werden müssen:

D	Alle n Replikate des logischen Objekts O^z, d.h. R^z_* (* steht für alle Komponenten).
E_i	Wenn $O^z \in E_i$, dann R^z_*, sonst kein Replikat.
O^o	Wenn $O^z = O^o$, dann R^z_*, sonst kein Replikat.
K_k	R^z_k.
R^o_k	Wenn $z = o$, dann R^z_k, sonst kein Replikat.

Bei der Ausführung einer Aktion ist zu beachten, ob die Aktion in Widerspruch zu einer anderen Regel, genauer gesagt Aktion einer Regel, steht. Ein Widerspruch liegt dann vor, wenn die Attribute von `ErgebnisSynchron` unterschiedlich geändert werden. In diesem Fall muss der Widerspruch gelöst werden, wofür Regeln spezifiziert werden können (siehe Abschnitt 5.3.5).

Der Ablauf einer Aktion bei der synchronen Aktualisierung ist im Listing 5.2 als Methode `actionSynchron` dargestellt. Als Parameter wird das logische Objekt O^z, das geschrieben werden soll, und die Regel `regel` übergeben. Dabei wird davon ausgegangen, dass `regel` neben anderen Informationen die Aktion in `regel.aktionsart` und den zugehörigen Parameter in `regel.aktionsparameter` speichert. Nachfolgend werden die einzelnen Aktivitäten des Ablaufs erläutert:

Listing 5.2: Ablauf einer Aktion bei der synchronen Aktualisierung

```
public void aktionSynchron(Object O^z, Object regel) {

  anhaengenRegel(regel);

  if (regel.aktionsart = "use_ruleset") {
    ladeRegelnAusDatei(regel.aktionsparameter);
    return;
  }

  List replikate := bestimmeReplikate(O^z,regel.aktionsparameter);

  Object status := behandleWiderspruch(replikate,regel);

  if (status = NoSolution) {
    initiiereErgebnis();
    throw AbbruchInferenz;
  }

  if (status = NoAction) return;

  setzeAttributeErgebnis(replikate,regel.aktionsart);
  erzeugeEreignis(regel.aktionsart,regel.aktionsparameter);
  return;
}
```

Zeile 3 Die aktuell ausgeführte Regel wird der Regelliste der ausgeführten Regeln, die in `ErgebnisSynchron` unter dem Attribut `regeln` geführt wird, angehängt.

Zeile 5 Wenn es sich um eine Aktion `use_ruleset` handelt, folgt ein gesonderter Ablauf, der in den nachfolgenden Zeilen 6 und 7 beschrieben ist.

Zeile 6 Regeln, die in eine Datei ausgelagert wurden, werden wieder eingebunden (siehe Anforderung 36 auf Seite 115). Mit dem Parameter `regel.aktionsparameter` wird die zu lesende Datei identifiziert, wobei eine plattformspezifische Notation verwendet wird. Es muss geprüft werden, ob für die eingebundenen Regeln Ereignisse, die bereits bearbeitet wurden, eingetreten sind.

Zeile 7 Weil bei einer Aktion `use_ruleset` kein Ereignis ausgelöst wird, terminiert der Ablauf in diesem Fall an dieser Stelle.

Zeile 10 In Abhängigkeit des zu schreibenden Objekts O^z und des Parameters der Aktion `regel.aktionsparameter` werden in einer Liste die Replikate ermittelt, deren entsprechenden Attribute in `ErgebnisSynchron` potenziell geändert werden müssen (siehe oben).

Zeile 12 In der Variablen `status` wird der Rückgabewert der Widerspruchsbehandlung gespeichert, die in jedem Fall aufgerufen wird. Die Behandlung von Widersprüchen ist detailliert in Abschnitt 5.3.5 erläutert. Die Variable `status` kann folgende Werte annehmen:

`Action` bedeutet, dass kein Widerspruch aufgetreten ist oder die aktuelle Regel ausgeführt werden kann, weil z.B. die Priorität höher ist als die Priorität der in Widerspruch stehenden Regel.

`NoAction` bedeutet, dass ein Widerspruch aufgetreten ist und die aktuelle Regel nicht ausgeführt werden darf, weil z.B. die Priorität niedriger ist als die Priorität der in Widerspruch stehenden Regel.

`NoSolution` bedeutet, dass ein Widerspruch aufgetreten ist und der Widerspruch nicht aufgelöst werden konnte, weil z.B. keine geeignete Widerspruchsregel vorlag.

Zeile 14 Wenn ein Widerspruch aufgetreten ist, der nicht gelöst werden konnte, dann muss die Inferenz abgebrochen werden, was durch die Zeilen 15 und 16 zum Ausdruck kommt.

Zeile 15 Die Attribute von `ErgebnisSynchron` werden gemäß der Anforderung 1 auf Seite 79 erneut initialisiert. Alle Replikate werden den synchron zu aktualisierenden Replikaten zugeordnet.

Zeile 16 Mittels des Befehls **throw** wird eines Ausnahme angezeigt, die von der aufrufenden Methode behandelt wird: Die Inferenz muss abgebrochen werden. Das Ergebnis der Inferenz ist in diesem Fall, dass alle Replikate synchron aktualisiert werden müssen (siehe Anforderung 5 auf Seite 80).

Zeile 19 Wenn ein Widerspruch aufgetreten ist, der derart gelöst wurde, dass keine Aktion ausgeführt werden darf, dann wird der Ablauf der Aktion an dieser Stelle beendet. Andernfalls können die Hauptaktivitäten der Aktion, nämlich das Manipulieren der Attribute und das Erzeugen eines Ereignisses, ausgeführt werden (siehe Zeilen 21 und 22).

Zeile 21 Die Attribute von `ErgebnisSynchron`, die zu den Replikaten `replikate` gehören, werden entsprechend der aktuell ausgeführten Aktion `regel.aktionsart` manipuliert. Eine detaillierte Beschreibung folgt unten bei der Erläuterung der möglichen Aktionen der RRML.

Zeile 22 Es wird ein Ereignis mit dem gleichen Namen der Aktion und dem gleichen Parameter erzeugt. Anschließend muss geprüft werden, ob Regeln für dieses Ereignis vorliegen.

Zeile 23 Der Ablauf der Aktion wird beendet.

Aktionen der RRML bei der synchronen Aktualisierung

Im Folgenden werden die einzelnen Aktionen beschrieben, die bei der Inferenz zur Ermittlung der Partitionierung bei der asynchronen Aktualisierung ausgeführt werden können. Die möglichen Aktionen sind in Listing 5.1 auf Seite 117 durch das Nichtterminal `<AktionSynchron>` definiert:

- `sync_update(<Parameter>)`: Die durch diese Aktion betroffenen Replikate (siehe oben) werden den synchron zu aktualisierenden Replikaten zugeordnet, d.h. in `ErgebnisSynchron` wird das Attribut `aktualisierungsart` der entsprechenden Replikate auf `synchron` gesetzt.

  `<Parameter>`:

  D Default-Aktion, d.h. die Aktion wird für alle logischen Objekte durchgeführt.

  E_i Die Aktion wird für die i-te Replikationseinheit durchgeführt.

  O^o Die Aktion wird für das o-te logische Objekt durchgeführt.

  K_k Die Aktion wird für die k-te Komponente mit Replikat durchgeführt.

  R_k^o Die Aktion wird für das Replikat durchgeführt, das zum o-ten logischen Objekt gehört und auf der k-ten Komponente mit Replikat lokalisiert ist.

- `async_update(<Parameter>)`: Analog zur Aktion `sync_update()` werden die von dieser Aktion betroffenen Replikate den asynchron zu aktualisierenden Replikaten zugeordnet, d.h. in `ErgebnisSynchron` wird das Attribut `aktualisierungsart` der entsprechenden Replikate auf `asynchron` gesetzt.

  `<Parameter>`: wie bei der Aktion `sync_update()`.
- `changeableTrue(<Parameter>)`: Die durch diese Aktion betroffenen Replikate werden als „wechselfähig" gekennzeichnet (siehe Abschnitt 4.2.2), d.h. in `ErgebnisSynchron` wird das Attribut `wechselfaehig` der entsprechenden Replikate auf `true` gesetzt.

  `<Parameter>`: wie bei der Aktion `sync_update()`.
- `changeableFalse(<Parameter>)`: Analoge Aktion zu `changeableTrue()`, wobei hier die durch die Aktion betroffenen Replikate als nicht wechselfähig gekennzeichnet werden, d.h. in `ErgebnisSynchron` wird das Attribut `wechselfaehig` der entsprechenden Replikate auf `false` gesetzt.

  `<Parameter>`: wie bei der Aktion `sync_update()`.
- `use_ruleset(<FileID>)`: Eine Menge bzw. Gruppe von Regeln werden eingebunden, indem sie aus einer Datei eingelesen werden (siehe Anforderung 36 auf Seite 115).

  `<FileID>`: Dateiname in Form des zugrunde liegenden Dateisystems.

Startereignisse der RRML bei der synchronen Aktualisierung
Damit die Inferenz starten kann, müssen zu Beginn Ereignisse eingetreten sein. Diese Ereignisse werden hier als *„Startereignisse"* bezeichnet. Bei der Inferenz, die die Partitionierung bei der synchronen Aktualisierung ermittelt (InferenzSynchron), werden zu Beginn folgende Startereignisse ausgelöst, wobei angenommen sei, dass von der Komponente K_k ein Schreibzugriff auf das logische Objekt O^z durchgeführt werden soll:

- `update`(D)
- `update`(E_i), mit $O^z \in E_i$ oder $K_k \in E_i$.
  Anmerkung: Es können mehrere Replikationseinheiten E_i betroffen sein.
- `update`(O^z)
- `update`(K_k)

Angemerkt sei, dass nicht für jedes Startereignis eine Regel formuliert sein muss. Falls für ein Startereignis oder ein intern ausgelöstes Ereignis keine Regel formuliert ist, wird trivialerweise keine Aktion ausgeführt.

5.3.3. Regeln für die asynchrone Aktualisierung

Die Semantik der Regeln für die asynchrone Aktualisierung wird analog zu den Regeln für die synchrone Aktualisierung erläutert (siehe Abschnitt 5.3.2), d.h. es werden folgende Punkte diskutiert:

- Ergebnis der Inferenz bei der asynchronen Aktualisierung
- Ereignisse der RRML bei der asynchronen Aktualisierung
- Aktivitäten einer Aktion der RRML bei der asynchronen Aktualisierung
- Aktionen der RRML bei der asynchronen Aktualisierung
- Startereignisse der RRML bei der asynchronen Aktualisierung

Ergebnis der Inferenz bei der asynchronen Aktualisierung
Das Ziel der Inferenz bei der asynchronen Aktualisierung (InferenzAsynchron) ist die Ermittlung der Entscheidung, ob ein Schreibzugriff auf das Replikat derzeit durchgeführt werden darf oder nicht. Wenn der Schreibzugriff derzeit nicht durchgeführt werden darf, dann verbleibt der

Auftrag in der Replica Queue (siehe Abschnitt 4.2.2). Andernfalls kann bei erfolgreichem Schreibzugriff auf das Replikat der Auftrag aus der Replica Queue gelöscht werden. Dabei wird unterschieden, ob alle Aufträge dieses Replikats ohne erneute Inferenz abgearbeitet werden oder nicht. Dieses Ergebnis wird zum Abschluss der InferenzAsynchron dem Aufrufer geliefert, nämlich der Methode `schreibzugriffAsynchron` (siehe Listing 4.2 auf Seite 74).

In ähnlicher Weise wie bei der InferenzSynchron wird bei der InferenzAsynchron ein Ergebnisobjekt mitgeführt, das nachfolgend `ErgebnisAsynchron` genannt wird. Weil bei der asynchronen Aktualisierung jedoch nur ein einzelnes Replikat geschrieben wird, muss lediglich ermittelt werden, ob derzeit ein Schreibzugriff erlaubt ist. Die Attribute von `ErgebnisAsynchron` lauten somit:

- `zugriff`: Das Attribut `zugriff` speichert die möglichen Ergebnisse der InferenzAsynchron (siehe Abschnitt 4.2.4): `KeinZugriff` bedeutet, dass derzeit kein Schreibzugriff erlaubt ist. `EinAuftrag` bedeutet, dass ein Auftrag aus der Replica Queue bearbeitet werden kann, aber anschließend eine erneute Inferenz durchgeführt werden muss. `AlleAuftraege` bedeutet, dass alle Aufträge für dieses Replikat aus der Replica Queue ohne erneute Inferenz bearbeitet werden können. Dieses Attribut muss mit `AlleAuftraege` initialisiert werden (siehe Anforderung 6 auf Seite 80).
- `regeln`: Speichert eine Liste der Regeln, die bei der Inferenz berücksichtigt wurden. Hierüber können ggf. Widersprüche in den Regeln erkannt werden.

Auch hier dient das Attribut `regeln` lediglich zur Erkennung von widersprüchlichen Regeln und wird ausschließlich während der Inferenz benötigt. An die Methode `schreibzugriffAsynchron` (siehe Listing 4.2 auf Seite 74) muss nur das Attribut `zugriff` als Ergebnis der Inferenz geliefert werden.

Ereignisse der RRML bei der asynchronen Aktualisierung

Durch das Nichtterminal `<EreignisAsynchron>` in Listing 5.1 auf Seite 117 ist definiert, auf welche Ereignisse bei der asynchronen Aktualisierung reagiert wird. Diese Ereignisse werden im Folgenden erläutert.

- `write(<Parameter>)`: Ein Schreibzugriff, der aus der Replica Queue stammt, wird vom Replikationsmanager gestellt. Hierbei handelt es sich um ein externes Ereignis, dass durch den Replikationsmanager initiiert wird. Somit werden hierüber die Startereignisse gebildet (siehe unten).

  `<Parameter>`:

  D Default-Ereignis. Mittels des Default-Parameters können Regeln für alle Objekte und Replikationseinheiten formuliert werden.

  E_i i-te Replikationseinheit. Weil es sich um ein Startereignis handelt, sind die Replikationseinheiten betroffen, die entweder das logische Objekt, dessen Replikat vom Replikationsmanager geschrieben werden soll, oder die initiierende Komponente beinhalten. Bei einem Schreibzugriff können mehrere Replikationseinheiten betroffen sein, wenn z.B. ein logisches Objekt in mehreren Replikationseinheiten auftaucht oder wenn die initiierende Komponente oder Komponenten mit Replikat zu Replikationseinheiten zusammengefasst sind (siehe unten, Startereignisse).

  O^o Das logische Objekt, dessen Replikat vom Replikationsmanager geschrieben wird.

  K_k Die initiierende Komponente, die den ursprünglichen Schreibzugriff auf das logische Objekt O^o gestellt hat (siehe Ereignis update(K_k) im Abschnitt 5.3.2).

  R_k^o Das zu schreibende Replikat.

- `later(<Parameter>)`: Für die in `<Parameter>` übergebene Einheit wurde ein Ereignis ausgelöst, das besagt, dass derzeit ein Schreibzugriff nicht erlaubt ist. Es handelt sich hierbei um ein intern ausgelöstes Ereignis, d.h. das Ereignis wurde als Aktion einer Regel ausgelöst.

  `<Parameter>`:

  D wie beim Ereignis `write()`.

  E_i wie beim Ereignis `write()`, jedoch kann eine beliebige Replikationseinheit betroffen sein.

  O^o wie beim Ereignis `write()`, jedoch kann ein beliebiges logisches Objekt betroffen sein.

  K_k Eine Komponente mit Replikat. Im Gegensatz zu `write()` sind hier nur die „Zielsysteme" gemeint, d.h. die Komponenten, die ein Replikat speichern.

  R_k^o Ein bestimmtes Replikat, für das eine Regel formuliert wird. Dies ist die feinste Granularitätsstufe.

- `one(<Parameter>)`: Analog zum Ereignis `later()` wurde für die in `<Parameter>` übergebene Einheit ein Ereignis ausgelöst, das besagt, dass derzeit ein Schreibzugriff erlaubt ist, aber nur ein Auftrag aus der Replica Queue bearbeitet werden darf. Es handelt sich hierbei ebenfalls um ein intern ausgelöstes Ereignis.

  `<Parameter>`: wie beim Ereignis `later()`.

- `all(<Parameter>)`: Analog zum Ereignis `one()` wurde für die in `<Parameter>` übergebene Einheit ein Ereignis ausgelöst, das besagt, dass derzeit ein Schreibzugriff erlaubt ist, in diesem Fall aber alle Aufträge des Replikats aus der Replica Queue bearbeitet werden dürfen. Es handelt sich hierbei ebenfalls um ein intern ausgelöstes Ereignis.

  `<Parameter>`: wie beim Ereignis `later()`.

Der Bedingungsteil der Regeln wird, wie bereits erwähnt, in Abschnitt 5.3.6 gemeinsam für alle Regelarten erläutert. Hier folgt zunächst eine Darstellung der Aktivitäten, die bei der Ausführung einer Aktion stattfinden, die innerhalb der Inferenz bei der asynchronen Aktualisierung ausgeführt werden.

Aktivitäten einer Aktion der RRML bei der asynchronen Aktualisierung

Die Aktivitäten bzw. der Ablauf einer Aktion der RRML bei der asynchronen Aktualisierung wird analog zu den Aktivitäten einer Aktion der RRML bei der synchronen Aktualisierung in Pseudo-Sprache beschrieben (siehe Abschnitt 5.3.2). Auch hier wird vorausgesetzt, dass ein Ergebnisobjekt, in diesem Fall `ErgebnisAsynchron`, mit den oben genannten Attributen existiert, auf das einfachheitshalber global zugegriffen werden kann. Das Ziel einer Aktion ist die Manipulation des Attributs `zugriff`, um zu ermitteln, ob das zu aktualisierende Replikat derzeit geschrieben werden darf. Eine Ausnahme bildet auch hier die Aktion `use_ruleset`, mit der Regeln aus einer Datei nachgeladen werden.

Ob das Attribut `zugriff` potenziell geändert wird, hängt einerseits vom Parameter der Aktion ab und andererseits vom zu aktualisierenden Replikat. Mit „potenziell ändern" ist gemeint, dass die Änderung nur dann durchgeführt wird, wenn kein Widerspruch in den Regeln auftritt bzw. ein Widerspruch so gelöst wird, dass die aktuelle Aktion ausgeführt werden darf. Die Behandlung von widersprüchlichen Regeln ist in Abschnitt 5.3.5 beschrieben. Es wird angenommen, dass R_y^z das zu schreibende Replikat ist, das zum logischen Objekt O^z gehört und auf der Komponente K_y gespeichert ist. Dann muss in Abhängigkeit des Parameters der Aktion das Attribut `zugriff` wie folgt potenziell geändert werden:

D Änderung potenziell durchführen.

E_i Wenn $O^z \in E_i$, dann Änderung potenziell durchführen.

Listing 5.3: Ablauf einer Aktion bei der asynchronen Aktualisierung

```
1  public void aktionAsynchron(Object R_y^z, Object regel) {
2
3    anhaengenRegel(regel);
4
5    if (regel.aktionsart = "use_ruleset") {
6      ladeRegelnAusDatei(regel.aktionsparameter);
7      return;
8    }
9
10   Object aendern := bestimmeAenderung(R_y^z,regel.aktionsparameter);
11
12   Object status := behandleWiderspruch(aendern,regel);
13
14   if (status = NoSolution) {
15     initiiereErgebnis();
16     throw AbbruchInferenz;
17   }
18
19   if (status = NoAction) return;
20   if (aendern) setzeAttributErgebnis(regel.aktionsart);
21   erzeugeEreignis(regel.aktionsart,regel.aktionsparameter);
22   return;
23 }
```

O^o Wenn $O^z = O^o$, dann Änderung potenziell durchführen.

K_k Wenn $K_y = K_k$, dann Änderung potenziell durchführen.

R_k^o Wenn $R_y^z = R_k^o$, dann Änderung potenziell durchführen.

Der Ablauf einer Aktion bei der asynchronen Aktualisierung ist im Listing 5.3 als Methode `actionAsynchron` dargestellt. Als Parameter wird das zu schreibende Replikat R_y^z und die Regel `regel` übergeben. Dabei wird davon ausgegangen, dass `regel` neben anderen Informationen die Aktion in `regel.aktionsart` und den zugehörigen Parameter in `regel.aktionsparameter` speichert. Der Ablauf einer Aktion bei der asynchronen Aktualisierung ähnelt dem Ablauf einer Aktion bei der synchronen Aktualisierung (siehe Listing 5.2 auf Seite 121), sodass bei der nachfolgenden Erläuterung der einzelnen Aktivitäten auf die dortige Beschreibung Bezug genommen wird:

Zeile 3 bis Zeile 7 wie Listing 5.2.

Zeile 10 Im Gegensatz zum Listing 5.2 muss hier nicht eine Liste von betroffenen Replikaten ermittelt werden, sondern in Abhängigkeit des zu schreibenden Replikats R_y^z und des Parameters der Aktion `regel.aktionsparameter` wird ermittelt, ob das Attribut `zugriff` in `ErgebnisAsynchron` potenziell geändert werden kann (siehe oben).

Zeile 12 bis Zeile 19 wie Listing 5.2, außer dass in Zeile 15 die Initialisierung die asynchrone Aktualisierung betrifft, d.h. das Attribut `zugriff` wird gemäß der Anforderung 6 auf Seite 80 mit `AlleAuftraege` initialisiert.

Zeile 20 Wenn eine potenzielle Änderung erlaubt ist, dann wird das Attribut `zugriff` von `ErgebnisAsynchron` entsprechend der aktuell ausgeführten Aktion `regel.aktionsart` manipuliert. Eine detaillierte Beschreibung folgt unten bei der Erläuterung der möglichen Aktionen der RRML.

Zeile 21 bis Zeile 22 wie Listing 5.2.

Aktionen der RRML bei der asynchronen Aktualisierung

Nachfolgend werden die Aktionen beschrieben, die bei der Inferenz zur asynchronen Aktualisierung ausgeführt werden können. Die möglichen Aktionen sind in Listing 5.1 auf Seite 117 durch das Nichtterminal `<AktionAsynchron>` definiert:

- `later(<Parameter>)`: Wenn das zu schreibende Replikat durch diese Aktion betroffen ist (siehe oben), dann wird das Attribut `zugriff` von `ErgebnisAsynchron` auf `KeinZugriff` gesetzt, d.h. derzeit ist kein Schreibzugriff erlaubt.

  `<Parameter>:`

  D Default-Aktion. Mittels des Default-Parameters wird eine Aktion definiert, die für alle Replikate ausgeführt wird.

  E_i Die Aktion wird für die i-te Replikationseinheit durchgeführt.

  O^o Die Aktion wird für das o-te logische Objekt durchgeführt.

  K_k Die Aktion wird für die k-te Komponente mit Replikat durchgeführt.

  R_k^o Die Aktion wird für das Replikat durchgeführt, das zum o-ten logischen Objekt gehört und auf der k-ten Komponente mit Replikat lokalisiert ist.

- `one(<Parameter>)`: Analog zur Aktion `later()` wird das Attribut `zugriff` auf `EinAuftrag` gesetzt, d.h. derzeit ist ein Schreibzugriff erlaubt, aber nach Bearbeitung des Auftrags muss erneut eine Inferenz durchgeführt.

  `<Parameter>:` wie bei der Aktion `later()`.

- `all(<Parameter>)`: Analog zur Aktion `later()` wird das Attribut `zugriff` auf `AlleAuftraege` gesetzt, d.h. derzeit ist ein Schreibzugriff erlaubt und es können alle Aufträge des Replikats aus der Replica Queue bearbeitet werden.

  `<Parameter>:` wie bei der Aktion `later()`.

- `use_ruleset(<FileID>)`: Eine Menge bzw. Gruppe von Regeln werden eingebunden, indem sie aus einer Datei eingelesen werden (siehe Anforderung 36 auf Seite 115).

  `<FileID>:` Dateiname des zu Grunde liegenden Dateisystems.

Startereignisse der RRML bei der asynchronen Aktualisierung:

Zum Start der InferenzAsynchron werden folgende Ereignisse als eingetreten gesetzt, wobei angenommen sei, dass das zu schreibende Replikat R_y^z sei, d.h. das zugehörige logische Objekt ist O^z und die Komponente, auf dem das Replikat lokalisiert ist, ist K_y und dass K_k die Komponente sei, die ursprünglich den Schreibzugriff auf das logische Objekt O^z initiiert hat (siehe Ereignis `update` in Abschnitt 5.3.2):

- `write(`D`)`
- `write(`E_i`)`, mit $O^z \in E_i$ oder $K_k \in E_i$.
  Anmerkung: Es können mehrere Replikationseinheiten E_i betroffen sein.
- `write(`O^z`)`
- `write(`K_k`)`
- `write(`R_y^z`)`

Auch bei der InferenzAsynchron muss nicht für jedes Startereignis eine Regel formuliert sein. Falls für ein Startereignis oder ein intern ausgelöstes Ereignis keine Regel formuliert ist, wird trivialerweise keine Aktion ausgeführt.

5.3.4. Regeln für Lesezugriffe

In Abschnitt 4.2.3 wurde das Protokoll für die Koordination von Lesezugriffen der hier vorgestellten Replikationsstrategie RegRess erläutert, wobei zwei Varianten diskutiert wurden: Beim Lesen eines beliebigen Replikats, z.B. eines lokalen Replikats, ist keine Koordination und damit Inferenz von Regeln nötig. Beim Lesen eines Replikats, das bestimmten Eigenschaften genügt, erfolgt die Ermittlung der potenziellen Replikate über die Inferenz von Regeln, mit denen passende Replikate bestimmt werden. Darüber hinausgehende Möglichkeiten, wie z.B. die Umsetzung von Votierungsverfahren mittels Regeln, werden im Ausblick betrachtet (siehe Kapitel 9).

Im Gegensatz zu den Aktionen der Regeln, die bei der Inferenz für die synchrone und asynchrone Aktualisierung spezifiziert werden können, lösen die Aktionen der Regeln, die bei der Inferenz für Lesezugriffe bestimmt werden können, in der vorliegenden Version 2.0 keine Ereignisse aus. Ansonsten gleicht sich die Semantik der Regeln, sodass analog zu den Regeln für die synchrone Aktualisierung (siehe Abschnitt 5.3.2) folgender Aufbau für die Erläuterung dient:

- Ergebnis der Inferenz bei einem Lesezugriff
- Ereignisse der RRML bei einem Lesezugriff
- Aktivitäten einer Aktion der RRML bei einem Lesezugriff
- Aktionen der RRML bei einem Lesezugriff
- Startereignisse der RRML bei einem Lesezugriff

Ergebnis der Inferenz bei einem Lesezugriff

Bei der Inferenz, die bei Lesezugriffen durchgeführt wird, wird eine unsortierte Liste von Referenzen auf Replikate ermittelt, die speziellen Eigenschaften genügen. Die ermittelten Referenzen bei der InferenzLesen (siehe Abschnitt 4.2.4) werden als Ergebnis der Methode `lesezugriff` (siehe Listing 4.3 auf Seite 77) geliefert. Auch bei der InferenzLesen wird ein Ergebnisobjekt mitgeführt, das bei den Aktionen bearbeitet wird, und nachfolgend als `ErgebnisLesen` bezeichnet wird. `ErgebnisLesen` verfügt über folgende Attribute:

- `regelKonsistenz`: Eine boolsche Variable, mit der festgehalten wird, ob eine Regel ausgeführt wurde, die die Menge der möglichen Replikate hinsichtlich der Konsistenz einschränkt (`true`) oder nicht (`false`). Das Attribut `regelKonsistenz` wird mit `false` initialisiert.
- `refKonsistenz`: Speichert eine unsortierte Liste von Referenzen auf Replikate, die den speziellen Konsistenzeigenschaften einer Regel genügen. Dieses Attribut muss mit `null` initialisiert werden, d.h. die Liste ist initial leer.
- `regelPerformance`: Eine boolsche Variable, mit der festgehalten wird, ob eine Regel ausgeführt wurde, die die Menge der möglichen Replikate hinsichtlich der Performance einschränkt (`true`) oder nicht (`false`). Das Attribut `regelKonsistenz` wird mit `false` initialisiert.
- `refPerformance`: Speichert eine unsortierte Liste von Referenzen auf Replikate, die den speziellen Performanceeigenschaften einer Regel genügen. Dieses Attribut muss mit `null` initialisiert werden, d.h. die Liste ist initial leer.
- `regeln`: Speichert eine Liste der Regeln, die bei der Inferenz berücksichtigt wurden. Hierüber können ggf. Widersprüche in den Regeln erkannt werden.

Auch hier dient das Attribut `regeln` lediglich während der Inferenz zur Erkennung von widersprüchlichen Regeln. Mit den übrigen Attributen von `ErgebnisLesen` wird das Ergebnis ermittelt, d.h. die unsortierte Liste mit Referenzen auf passende Replikate. Dabei wird wie folgt verfahren: In Widerspruch stehen zwei Regeln, wenn die Regeln die gleiche Aktionsart beinhalten (siehe Abschnitt 5.3.5). Regeln mit unterschiedlicher Aktionsart stehen nicht in Widerspruch, sondern als Ergebnis der Inferenz wird die Vereinigung der beiden Listen `refKonsistenz` bzw. `refPerformance` ermittelt, d.h. die „passenden“ Replikate sind diejenigen, die beide Kriterien erfüllen und somit in beiden Listen auftauchen. Die Ermittlung des Ergebnisses nach dieser Vorgabe ist im Listing 5.4 dargestellt:

Listing 5.4: Ermittlung der Ergebnisreferenzen

```
public List ermitteleErgebnis() {
  if (ErgebnisLesen.regelKonsistenz and
      ErgebnisLesen.regelPerformance)
    return bildeVereinigung();

  else if (ErgebnisLesen.regelKonsistenz)
    return ErgebnisLesen.refKonsistenz;

  else if (ErgebnisLesen.regelPerformance)
    return ErgebnisLesen.refPerformance;

  return null;
}
```

Zeile 2 Wenn sowohl eine Regel für eine Konsistenz- als auch für eine Performancebedingung ausgeführt wurde, dann wird in

Zeile 4 die Vereinigung der beiden Listen, die die passenden Replikate für das jeweilige Kriterium speichern, als Ergebnis geliefert.

Zeile 6 Wenn nur Regeln für die Konsistenzbedingung ausgeführt wurden, dann wird in

Zeile 7 die Liste der Replikate, die die Konsistenzbedingung erfüllen, als Ergebnis geliefert.

Zeile 9 Wenn nur Regeln für die Performancebedingung ausgeführt wurden, dann wird in

Zeile 10 die Liste der Replikate, die die Performancebedingung erfüllen, als Ergebnis geliefert.

Zeile 12 Wenn keine Regel ausgeführt wurde, dann wird eine leere Liste als Ergebnis geliefert (siehe Anforderung 11 auf Seite 81).

Angemerkt sei, dass die Verwendung der Attribute von `ErgebnisLesen` und eine Implementierung der Methode `ermitteleErgebnis()` (siehe Listing 5.4) mit fester Codierung der Aktionsarten ungünstig ist, aber hier leichter verständlich. Eine Erweiterung der RRML 2.0 um weitere Aktionsarten bei der Inferenz für Lesezugriffe würde hier eine Anpassung erforderlich machen. Besser wäre z.B. eine indirekte Indizierung.

Ereignisse der RRML bei einem Lesezugriff

Auf welche Ereignisse bei Lesezugriffen reagiert wird, ist in Listing 5.1 auf Seite 117 durch das Nichtterminal `<EreignisLesen>` definiert. Weil in der RRML, Version 2.0, keine internen Ereignisse bei der InferenzLesen erzeugt werden, gibt es nur ein externes Ereignis, das nachfolgend erläutert wird:

- `read(<Parameter>)`: Ein Lesezugriff eines Clients wird an den Replikationsmanager gestellt. Hierüber werden die Startereignisse gebildet, d.h. es handelt sich um ein externes Ereignis:

  `<Parameter>`:

  D Default-Ereignis. Mittels des Default-Parameters können Regeln für alle Objekte und Replikationseinheiten formuliert werden.

  E_i i-te Replikationseinheit. Weil es sich um ein Startereignis handelt, sind die Replikationseinheiten betroffen, die entweder das logische Objekt, auf das vom Client zugegriffen wird, oder die initiierende Komponente beinhalten. Bei einem Lesezugriff können mehrere Replikationseinheiten betroffen sein, wenn z.B. ein logisches Objekt in mehreren Replikationseinheiten auftaucht oder wenn die initiierende Komponente oder Komponenten mit Replikat zu Replikationseinheiten zusammengefasst sind (siehe unten, Startereignisse).

O^o Das logische Objekt, auf das vom Client zugegriffen werden soll.

K_k Die initiierende Komponente. Hierbei muss es sich nicht um eine Komponente mit Replikat handeln. Es soll gelten: $k = 1, .., n$ Komponente mit Replikat (n = Replikationsgrad), $k = n + 1, .., m$ Komponente ohne Replikat

R_k^o Derzeit nicht für die Ereignisart `read` benötigt. Wenn ein Client ein spezielles Replikat lesen möchte, wird keine Inferenz durchgeführt (siehe Abschnitt 4.2.3).

Auch der Bedingungsteil der Regeln für Lesezugriffe wird, wie bereits erwähnt, in Abschnitt 5.3.6 gemeinsam für alle Regelarten erläutert.

Aktivitäten einer Aktion der RRML bei einem Lesezugriff

Auch hier werden die Aktivitäten einer Aktion der RRML bei einem Lesezugriff in Pseudo-Sprache beschrieben (siehe Abschnitt 5.3.2). Es wird ebenso vorausgesetzt, dass die Ergebnisvariable `ErgebnisLesen` (siehe oben) existiert und global zugegriffen werden kann. Das Ziel der Aktion ist die Manipulation der Referenzliste, die der entsprechenden Aktionsart entspricht. Wie bereits erwähnt, wird die Ergebnisliste an Hand der jeweiligen Referenzlisten gebildet. In die Referenzliste werden die Referenzen der Replikate aufgenommen, die das entsprechende Kriterium erfüllen. Potenziell können somit alle Replikate des betreffenden logischen Objekts in Betracht kommen.

Die Aktivitäten, die bei der Ausführung einer Aktion für Lesezugriffe stattfinden, unterscheidet sich insofern von den Aktivitäten bei der Ausführung von Aktionen für Aktualisierungen (ob synchron oder asynchron), als dass hier keine internen Ereignisse ausgelöst werden (siehe Zeile 22 des Listings 5.2 auf Seite 121). Ansonsten gleicht sich der Ablauf, sodass in der nachfolgenden Erläuterung auf die wesentlichen Unterschiede eingegangen wird.

Der Ablauf einer Aktion bei einem Lesezugriff ist im Listing 5.5 als Methode `actionLesen` dargestellt, der folgende Parameter übergeben werden: Das logische Objekt O^z ist das Objekt, das gelesen werden soll. Die Regel `regel` ist die aktuell ausgeführte Regel, wobei davon ausgegangen wird, dass `regel` neben anderen Informationen die Aktion in `regel.aktionsart` und den zugehörigen Parameter in `regel.aktionsparameter` speichert. Der Parameter `eigenschaften`, der von einem Client über die Methode `lesezugriff` (siehe Listing 4.3 auf Seite 77) geliefert wird, enthält die Kriterien, die von den Replikaten des logischen Objekts O^z eingehalten werden müssen, z.B. in Form einer Liste von Zweiertupeln, wobei jedes Zweiertupel die Kriteriumart und das Kriterium beinhaltet. Ein Beispiel ist ein geforderter Versionsabstand von höchstens 3, der im Zweiertupel als (Versionsabstand,3) realisiert sein könnte. Die Aktivitäten im Listing 5.5 haben folgende Bedeutung:

Zeile 3 bis Zeile 7 wie Listing 5.2.

Zeile 10 Die zum logischen Objekt O^z gehörenden Replikate werden bestimmt. Da von einer vollreplizierten Datenbank mit Replikationsgrad n ausgegangen wird, gibt es genau n Replikate.

Zeile 12 bis Zeile 19 wie Listing 5.2, außer dass in Zeile 15 die Initialisierung den Lesezugriff betrifft, d.h. es wird gemäß der Anforderung 11 auf Seite 81 eine leere Ergebnisliste geliefert.

Zeile 21 Es wird in `ErgebnisLesen` gesetzt, dass eine bestimmte Aktionsart ausgeführt wurde, d.h. das Attribut `regelKonsistenz` bzw. `regelPerformance` von `ErgebnisLesen` wird auf `true` gesetzt.

Zeile 22 Die entsprechende Referenzliste von `ErgebnisLesen` wird als leere Liste initialisiert, d.h. das Attribut `refKonsistenz` bzw. `refPerformance` von `ErgebnisLesen` wird `null` gesetzt.

Listing 5.5: Ablauf einer Aktion bei einem Lesezugriff

```
public void aktionLesen(Object Oᶻ, Object regel
                        Object eigenschaften) {
  anhaengenRegel(regel);

  if (regel.aktionsart = "use_ruleset") {
    ladeRegelnAusDatei(regel.aktionsparameter);
    return;
  }

  List replikate := ermitteleReplikate(Oᶻ);

  Object status := behandleWiderspruch(regel);

  if (status = NoSolution) {
    initiiereErgebnis();
    throw AbbruchInferenz;
  }

  if (status = NoAction) return;

  setzeRegelausfuehrung(regel.aktionsart);
  initialisiereReferenzliste(regel.aktionsart);

  for(int i; i < n; i++) {
    if (passtReplikat(regel,eigenschaften,replikate[i]))
      anhaengen(regel.aktionsart,replikate[i]);
  }

  return;
}
```

Zeile 24 Weil es laut Annahme genau n Replikate eines logischen Objekts gibt, werden in einer Schleife n Prüfungen durchgeführt, ob das entsprechende Replikat den Anforderungen genügt.

Zeile 25 Es wird geprüft, ob das n-te Replikat dem Kriterium genügt, d.h. es wird geprüft, ob gemäß der Aktionsart inklusive Aktionsparameter eine entsprechende Eigenschaft definiert ist und diese erfüllt ist. In diesem Fall liefert die Prüfung `true`, andernfalls `false`.

Zeile 26 Wenn das Replikat dem Kriterium genügt, dann wird die Referenz auf das Replikat gemäß der Aktionsart an die entsprechende Referenzliste `refKonsistenz` bzw. `refPerformance` angehängt.

Zeile 29 Der Ablauf der Aktion wird beendet

Es sei angemerkt, dass die Berechnung, ob ein Replikat ein bestimmtes Kriterium erfüllt oder nicht, durchaus aufwändig sein kann. Die Performance in Form von Antwortzeiten könnten z.B. je Komponente mit Replikat vom Replikationsmanager protokolliert werden und somit über eine lokale Tabelle referenziert werden. Ein Versionsabstand als Konsistenzbedingung lässt sich verhältnismäßig einfach über die Replica Queue (siehe Abschnitt 4.2.2) ermitteln. Weitere Konsistenzbedingungen wie zeitlicher Abstand oder Wertdifferenz bedürfen im Allgemeinen eines Zugriffs auf das entfernte Replikat bzw. dessen Zeitstempel oder das Führen umfangreicher lokaler Kontrollinformationen, was in beiden Fällen aufwändig wäre.

Aktionen der RRML bei einem Lesezugriff

Im Folgenden werden die einzelnen Aktionen beschrieben, die bei der Inferenz zur Ermittlung der passenden Replikate für einen Lesezugriff ausgeführt werden können. Die möglichen Aktionen sind in Listing 5.1 auf Seite 117 durch das Nichtterminal `<AktionLesen>` definiert:

- `getConsistency(<Parameter>)`: Falls möglich, werden Referenzen auf Replikate ermittelt, die speziellen Konsistenzeigenschaften genügen (siehe auch Kohärenzbedingungen, Definition 22 auf Seite 37).

  `<Parameter>`:

  V Versionsabstand (Version Condition)

  T Zeitverzug (Time Condition)

  A Wertdifferenz (Arithmetic Condition)

- `getPerformance(<Parameter>)`: Falls möglich, werden Referenzen auf Replikate ermittelt, die speziellen Performanceeigenschaften genügen.

  `<Parameter>`:

  R Antwortzeit (Responsetime Condition)

  C CPU-Zeit (CPU Condition)

- `use_ruleset(<FileID>)`: Eine Menge bzw. Gruppe von Regeln werden eingebunden, indem sie aus einer Datei eingelesen werden (siehe Anforderung 36 auf Seite 115).

  `<FileID>`: Dateiname des zu Grunde liegenden Dateisystems.

Startereignisse der RRML bei einem Lesezugriff

Zum Start der InferenzLesen werden folgende Ereignisse als eingetreten gesetzt, wobei angenommen sei, dass von der Komponente K_k das logische Objekt O^z gelesen werden soll:

- `read(`Def`)`
- `read(`E_i`)`, mit $O^z \in E_i$ oder $K_k \in E_i$.
  Anmerkung: Es können mehrere Replikationseinheiten E_i betroffen sein.
- `read(`O^z`)`
- `read(`K_k`)`, wobei K_k die initiierende Komponente ist.

Auch bei der InferenzLesen muss nicht für jedes Startereignis eine Regel formuliert sein, ggf. wird keine Aktion ausgeführt.

5.3.5. Behandlung von widersprüchlichen Regeln

Zwei Regeln stehen in Widerspruch, wenn deren Aktionen zu unterschiedlichen Ergebnissen bei der Inferenz führen, d.h. wenn die Aktionen der beiden Regeln die entsprechenden Attribute der jeweiligen Ergebnisvariablen unterschiedlich manipulieren (siehe Abschnitte 5.3.2 bis 5.3.4). Das zwei Regeln potentiell in Widerspruch stehen, kann schon bei der Spezifikation der Regeln erkannt werden. Weil aber erst bei der Auswertung des Bedingungsteils einer Regel zur Laufzeit, d.h. während der Inferenz, geklärt wird, ob die Aktion der Regel ausgeführt wird, tritt der potentielle Widerspruch erst zur Laufzeit als eingetretener Widerspruch auf. Somit ist eine Behandlung zur Laufzeit notwendig. Der Widerspruch wird allgemein derart gelöst, dass eine überlebende Aktion bestimmt wird. Die Bestimmung der überlebenden Aktion basiert auf Regeln, die hier als *„Widerspruchsregeln“* bezeichnet werden.

In der Version 2.0 der RRML nehmen die Widerspruchsregeln nicht in dem Sinne an der Inferenz teil, dass sie gleichberechtigt neben den anderen Regeln stehen, sondern während der Ausführung einer Aktion der anderen Regeln in der Methode `behandleWiderspruch()` (siehe

Listing 5.6: Behandlung von widersprüchlichen Regeln

```
1  public Object behandleWiderspruch(List parameterliste) {
2
3    if (not pruefeWiderspruch(parameterliste)) return Action;
4
5    Object rangfolge := ermittleRangfolge(parameterliste);
6    if (rangfolge = NoS) return NoSolution;
7
8    Object status = vergleicheAktuelleRegel(rangfolge,parameterliste);
9    return status;
10 }
```

Zeile 12 der Listings 5.2, 5.3 bzw. 5.5) bei Auftreten eines Widerspruchs herangezogen werden, um die Rangfolge der widersprüchlichen Regeln zu bestimmen. Sie sind also nicht Regeln, die bei der Inferenz für Schreib- bzw. Lesezugriffe ausgewertet werden (siehe InferenzSynchron, InferenzAsynchron bzw. InferenzLesen im Abschnitt 4.2.4), sondern Teil der Aktionen dieser Regeln, für die eine eigene Inferenz benötigt wird, die im Folgenden als *„InferenzWiderspruch“* bezeichnet wird:

InferenzWiderspruch: Inferenz auf Basis von Widerspruchsregeln, die bei der Behandlung von widersprüchlichen Regeln durchgeführt wird und mit der die überlebende Aktion bestimmt wird.

Denkbar wäre auch, Widerspruchsregeln gleichberechtigt zu den anderen Regeln während der InferenzSynchron, InferenzAsynchron bzw. InferenzLesen auszuwerten, um mittels der Aktion der Widerspruchsregeln geeignete Lösungen zu bestimmen. Hierfür würde eine konzeptionelle Änderung benötigt, die im Ausblick (siehe Kapitel 9) diskutiert wird. Weiterhin wird an dieser Stelle der Einfachheit halber davon ausgegangen, dass Widersprüche bei den Widerspruchsregeln nicht behandelt werden, sondern bei der Behandlung von widersprüchlichen Widerspruchsregeln das Verfahren „PFG“ (siehe unten) verwendet wird. Wenn dadurch keine eindeutige Lösung bestimmt werden kann, dann ist eine Behandlung des Widerspruchs nicht möglich.

In Listing 5.6 ist die Behandlung von widersprüchlichen Regeln in Pseudo-Code erläutert. Die Methode `behandleWiderspruch()` wird innerhalb der Aktion von Regeln ausgeführt, die für die synchrone bzw. asynchrone Aktualisierung und die Koordination von Lesezugriffen spezifiziert werden können, wobei für jede der drei Inferenzen unterschiedliche Parameter übergeben werden. Somit gibt es drei Varianten der Methode `behandleWiderspruch()`. Weil die Funktionalität in den drei Fällen gleich ist, folgt eine gemeinsame Beschreibung, ohne detailliert auf die Parameterliste der unterschiedlichen Varianten einzugehen. Als Ergebnis liefert die Methode `behandleWiderspruch()` entweder `Action`, d.h. die Aktion der aktuellen Regel kann ausgeführt werden, oder `NoAction`, d.h. die Aktion der aktuellen Methode darf nicht ausgeführt werden, oder `NoSolution`, d.h. es sind widersprüchliche Regeln aufgetaucht, aber der Widerspruch konnte nicht gelöst werden. Im einzelnen werden folgende Operationen durchgeführt:

Zeile 3 Es wird an Hand der Liste der ausgeführten Regeln geprüft, ob ein Widerspruch aufgetreten ist. Wenn die aktuelle Regel nicht in Widerspruch zu einer ausgeführten Regel steht, kann deren Aktion ausgeführt werden.

Zeile 5 Mittels Inferenz wird ermittelt, ob es eine passende Widerspruchsregel gibt. Wie ermittelt wird und welche Werte berechnet werden, wird unten bei den Widerspruchsregeln erläutert.

Zeile 6 Wenn keine passende Widerspruchsregel ermittelt wurde, dann wird `NoSolution` geliefert, d.h. es gibt keine Lösung des Widerspruchs und die Inferenz wird abgebrochen (siehe Abschnitte 5.3.2 bis 5.3.4).

Zeile 8 Die aktuelle Regel wird mit der Regel verglichen, die in Widerspruch steht und ausgeführt wurde. Dabei wird ein Verfahren angewendet, dass durch das Ergebnis der InferenzWiderspruch (siehe unten) bestimmt wurde. Der Vergleich liefert `NoSolution`, falls die beiden Regeln gleiche Granularität und Priorität haben (siehe unten), d.h. es gibt keine Lösung des Widerspruchs und die Inferenz wird abgebrochen (siehe Abschnitte 5.3.2 bis 5.3.4). Er liefert `Action`, falls die aktuelle Regel gemäß dem vorgegebenen Verfahren der in Widerspruch stehenden Regel zu bevorzugen ist. Andernfalls liefert der Vergleich `NoAction`.

Zeile 9 Weil der Vergleich der widersprüchlichen Regeln (siehe Zeile 8) genau den Rückgabewert liefert, den auch die Methode `behandleWiderspruch()` selbst liefert, kann dieser Rückgabewert dem Aufrufer geliefert werden.

Bei der InferenzWiderspruch werden ausschließlich Widerspruchsregeln herangezogen. Der Aufbau der Widerspruchsregeln entspricht den Regeln für Schreib- und Lesezugriffe, sodass die Semantik analog zu den Regeln für die synchrone Aktualisierung (siehe Abschnitt 5.3.2) erläutert wird:

- Ergebnis der InferenzWiderspruch
- Ereignisse der Widerspruchsregeln
- Aktivitäten einer Aktion der Widerspruchsregeln
- Aktionen der Widerspruchsregeln
- Startereignisse der InferenzWiderspruch

Ergebnis der InferenzWiderspruch

Das Ziel der InferenzWiderspruch ist es, ein Verfahren festzulegen, mit dem widersprüchliche Regeln behandelt werden können (siehe Listing 5.6). In der Version 2.0 der RRML können derartige Widersprüche für unterschiedliche Replikationseinheiten oder logische Objekte unterschiedlich behandelt werden, indem für die entsprechenden Objekte geeignete Widerspruchsregeln spezifiziert werden. Durch das Verfahren wird festgelegt, ob die aktuelle Regel, die in Widerspruch zu einer bereits ausgeführten Regel steht, gegenüber dieser ausgeführten Regel bevorzugt zu behandeln ist oder nicht. Folgende Verfahren sowie ein ungültiges Verfahren sind in der RRML 2.0 definiert:

CGP Gröbste Granularität plus Priorität (Coarse Granularity plus Priority): Zunächst wird die Granularitätsstufe der widersprüchlichen Regeln betrachtet. Die Granularität bezieht sich auf den Parameter der Aktion einer Regel (siehe das Nichtterminal `<Parameter>` im Listing 5.1). Dabei gilt: Default D vor Replikationseinheit E_i vor logisches Objekt O^o vor Komponente K_k vor Replikat R_k^o. Wenn Regeln auf gleicher Granularitätsstufe existieren, dann werden Prioritäten herangezogen, d.h. in diesem Fall wird die Regel mit höchster Priorität gewählt. Dabei wird davon ausgegangen, dass Regeln natürliche Zahlen als Priorität zugeordnet werden können. Falls keine Priorität vergeben wurde, wird der Wert 0 gesetzt. Wenn widersprüchliche Regeln gleiche Granularitätsstufe und Priorität haben, dann kann der Widerspruch nicht behandelt werden (siehe Zeile 8 des Listings 5.6).

FGP Feinste Granularität plus Priorität (Fine Granularity plus Priority): Wie CGP, jedoch mit umgekehrter Granularitätsreihenfolge.

PCG Priorität plus gröbste Granularität (Priority plus Coarse Granularity): Zunächst wird die Priorität ausgewertet. Wenn Widerspruchsregeln gleiche Priorität haben, dann wird die Widerspruchsregel mit höchster Granularitätsstufe gewählt (siehe CGP). Wenn Widerspruchsregeln gleiche Priorität und Granularitätsstufe haben, dann kann der Widerspruch nicht behandelt werden.

PFG Priorität plus feinste Granularität (Priority plus Fine Granularity): Wie PCG, jedoch mit umgekehrter Granularitätsreihenfolge.

NoS Kein Ergebnis (No Solution): Die Inferenz auf Basis der Widerspruchsregeln lieferte kein Ergebnis bzw. kein eindeutiges Ergebnis. Die Behandlung der widersprüchlichen Regeln wird abgebrochen (siehe Zeile 6 des Listings 5.6), mit der Konsequenz, dass die ursprüngliche Inferenz (InferenzSynchron, InferenzAsynchron bzw. InferenzLesen, siehe oben) abgebrochen werden muss.

In der Version 2.0 der RRML wird auf eine Behandlung von Widersprüchen in Widerspruchsregeln einfachheitshalber verzichtet. Bei Widersprüchen innerhalb von Widerspruchsregeln wird das Verfahren PFG angewendet. Weil Widerspruchsregeln keine internen Ereignisse auslösen, kann somit während der InferenzWiderspruch die Widerspruchsregel aus der Menge der passenden Widerspruchsregeln (siehe unten, Startereignisse) ausgewählt werden, die die höchste Priorität mit feinster Granularität hat. Das Verfahren dieser Widerspruchsregel ist das Ergebnis der InferenzWiderspruch. Wenn es mehrere Widerspruchsregeln mit höchster Priorität und feinster Granularität gibt, dann liefert die InferenzWiderspruch das ungültige Verfahren NoS.

Ereignisse der Widerspruchsregeln

Das mögliche Ereignis der Widerspruchsregeln ist in Listing 5.1 auf Seite 117 durch das Nichtterminal `<EreignisWiderspruch>` definiert. Hierbei handelt es sich um das folgende externe Ereignis:

- `conflict(<Parameter>)`: Es wurden widersprüchliche Regeln bei der InferenzSynchron, InferenzAsynchron bzw. InferenzLesen erkannt, die behandelt werden müssen:

  `<Parameter>`:

  D Default-Ereignis. Mittels des Default-Parameters können Widerspruchsregeln für alle widersprüchlichen Regeln formuliert werden.

  E_i i-te Replikationseinheit. Es sind die Replikationseinheiten betroffen, die entweder das logische Objekt, auf das zugegriffen wird, oder die Komponente mit Replikat beinhalten.

  O^o Das logische Objekt, auf das zugegriffen werden soll.

  K_k Die Komponente mit Replikat, auf die zugegriffen werden soll.

  R_k^o Das Replikat, auf das zugegriffen werden soll.

In der Version 2.0 der RRML werden gleiche Widerspruchsregeln für die drei Inferenzarten InferenzSynchron, InferenzAsynchron und InferenzLesen verwendet. Wenn für die drei Inferenzarten unterschiedliche Widerspruchsregeln gewünscht werden, ist eine Erweiterung nötig, die z.B. drei unterschiedliche Ereignisse `conflictSynchron`, `conflictAsynchron` und `conflictLesen` ermöglicht. Der Bedingungsteil der Widerspruchsregeln, der den anderen Regeln gleicht, wird in Abschnitt 5.3.6 gemeinsam für alle Regelarten erläutert.

Aktivitäten einer Aktion der Widerspruchsregeln

In der Version 2.0 der RRML wird kein Ablauf bei der Aktion einer Widerspruchsregel ausgeführt. Der Aktionsteil dient lediglich zur Identifizierung des Verfahrens, das für die Widerspruchsbehandlung verwendet werden soll. Im Ausblick (siehe Kapitel 9) wird aufgezeigt, wie die Aktivitäten einer Aktion aussehen könnten, falls die Widerspruchsregeln die Ergebnisvariablen der entsprechenden Inferenzen direkt manipulieren sollen.

Aktionen der Widerspruchsregeln

Auf ein Nachladen von Regeln mit der Aktion `use_ruleset` wird bei Widerspruchsregeln verzichtet. Eine Gruppierung von Widerspruchsregeln kann innerhalb der anderen Regeln vorgenommen werden. Damit verbleibt eine Aktion, mit der ein Verfahren gesetzt wird:

- `set(<Parameter>)`: Das Verfahren, das bei der Behandlung von widersprüchlichen Regeln zum Einsatz kommt, wird ausgewählt.

  `<Parameter>`:

  CGP Das Verfahren CGP wird als Widerspruchsbehandlung gesetzt (siehe oben).

  FGP Das Verfahren FGP wird als Widerspruchsbehandlung gesetzt (siehe oben).

  PCG Das Verfahren PCG wird als Widerspruchsbehandlung gesetzt (siehe oben).

  PFG Das Verfahren PFG wird als Widerspruchsbehandlung gesetzt (siehe oben).

Startereignisse der InferenzWiderspruch
Zum Start der InferenzWiderspruch werden folgende Ereignisse als eingetreten gesetzt, wobei angenommen sei, dass entweder auf ein logisches Objekt O^z oder auf ein Replikat R_y^z zugegriffen wird (das zugehörige logische Objekt zum Replikat R_y^z ist das Objekt O^z und die zugehörige Komponente, auf dem das Replikat lokalisiert ist, ist die Komponente K_y):

- `conflict(`D`)`
- `conflict(`E_i`)`, mit $O^z \in E_i$ oder $K_y \in E_i$.
  Anmerkung: Es können mehrere Replikationseinheiten E_i betroffen sein.
- `conflict(`O^z`)`
- `conflict(`K_y`)`
- `conflict(`R_y^z`)`, wobei dieses Ereignis nur bei der asynchronen Aktualisierung ausgelöst wird oder bei der synchronen Aktualisierung, wenn der Parameter des Aktionsteils einer Regel das Replikat R_y^z betrifft.

Wenn für ein Startereignis keine Regel formuliert ist, dann wird trivialerweise keine Aktion, d.h. kein Setzen eines Verfahrens, berücksichtigt.

5.3.6. Bedingungen der Regeln

In diesem Abschnitt werden die Bedingungen in den Regeln der RRML gezeigt, die für die vier vorgestellten Regelarten (Regeln für die synchrone Aktualisierung, für die asynchrone Aktualisierung, für Lesezugriffe und für Widerspruchsregeln) gleich sind. Dabei wird nicht darauf eingegangen, ob jede Bedingung in den verschiedenen Regelarten sinnvoll ist. Die formale Spezifikation einer Bedingung ist in Listing 5.1 auf Seite 117 durch das Nichtterminal `<Bedingung>` definiert. Hier folgt die semantische Bedeutung der durch die Sprache bereitgestellten Konstrukte.

Gegenüber den Bedingungen in der Version 1.0 der RRML [Add05] werden in der Version 2.0 weitere Funktionen eingeführt, wodurch der Sprachumfang erweitert wird, um z.B. die hinzugekommenen Konsistenzbedingungen formulieren zu können. Der Aufbau einer Bedingung bleibt in der Version 2.0 gegenüber der Version 1.0 gleich. Bei einer Bedingung handelt es sich um einen logischen Ausdruck, der entweder den logischen Wahrheitswert `true` oder `false` liefert. Eine Bedingung kann mehrere Operatoren enthalten (siehe unten), sodass bei der Berechnung des Wahrheitswerts der Bedingung die Reihenfolge der Verknüpfungen von Bedeutung ist. Bei der *„Infixnotation“*, bei der ein Operator zwischen zwei Operanden steht, werden Klammern benötigt, um eine eindeutige Reihenfolge der anzuwendenden Operationen festzulegen. Bei der *„Präfixnotation“*, bei der die Operatoren vor den Operanden stehen, kann auf Klammern verzichtet werden. Da die Infixnotation leichter verständlich ist, wird sie hier bei Beispielen und bei der Erläuterung des Entwurfs der RRML genutzt. In der XML-Repräsentation (siehe Abschnitt 5.4) wird analog zur Version 1.0 die Präfixnotation verwendet.

Bei einer Bedingung handelt es sich um eine prädikatenlogische Formel [BE05, KK06]. In der RRML wird jedoch nicht die komplette Syntax der Prädikatenlogik benötigt. So werden beispielsweise nur zweistellige Prädikate verwendet und auf Quantoren verzichtet. Daher dient

folgende Beschreibung der Konstruktion einer Bedingung in der RRML, Version 2.0 (Hinweis: die tiefer gestellten Indize dienen der Unterscheidung der Konstrukte):

- Konstanten: Eine Konstante $a_i, i \in \mathbb{N}$ ist eine reelle Zahl, wobei nicht zwischen Typen unterschieden wird, oder der Wahrheitswert `true` bzw. der Wahrheitswert `false`.
- Variablen: Eine Variable $x_i, i \in \mathbb{N}$ ist Platzhalter für ein logisches Objekt, für eine Komponente, für ein Replikat oder für eine Systemkennzahl, wie z.B. für die Systemzeit.
- Funktionen: Eine Funktion $f_i, i \in \mathbb{N}$ hat höchstens einen Parameter, ist also einstellig: $f_i(x_j)$. Eine Funktion liefert einen Wert, z.B. die Stunde der Systemzeit oder einen Versionsabstand eines Replikats. Mögliche Funktionen werden unten beschrieben. Wenn der Parameter eindeutig ist, kann er weggelassen werden.
- Prädikate: Ein Prädikat $P_i, i \in \mathbb{N}$ ist entweder die Konstante `true`, die Konstante `false` oder der Vergleich einer Funktion mit einer Konstanten, d.h. es handelt sich um ein zweistelliges Prädikat in der Form $f_i(x_j) \diamond a_k$. Bei $\diamond$ handelt es sich um die bekannten Vergleichsoperatoren: $=, \neq, >, \geq, <, \leq$. An dieser Stelle wird die Typkonformität der Funktion f_i und der Konstanten a_k nicht berücksichtigt (siehe unten). Ein Prädikat hat entweder den Wert `true` oder `false`.
- Bedingungen: Eine Bedingung $B_i, i \in \mathbb{N}$ ist eine Formel, für die gilt:
  a) Ein Prädikat P_i ist eine Formel.
  b) Sind F, G Formeln, dann sind auch $\neg(F)$, $(F \wedge G)$ und $(F \vee G)$ Formeln, mit $\neg$: logische Negation, $\wedge$: logische Konjunktion und $\vee$: logische Disjunktion.

Eine Bedingung ist also die logische Verknüpfung mehrerer Prädikate. Da hier die Infixnotation gewählt wurde, sind Klammern nötig, auf die, wie bereits erwähnt, bei der Präfixnotation verzichtet werden kann. Der Parameter der Funktionen wird aus dem Kontext abgeleitet (siehe unten) und wird daher nicht angegeben. Damit kann auch analog zur Version 1.0 der RRML das Klammerpaar einer Funktion entfallen, sodass in den nachfolgenden Erläuterungen und in den Regeln die Funktionen nur mit ihrem Namen genannt sind.

Funktionen für Datum und Uhrzeit

Um Regeln abhängig vom Datum und/oder von der Tageszeit spezifizieren zu können, muss eine konstante Vorgabe mit der Systemzeit verglichen werden. Als Systemzeit wird die Systemzeit des Replikationsmanagers gesetzt. Auf die Problematik einer exakten Uhrzeit in verteilten Systemen wird an dieser Stelle nicht eingegangen (siehe Abschnitt 2.1.1). Hier reicht im Allgemeinen eine grobe Genauigkeit aus, um Aktionen zeitgesteuert auszuführen. Daher wird auch auf eine Funktion, die die Sekunden ermittelt, verzichtet. Die folgenden Funktionen, die alle eine natürliche Zahl liefern, basieren somit auf der Systemzeit des Replikationsmanagers:

- `day:` Die Funktion `day` liefert den aktuellen Tag des aktuellen Monats.
- `month:` Die Funktion `month` liefert den aktuellen Monat im Bereich [1..12].
- `year:` Die Funktion `year` liefert das aktuelle Jahr.
- `weekday:` Die Funktion `weekday` liefert den Wochentag des aktuellen Tags, wobei 1 für Montag, 2 für Dienstag, ..., 7 für Sonntag steht.
- `hour:` Die Funktion `hour` liefert die Stunde der aktuellen Uhrzeit.
- `minute:` Die Funktion `minute` liefert die Minute der aktuellen Uhrzeit.

Beispiele:

(1)	ON	*update(E_1)*	IF	$weekday > 5 \wedge hour > 11$	THEN	*async_update(E_1)*
(2)	ON	*write(E_2)*	IF	$month = 12 \wedge hour > 7$	THEN	*later(E_2)*

Das erste Beispiel ist eine Regel, die bei der synchronen Aktualisierung angewendet wird. Die Replikate der logischen Objekte, die zur Replikationseinheit E_1 gehören, werden am Wochenende ab 12:00 Uhr asynchron aktualisiert. Die zweite Regel wird bei der asynchronen Aktualisierung

berücksichtigt. Die Replikate des logischen Objekts, die zur Replikationseinheit E_2 gehören, werden im Dezember nur vor 8:00 Uhr aktualisiert, weil z.B. tagsüber das System wegen Jahresabschlussarbeiten nicht zu sehr beschäftigt werden soll.

Funktionen zur Identifikation

Aus der Version 1.0 der RRML stammen die nachfolgenden Funktionen. Sie liefern den Index der initiierenden Komponente bzw. des logischen Objekts, auf das zugegriffen werden soll. Durch die Erweiterungen der Version 2.0 der RRML in Form von Replikationseinheiten werden diese Funktionen nicht zwingend benötigt, werden aber aus Gründen der Kompatibilität beibehalten:

- `init_node`: Die Funktion `init_node` liefert den Index der Komponente, die den Zugriff initiiert hat.
- `target_node`: Die Funktion `target_node` liefert den Index der Komponente, auf die zugegriffen werden soll.
- `object`: Die Funktion `object` liefert den Index des logischen Objekts, auf das zugegriffen werden soll. Falls auf ein Replikat bei der asynchronen Aktualisierung zugegriffen wird, dann wird der Index des zum Replikat gehörenden logischen Objekts geliefert.

Beispiele:

$$(3)\quad \text{ON}\quad \textit{update}(E_3)\quad \text{IF}\quad \textit{init_node} = 1\quad \text{THEN}\quad \textit{async_update}(K_2)$$

$$(4)\quad \text{ON}\quad \textit{update}(E_4)\quad \text{IF}\quad \textit{object} = 42\quad \text{THEN}\quad \textit{async_update}(K_7)$$

Bei den Beispielen 3 und 4 handelt es sich um Regeln für die synchrone Aktualisierung. Beim Beispiel 3 werden die Replikate der zweiten Komponente asynchron aktualisiert, sofern als Ereignis ein Schreibzugriff auf logische Objekte der Replikationseinheit E_3 eingetreten ist und die Bedingung, dass die erste Komponente den Zugriff initiiert hat, erfüllt ist. Im Beispiel 4 wird auf einen Schreibzugriff der logischen Objekte der Replikationseinheit E_4 reagiert: Wenn es sich um das 42-te Objekt handelt, dann wird die siebte Komponente asynchron aktualisiert, d.h. das Replikat R_7^{42} wird asynchron aktualisiert.

Funktionen für fachliche Konsistenzbedingungen

Die Funktionen für fachliche Konsistenzbedingungen werden benötigt, um die in Abschnitt 4.3.1 aufgestellten Anforderungen als Regeln in der RRML formulieren zu können. Es wird eine positive reelle Zahl (inklusive der Null) geliefert. Die Funktionen berechnen Abstände vom aktuellen Replikat zum zugegriffenen Replikat bzw. bei der Funktion *consistency* den Konsistenzgrad der replizierten Datenbank:

- `diff_version`: Wenn auf ein einzelnes Replikat zugegriffen wird, dann liefert die Funktion `diff_version` den Versionsabstand zum aktuellen Replikat (siehe Definition 22 auf Seite 37). Wenn auf ein logisches Objekt zugegriffen wird, dann liefert die Funktion `diff_version` den größten Versionsabstand der Versionsabstände der einzelnen Replikate des logischen Objekts.
- `diff_time`: Wenn auf ein einzelnes Replikat zugegriffen wird, dann liefert die Funktion `diff_time` den Zeitverzug in Sekunden zum aktuellen Replikat (siehe Definition 22 auf Seite 37). Wenn auf ein logisches Objekt zugegriffen wird, dann liefert die Funktion `diff_time` den größten Zeitverzug der Zeitverzüge der einzelnen Replikate des logischen Objekts.
- `diff_value`: Wenn auf ein einzelnes Replikat zugegriffen wird, dann liefert die Funktion `diff_value` die Wertdifferenz zum aktuellen Replikat (siehe Definition 22 auf Seite 37). Wenn auf ein logisches Objekt zugegriffen wird, dann liefert die Funktion `diff_value` die größte Wertdifferenz der Wertdifferenzen der einzelnen Replikate des logischen Objekts.
- `period`: Die Funktion `period` liefert den Zeitverzug in Minuten zur letzten periodischen Aktualisierung der Einheit, auf der die Regel beruht.
- `consistency`: Die Funktion `consistency` liefert den Konsistenzgrad der replizierten Datenbank gemäß der Definition 40 auf Seite 93.

Beispiele:

(5a)	ON	*update(D)*	IF	*true*	THEN	*async_update(K_8)*
(5b)	ON	*write(K_8)*	IF	*diff_version* < 5	THEN	*later(K_8)*
(5c)	ON	*async_update(K_8)*	IF	*diff_version* > 4	THEN	*sync_update(K_8)*
(6a)	ON	*update(D)*	IF	*true*	THEN	*async_update(K_9)*
(6b)	ON	*write(D)*	IF	*diff_time* < 900	THEN	*later(K_9)*
(6b)	ON	*write(D)*	IF	*diff_time* ≥ 900	THEN	*one(K_9)*

Hier werden zusammenhängende Regeln in Beispielen betrachtet: Die Regel 5a bewirkt, dass die Replikate der Komponente 8 grundsätzlich asynchron aktualisiert werden sollen. Wenn die asynchrone Aktualisierung für ein Replikat der Komponente 8 aktiv ist, dann wird geprüft, ob der Versionsabstand kleiner fünf ist. Falls das der Fall ist, wird nicht aktualisiert, andernfalls werden gemäß Voreinstellung alle Aufträge des Replikats aus der Replica Queue bearbeitet. Treffen während der asynchronen Aktualisierung weitere Schreibzugriffe auf das zum Replikat gehörende logische Objekt ein, kann der Versionsabstand auch größer als fünf werden. Mit der Regel 5c wird nun festgelegt, dass ein Replikat der Komponente 8 bei einem Versionsabstand größer als vier synchron aktualisiert werden muss. Die Regeln 5a und 5c stehen in Widerspruch, der durch Priorisierung von 5c gelöst werden muss. Weil ein synchroner Zugriff nicht erlaubt ist, solange Aufträge in der Replica Queue anstehen (siehe Abschnitt 4.2.2), wird der Schreibzugriff auf das zum Replikat gehörende logische Objekt bei einem Versionsabstand von fünf abgewiesen, sodass der Versionsabstand höchstens 5 betragen kann.

Im Beispiel 6 wird der 15 minütige Zeitverzug von Aktienkursen in Medien nachgestellt. Die Regeln 6a und 6b entsprechen den Regeln 5a und 5b, wobei hier die Komponente 9 betrachtet wird und in der Bedingung ein Zeitverzug bestimmt ist. Falls der Zeitverzug kleiner als 900 Sekunden, also 15 Minuten, ist, dann erfolgt keine Aktualisierung. Mit Regel 5c ist nun gesteuert, dass bei einem Zeitverzug größer/gleich 900 Sekunden genau ein Auftrag bearbeitet werden darf. Die Regel 5c greift genau solange, bis der Zeitverzug kleiner 900 Sekunden ist und solche Änderungen noch nicht auf der Komponente 9 gesehen werden sollen.

Angemerkt sei, dass die Berechnung der Funktionen aufwändiger ist als beispielsweise die Berechnung der Zeitwerte. Der Versionsabstand und der Konsistenzgrad können mittels der Anzahl an Aufträgen in der Replica Queue bestimmt werden (siehe Abschnitt 4.3.3). Periodische Aktualisierungen, die mittels der Funktion `period` gesteuert werden, bedürfen des Protokollierens entsprechender Informationen. Bei den Funktionen `diff_time` und `diff_value` ist sogar ein Zugriff auf das Replikat bzw. dessen Zeitstempel nötig, um den Abstand zum aktuellen Replikat zu ermitteln.

Funktionen für technische Konsistenzbedingungen

Die nachfolgenden Funktionen, die ebenfalls eine positive reelle Zahl liefern, können verwendet werden, um Anforderungen für technische Konsistenzbedingungen (siehe Abschnitt 4.3.2) als Regeln in der RRML zu formulieren:

- `datasize`: Die Funktion `datasize` liefert die Datengröße in Byte des logischen Objekts bzw. des Replikats, auf das zugegriffen wird.
- `conflicts`: Die Funktion `conflicts` liefert die Anzahl der Schreib-/Lesekonflikte$_R$ in einem vom Administrator spezifizierten Zeitraum.
- `connection_throughput`: Wenn auf ein Replikat zugegriffen wird, dann liefert die Funktion `connection_throughput` den Datendurchsatz der Komponente, die das Replikat speichert. Wenn auf ein logisches Objekt zugegriffen wird, dann wird der kleinste Datendurchsatz der Komponenten ermittelt, die ein Replikat des logischen Objekts speichern. Der Datendurchsatz wird in Transaktionen pro Sekunde gemessen.

- `responsetime`: Wenn auf ein Replikat zugegriffen wird, dann liefert die Funktion `responsetime` die Antwortzeit der Komponente, die das Replikat speichert. Wenn auf ein logisches Objekt zugegriffen wird, dann wird die größte Antwortzeit der Komponenten ermittelt, die ein Replikat des logischen Objekts speichern. Die Antwortzeit wird in Millisekunden vom Absetzen einer Transaktion bis zum Erhalt der Antwort gemessen.

Beispiele:

(7)	ON	$update(E_7)$	IF	$datasize > 1024$	THEN	$async_update(E_7)$
(8)	ON	$write(D)$	IF	$responsetime > 5000$	THEN	$later(K_9)$
(9)	ON	$update(D)$	IF	$conflicts > 99$	THEN	$sync_update(D)$

Im Beispiel 7 werden bei der InferenzSynchron die Replikate der logischen Objekte, die zur Replikationseinheit E_7 gehören, den asynchron zu aktualisierenden Replikaten zugeordnet, wenn die Datengröße der entsprechenden logischen Objekte ein Kilobyte übersteigt. Die Regel 8 wird bei der asynchronen Aktualisierung herangezogen. Wenn ein Replikat, dass auf der Komponente 9 lokalisiert ist, aktualisiert werden soll, aber die Antwortzeit der Komponente derzeit schlechter als fünf Sekunden ist, dann wird die Aktualisierung nicht ausgeführt.

Für das letzte Beispiel sei angenommen, dass es Regeln gibt, die eine asynchrone Aktualisierung gestatten. Wenn nun, wie in Regel 9 gezeigt, die Anzahl der Konflikte den Wert 99 übersteigt, dann müssen alle Replikate synchron aktualisiert werden. Steht diese Regel in Widerspruch zu einer anderen Regel, ist sie ggf. höher zu priorisieren. Es kann vorkommen, dass die Replikate nicht synchron aktualisiert werden können, weil noch Aufträge in der Replica Queue anstehen (siehe Abschnitt 4.2.2), sodass diese Zugriffe erst dann durchgeführt werden können, wenn entweder die entsprechenden Aufträge aus der Replica Queue abgearbeitet sind oder die Anzahl der Konflikte geringer geworden ist. Angemerkt sei, dass die Ermittlung der Anzahl der Konflikte aufwändig ist, weil lokale Lesezugriffe auf Aktualität geprüft werden müssen.

5.3.7. Verifikation der Anforderungen

In diesem Abschnitt wird gezeigt, wie die Anforderungen an die RRML umgesetzt wurden. Dabei muss unterschieden werden, ob die Anforderungen bei der Implementierung des Regelinterpreters oder durch Sprachkonstrukte der RRML erfüllt werden. Auf die Implementierungsaspekte wird detailliert in Abschnitt 7.3.2 eingegangen. Dabei ist auch die korrekte Realisierung der Semantik der Regeln von Bedeutung, z.B. die Ausführung der Aktionen, die in den Listings 5.2 bis 5.6 beschrieben ist.

Anforderung 1 bis Anforderung 15

Diese Anforderungen bestimmen den Startzustand, den Endzustand, Reaktionen im Fehlerfall, Behandlung von Terminierung und widersprüchlichen Regeln bei den Inferenzen für die synchrone Aktualisierung, für die asynchrone Aktualisierung und für Lesezugriffe. Die Erfüllung dieser Anforderungen müssen bei der Implementierung berücksichtigt werden. Bei der Erläuterung der Semantik der Regeln in den Abschnitten 5.3.2 bis 5.3.4 wurde spezifiziert, wie die entsprechenden Ergebnisvariablen zu initialisieren sind. Die Reaktion auf widersprüchliche Regeln, kann wie gefordert, auch mittels Widerspruchsregeln (siehe Abschnitt 5.3.5) beeinflusst werden. Die Terminierung der Inferenzen wird in Abschnitt 5.5 diskutiert.

Anforderung 16: Struktur von Regeln

Die Regeln der RRML sind so gestaltet, dass pro Parameter im Ereignis- und Aktionsteil für verschiedene Einheiten (z.B. Replikationseinheiten, logische Objekte, Komponenten) Regeln spezifiziert werden können. Somit können für eine Regeleinheit mehrere Regeln formuliert werden, die genau ein Zielsystem betreffen. Die RRML erleichtert durch die Parameter die Formulierung derart, dass in einer Regel mehrere dieser Einheiten angesprochen werden können.

Anforderung 17: Aktive Regeln bei einer Inferenz
Diese Anforderung wurde in der Version 1.0 der RRML erfüllt, indem je Regel im XML-Schema (siehe Abschnitt 5.4) ein entsprechendes Attribut gesetzt werden kann. Nur aktive Regeln werden bei der Inferenz berücksichtigt.

Anforderung 18: Gültigkeitszeitraum von Regeln
Der Gültigkeitszeitraum von Regeln, genauer gesagt, der Zeitraum, in dem die Aktion einer Regel ausgeführt wird, kann im Bedingungsteil durch die Funktionen für Datum und Uhrzeit gesteuert werden (siehe Abschnitt 5.3.6).

Anforderung 19: Zustand eines Replikats
Bei der Inferenz für die synchrone Aktualisierung wird die Ergebnisvariable `ErgebnisSynchron` (siehe Abschnitt 5.3.2) geführt, die es erlaubt, die Zustände der betroffenen Replikate hinsichtlich ihrer Aktualisierungsart und ihrer Wechselfähigkeit zu manipulieren. Hierbei handelt es sich um ein Implementierungsaspekt, mit dem der Zustand eines Replikats gesetzt wird.

Anforderung 20: Kohärenzbedingungen
Die Kohärenzbedingungen, d.h. der Versionsabstand, der Zeitverzug und/oder die Wertdifferenz, sind durch die Funktionen `diff_version`, `diff_time`, `diff_value` im Bedingungsteil (siehe Abschnitt 5.3.6) realisiert.

Anforderung 21: Periodische Aktualisierung
Die periodische Aktualisierung kann mittels der Funktion `period` im Bedingungsteil (siehe Abschnitt 5.3.6) umgesetzt werden.

Anforderung 22: Konsistenzgradbedingung
Eine Grenze für den Konsistenzgrad der replizierten Datenbank kann durch die Funktion `consistency` im Bedingungsteil (siehe Abschnitt 5.3.6) bestimmt werden.

Anforderung 23: Lesen veralteter Daten
Die Spezifikation des Protokolls für die Koordination von Lesezugriffen (siehe Abschnitt 4.2.3) erlaubt, dass entweder ohne Nutzung eines Replikationsmanagers auf ein lokales Replikat zugegriffen werden kann, das eventuell veraltet ist, oder dass über den Replikationsmanager auf ein logisches Objekt zugegriffen wird, wobei der Replikationsmanager Replikate des logischen Objekts bestimmt, die vorgegebenen Eigenschaften genügen. Im letzten Fall können entsprechende Regeln formuliert werden (siehe Abschnitt 5.3.4).

Anforderung 24: Verfügbarkeit
Wenn ein Replikat wechselfähig ist, dann kann es laut Protokoll für die synchrone Aktualisierung (siehe Listing 4.1 auf Seite 71) den asynchron zu aktualisierenden Replikaten zugeordnet werden, wenn die Komponente, auf der das Replikat gespeichert ist, nicht verfügbar ist. Ob ein Replikat wechselfähig ist, kann durch Regeln festgelegt werden (siehe Abschnitt 5.3.2).

Anforderung 25: Performance bei der synchronen Aktualisierung
Im Bedingungsteil einer Regeln kann durch die Funktion `responsetime` ein Grenzwert für die Antwortzeit, die als Metrik für Performance verwendet wird, gesetzt werden (siehe Abschnitt 5.3.6). Ab dem Grenzwert wird ein Replikat den asynchron zu aktualisierenden Replikaten zugeordnet.

Anforderung 26: Performance bei Lesezugriffen
Kriterien wie Performance und Konsistenz werden bei Lesezugriffen dadurch berücksichtigt, dass Regeln mit entsprechenden Aktionen spezifiziert werden (siehe Abschnitt 5.3.4).

Anforderung 27: Datengröße
Unterschiedliche Reaktionen auf die Datengröße können mittels der Funktionen `datasize` im Bedingungsteil (siehe Abschnitt 5.3.6) ausgelöst werden.

Anforderung 28: Konfliktbedingung
Um bei einer zu hohen Anzahl von Schreib-/Lesekonflikten$_R$, die durch die asynchrone Aktualisierung bedingt ist, wieder zur synchronen Aktualisierung zurückzukehren, können Regeln formuliert werden, die die Anzahl der Konflikte mit der Funktion `conflicts` im Bedingungsteil abfragt (siehe Abschnitt 5.3.6).

Anforderung 29: Regelung von Widersprüchen
Die Behandlung von widersprüchlichen Regeln ist durch das Konzept der Widerspruchsregeln gelöst (siehe Abschnitt 5.3.5).

Anforderung 30: Vergabe von Prioritäten für Regeln
Wie bei der Anforderung 17 handelt es sich hier um ein Attribut, das einer Regel bei der Definition beigefügt werden kann (siehe Abschnitt 5.4).

Anforderung 31: Weglassen von expliziten Replikationsregeln
Bei der Inferenz werden entsprechende Vorbelegungen gesetzt (siehe Anforderungen 1 bis 15), sodass keine ungültigen Ergebnisse durch Fehlen von Regeln möglich sind. Somit können auch Regeln weggelassen werden, die den Startzustand nicht ändern.

Anforderung 32: Default-Regeln
Um Regeln für alle logischen Objekte der replizierten Datenbank zu formulieren, wird in den Regeln der Parameter D genutzt (siehe Abschnitt 5.1).

Anforderung 33: Regeln für alle Replikate einer Komponente
Um Regeln für alle Replikate einer Komponente zu formulieren, wird in den Regeln der Parameter K_k, $k = 1, 2, ..., n$ genutzt (siehe Abschnitt 5.1).

Anforderung 34: Beschreibungsmöglichkeit von Regeln
Die Beschreibungsmöglichkeit von Regeln ist bereits in der Version 1.0 durch entsprechende Attribute im XML-Schema realisiert (siehe Abschnitt 5.4).

Anforderung 35: Gruppierung von Regeln
Die Gruppierung von Regeln ist bereits in der Version 1.0 durch die Marke (Tag) `<ruleset>` realisiert (siehe Abschnitt 5.4).

Anforderung 36: Auslagerung von Regeln
Regeln können in einer separaten Datei gespeichert werden und während der Inferenz mit der Aktion `use_ruleset` geladen werden (siehe Abschnitt 5.3.1).

Anforderung 37: Allgemeine Aktualisierungsart
Das Regeln möglichst einfach derart formuliert werden können, dass für logische Objekte bzw. Replikationseinheiten alle zugehörigen Replikate gleichermaßen behandelt werden, ergibt sich durch die Ermittlung der betroffenen Replikate in Abhängigkeit des Parameters einer Aktion (siehe Abschnitt 5.3.2).

5.4. XML-Repräsentation der RRML

Um die Regeln der RRML (Replication Rule Markup Language) zu persistieren und sie zwischen Komponenten austauschen zu können, wird die Auszeichnungssprache XML (Extensible Markup Language, [MBK00]) verwendet, genauer gesagt handelt es sich bei der RRML um eine XML-basierte Sprache. XML erlaubt unter anderem die Speicherung hierarchisch strukturierter Daten in einer Baumstruktur. Somit wird in diesem Abschnitt die Transformation der konzeptionellen Syntax der RRML aus Abschnitt 5.3 in die XML-basierte RRML-Syntax gezeigt.

Bereits die Version 1.0 der RRML basierte auf XML, sodass entsprechende XML-Elemente mit den zugehörigen Tags (Kennungen) definiert wurden. Des Weiteren wurde das Mapping

(die Abbildung) beispielsweise der logischen Objekte auf die Replikate spezifiziert sowie eine zur RRML gehörige Schema-Datei entworfen, mit der die syntaktische Gültigkeit der RRML geprüft werden kann. Weil die Version 2.0 der RRML eine Erweiterung der Version 1.0 ist, die keine Änderung der XML-Struktur nach sich zieht, wird die XML-Repräsentation der RRML in dieser Arbeit nur grob vorgestellt. Für eine detaillierte Beschreibung sei auf die Diplomarbeit von Jan Stefan Addicks verwiesen [Add05].

Hier wird zunächst in Abschnitt 5.4.1 auf die XML-basierte RRML-Syntax eingegangen, wobei auf die Erweiterungen der Version 2.0 gegenüber der Version 1.0 fokussiert wird. Im Abschnitt 5.4.2 wird das Mapping der logischen Objekte, Replikationseinheiten und Komponenten auf XML-basierte Elemente betrachtet. Abschließend wird in Abschnitt 5.4.3 kurz auf die Erweiterung des RRML-Schemas eingegangen, die für Version 2.0 nötig ist.

5.4.1. XML-basierte RRML-Syntax

Gegenüber der Version 1.0 ändert sich die XML-basierte RRML-Syntax in der Version 2.0 nur in den XML-Elementen, die die Beschreibung der Ereignisse, der Bedingungen, der Aktionen und der Parameter erlauben. Bei den XML-Elementen für den Ereignis- und Aktionsteil sind weitere Ereignis- und Aktionsarten hinzugekommen, ohne die Struktur dieser Elemente zu verändern. Weil es in den Bedingungen in der Version 2.0 neue Funktionen gibt, können innerhalb der entsprechenden XML-Elemente einer Bedingung die neuen Funktionen als Tags verwendet werden. Die Parameter, mittels derer ein Ereignis neben der Ereignisart identifiziert wird, wurde vor allem um die Replikationseinheiten erweitert.

Im Listing 5.7 sind die Elemente und Attribute der RRML dargestellt, wobei die Hierarchie durch Einrücken kenntlich gemacht worden ist. Es handelt sich nicht um ein Beispiel, sondern um eine Auflistung der Elemente mit ihren Attributen und Unterelementen in Baumstruktur. Die Attribute sind im Listing 5.7 in Anführungszeichen gesetzt, wobei ein Text einen Hinweis für einen möglichen Wert gibt und Punkte innerhalb der Anführungszeichen für einen beliebigen Wert stehen. Tags, die mit dem Sonderzeichen „!“ beginnen, dienen als Platzhalter für „echte“ Tags, die nachfolgend erläutert werden:

Zeile 1 wie Version 1.0: Eine RRML-Datei wird als XML-Datei deklariert. Das Attribut `version` zeigt die Version der RRML an. Zusätzlich kann die Zeichenkodierung als Attribut angegeben werden.

Zeile 2 wie Version 1.0: `<rrml>` ist das Wurzelelement. Als Attribute werden die Versionsnummer und die zugehörige Schemadatei angegeben.

Zeile 3 wie Version 1.0: Mit `<ruleset>` werden Regeln logisch gruppiert. Eine Regelmenge wird mit dem Attribut `id` identifiziert.

Zeile 4 wie Version 1.0: Mit dem optionalen Tag `<title>` kann ein Bezeichner für die Regelmenge vergeben werden.

Zeile 5 wie Version 1.0: Mit dem optionalen Tag `<description>` kann eine Beschreibung zur Regelmenge erstellt werden.

Zeile 7 wie Version 1.0: Jede Regelmenge muss mindestens eine Regel der RRML in der entsprechenden Version enthalten. Mit dem Attribut `id` wird ein Identifikator vergeben. Das optionale Attribut `active` definiert, ob die Regel verwendet werden soll (siehe Anforderung 17 auf Seite 85). Das neue, optionale Attribut `priority` erlaubt die Vergabe von Prioritäten, die bei der Widerspruchsbehandlung verwendet werden (siehe Anforderung 30 auf Seite 90).

Zeile 8 wie Version 1.0: Mit dem optionalen Tag `<title>` kann ein Bezeichner für die Regel vergeben werden.

Listing 5.7: XML-basierte RRML-Syntax

```
<?xml version="2.0" encoding="UTF-8"?>
<rrml version="2.0" xmlns="Schemadatei">
  <ruleset id="Identifikator">
    <title>"..."</title>
    <description>"..."</description>

    <rule id="..." active="..." priority="...">
      <title>"..."</title>
      <description>"..."</description>

      <event>
        <!Ereignisart>
          <!Parameter>"..."</!Parameter>
        </!Ereignisart>
      </event>

      <condition>
        <!LogischerOperator>
          <assert op="...">
            <!Funktion>"..."</!Funktion>
          </assert>
        </!LogischerOperator>
      </condition>

      <action>
        <!Aktionssart>
          <Parameter>"..."</Parameter>
        </!Aktionsart>
      </action>

    </rule>
  </ruleset>
</rrml>
```

Zeile 9 wie Version 1.0: Mit dem optionalen Tag `<description>` kann eine Beschreibung zur Regel erstellt werden.

Zeile 11 wie Version 1.0: Das `<event>`-Element kapselt den Ereignisteil einer Regel und darf je `<rule>`-Element nur einmal auftreten.

Zeile 12 Gegenüber der Version 1.0 sind in der Version 2.0 weitere Ereignisarten hinzugekommen, d.h. anstelle von `!Ereignisart` kann jedes Terminal der RRML-Grammatik eingesetzt werden, das sich von den Nichtterminalen `<EreignisSynchron>`, `<EreignisAsynchron>`, `<EreignisLesen>`und `<EreignisWiderspruch>` ableiten lässt (siehe Listing 5.1 auf Seite 117). Das Element muss genau einmal auftreten.

Zeile 13 Weil die Tags der Parameter für Ereignisteil und Aktionsteil gleich sind, werden sie gemeinsam unten beschrieben.

Zeile 14 bis Zeile 15: Endetags der entsprechenden Starttags.

Zeile 17 wie Version 1.0: Das `<condition>`-Element kapselt den Bedingungsteil einer Regel und ist optional, d.h. bei Fehlen des Bedingungsteils wird gesetzt, dass die Bedingung der Regel immer erfüllt ist.

Zeile 18 wie Version 1.0: Anstelle von `!LogischerOperator` können die logischen Operatoren eingesetzt werden, d.h. gültige Tags sind `<and>`, `<or>` und `<not>`. Der logische Ope-

rator ist in der RRML-Syntax als Präfix-Operand notiert. Das Element kann fehlen, wenn die Bedingung nur aus einem Prädikat besteht (siehe Abschnitt 5.3.6). Das XML-Element für den logischen Operator, d.h. der Bereich `<!LogischerOperator>` bis `</!LogischerOperator>` kann mehrfach und verschachtelt auftreten, je nachdem, welche Bedingung formuliert werden soll.

Zeile 19 wie Version 1.0: Das `<assert>`-Tag kapselt ein Prädikat (siehe Abschnitt 5.3.6), wobei der arithmetische Operator als Attribut `op` vorangestellt wird. Mögliche Operatoren sind: EQ (entspricht =), NEQ ($\neq$), GT ($>$), GE ($\geq$), LT ($<$), LE ($\leq$).

Zeile 20 Gegenüber der Version 1.0 können in der Version 2.0 weitere Funktionen verwendet werden, d.h. anstelle von `!Funktion` kann jedes Terminal der RRML-Grammatik eingesetzt werden, das sich von dem Nichtterminal `<Funktion>` ableiten lässt (siehe Listing 5.1 auf Seite 117). Das Element muss genau einmal auftreten.

Zeile 21 bis Zeile 23: Endetags der entsprechenden Starttags.

Zeile 25 wie Version 1.0: Das `<action>`-Element kapselt den Aktionsteil einer Regel und darf je `<rule>`-Element nur einmal auftreten.

Zeile 26 Gegenüber der Version 1.0 sind in der Version 2.0 weitere Aktionsarten zugelassen, d.h. anstelle von `!Aktionsart` kann jedes Terminal der RRML-Grammatik eingesetzt werden, das sich von den Nichtterminalen `<AktionSynchron>`, `<AktionAsynchron>`, `<AktionLesen>`und `<AktionWiderspruch>` ableiten lässt (siehe Listing 5.1 auf Seite 117). Das Element muss genau einmal auftreten und gemäß der RRML-Grammatik zur Ereignisart passen.

Zeile 27 Weil die Tags der Parameter für Ereignisteil und Aktionsteil gleich sind, werden sie gemeinsam unten beschrieben.

Zeile 28 bis Zeile 33: Endetags der entsprechenden Starttags.

Jedem Ereignis und jeder Aktion ist ein Parameter beigestellt, wobei in beiden Fällen gleiche Werte zulässig sind. Die möglichen Parameter sind in Abschnitt 5.1 genannt. Gegenüber der Version 1.0 wurden in der Version 2.0 Tags für „Default“ und Replikationseinheiten eingeführt, sodass folgende XML-Elemente anstelle von `<!Parameter>`“...“`</!Parameter>` in den Zeilen 13 und 27 des Listings 5.7 treten können:

D `<default />`

E_i `<unit>i</unit>`

O^o `<object>o</object>`

K_k `<node>k</node>`

R_k^o `<replica><object>o</object><node>k</node></replica>`

Das XML-Element für ein Replikat setzt sich aus den XML-Elementen für ein Objekt und für eine Komponente zusammen, wobei jedes der beiden XML-Elemente genau einmal auftreten muss. Aus Kompatibilitätsgründen zur Version 1.0 wird in der Version 2.0 für eine Komponente das Tag `<node>`, eigentlich Knoten, beibehalten. Weil für den Default-Parameter kein Index nötig ist, handelt es sich um ein so genanntes Empty-Tag. Das Mapping der Objekte und Komponenten auf die Indize wird im Abschnitt 5.4.2 diskutiert.

Die Behandlung von Widersprüchen erfolgte in der Version 1.0 der RRML nicht wie in der Version 2.0 über Widerspruchsregeln, sondern durch *„Ausnahmebehandlung“*, d.h. es wurde beim Auftreten widersprüchlicher Regeln eine *„Exception“* geworfen, die abgefangen und behandelt werden muss. Die XML-Elemente zur Beschreibung der Ausnahmebehandlung wurden in der Version 2.0 eliminiert.

Listing 5.8: Mapping mittels XML

```
<?xml version="2.0" encoding="UTF-8"?>
<mapping version="2.0" xmlns="Schemadatei">
  <node id="Laufende␣Nummer">"Identifikator"</node>
  <object id="Laufende␣Nummer">"Identifikator"</object>
  <unit id = "Laufende␣Nummer">
    <node>"Index"</node>
    <object>"Index"</object>
  </unit>
</mapping>
```

5.4.2. Mapping in der RRML

In der Version 1.0 der RRML wurde von Jan Stefan Addicks eine Mapping-Datei [Add05] auf Basis der XML spezifiziert, die eine Abbildung der logischen Objekte auf die Replikate ermöglicht. Dabei wird davon ausgegangen, dass es sich nicht um eine voll-replizierte Datenbank, sondern um eine partitiell-replizierte Datenbank handelt. Somit ist eine Zuordnung erforderlich gewesen, die aufzeigt, auf welchen Knoten (hier Komponenten) die Replikate eines logischen Objekts gespeichert sind.

In dieser Arbeit wird einfachheitshalber angenommen, dass es sich um eine voll-replizierte Datenbank handelt und es n Komponenten mit Replikat gibt (siehe Abschnitt 4.1.1), d.h. zu jedem logischen Objekt existiert ein Replikat auf jedem der n Komponenten. Daher kann hier auf die Zuordnungstabelle aus der Version 1.0 verzichtet werden. In der Version 2.0 wurden als Erweiterung Replikationseinheiten eingeführt, sodass eine Zuordnung von logischen Objekten und Komponenten zu einer Replikationseinheit erforderlich ist (siehe Definition 35 auf Seite 65).

Replikate, logische Objekte, Komponenten und Replikationseinheiten werden in dieser Arbeit über Indizes identifiziert, die entweder hoch- oder tiefgestellt sind (siehe Abschnitt 4.1.2). Diese Notation wird sowohl bei der Konzeption der Replikationsstrategie RegRess als auch beim Entwurf der RRML verwendet. Praxistauglicher ist im Allgemeinen die Verwendung von eindeutigen Namen als Identifikator, was allerdings hier die Ausführungen erschweren würde. Ein weiteres Problem ist es, dass insbesondere die Identifikation der logischen Objekte und den zugehörigen Replikaten vom Datenspeichersystem, z.B. Dateisystem, relationale oder objektorientierte Datenbank, und von der Granularität, z.B. bei einer relationalen Datenbank Tupel oder Tabelle, abhängt. Der Form halber wird in diesem Abschnitt grob eine auf XML-basierte Möglichkeit des Mappings in Listing 5.8 gezeigt.

Zeile 1 wie Version 1.0: Eine Mapping-Datei wird als XML-Datei deklariert. Das Attribut `version` zeigt die Version der RRML an. Zusätzlich kann die Zeichenkodierung als Attribut angegeben werden.

Zeile 2 wie Version 1.0: `<mapping>` ist das Wurzelelement. Als Attribute werden die Versionsnummer und die zugehörige Schemadatei angegeben.

Zeile 3 Mit dem Tag `<node>` werden die Komponenten durchnummeriert. Bei den Komponenten kann es sich um Komponenten mit Replikat oder um Komponenten ohne Replikat handeln. Mit „Identifikator“ sind plattformabhängige Verweise gemeint, ohne ins Detail zu gehen.

Zeile 4 Mit dem Tag `<object>` werden die logischen Objekte durchnummeriert. Mit „Identifikator“ sind plattformabhängige Verweise gemeint, ohne ins Detail zu gehen.

Zeile 5 Das Tag `<unit>` kapselt eine Replikationseinheit. Das Attribut `id` dient zur laufenden Nummerierung.

Zeile 6 Das XML-Element `<node>` ist optional und kann mehrfach auftreten. Es kennzeichnet die zu einer Replikationseinheit gehörenden Komponenten über deren Indize.

Zeile 7 Das XML-Element `<object>` ist optional und kann mehrfach auftreten. Es kennzeichnet die zu einer Replikationseinheit gehörenden logischen Objekte über deren Indize.

Zeile 8 bis Zeile 9: Endetags der entsprechenden Starttags.

Weil eine Identifizierung der Komponenten und logischen Objekte, wie bereits erwähnt, abhängig von der Systemlandschaft etc. ist, wird in den Zeilen 3 und 4 auf eine detaillierte Spezifikation verzichtet. In dieser Arbeit wird bei der Implementierung und Simulation (siehe Kapitel 7) davon ausgegangen, dass die Komponenten und logischen Objekte nummeriert sind.

5.4.3. RRML-Schema

Ein XML-Schema [vdV02] definiert mittels einer Schemasprache, die selbst auf XML basiert, die Strukturen und Datentypen eines XML-Dokuments. Ein konkretes XML-Schema wird als XML-Schema-Definition (XSD) bezeichnet und erlaubt die Überprüfung der syntaktischen Gültigkeit des zugehörigen XML-Dokuments. Das XML-Schema, mit dem die Syntax eines RRML-Dokuments überprüft werden kann, wird hier als RRML-Schema bezeichnet. Bei Änderungen und/oder Erweiterungen der RRML muss das RRML-Schema angepasst werden. Daher enthält ein RRML-Schema auch eine zu den RRML-Dokumenten konforme Versionsnummer, wie im Allgemeinen jedes XML-Schema.

In der Diplomarbeit von Jan Stefan Addicks [Add05] wurde das RRML-Schema in der Version 1.0 ausführlich erläutert. Gegenüber der Version 1.0 müssen die in Abschnitt 5.4.1 genannten Erweiterungen eingebracht werden, d.h. die neuen Ereignis- und Aktionsarten sowie die neuen Funktionen und Parameter müssen analog zu den bestehenden Strukturen definiert werden. Dabei ergeben sich in der Version 2.0 strukturell keine Änderungen zur Version 1.0. In dieser Dissertation wurde auf eine Erstellung des RRML-Schemas für die Version 2.0 verzichtet, weil bei der Evaluierung die ON-IF-THEN-Darstellung von Regeln verwendet wird (siehe Abschnitt 7.3.2).

5.5. Analyse und Bewertung der RRML

Zunächst erfolgt in diesem Abschnitt eine Analyse der RRML (Replication Rule Markup Language) und der darauf basierenden Inferenz, d.h. es wird diskutiert, ob ein korrekter Ablauf der Inferenz der Regeln möglich ist und ob die Inferenz zu einem brauchbaren Ergebnis führt. Dabei muss einerseits die Sprache, d.h. die RRML, als solche betrachtet werden, andererseits die Implementierung eines Regelinterpreters (siehe Abschnitt 6.2), der Regeln der RRML auswertet. Weil an dieser Stelle keine Implementierungsaspekte betrachtet werden sollen, werden stattdessen die Anforderungen an die Inferenz und die RRML herangezogen, die ein Regelinterpreter zu erfüllen hat.

Im Forschungsbereich der aktiven Datenbanken [WC95] wurden verschiedene Analysemethoden diskutiert, die z.B. Terminierung oder Verhalten so genannter „aktiver Regeln“ (Reaktionsregeln, siehe Abschnitt 5.1) untersuchen. Weil es sich bei den Regeln der RRML ebenfalls um Reaktionsregeln handelt, können die entsprechenden Analysemethoden und -ansätze aufgegriffen werden. In [AWH92] werden die drei folgenden Eigenschaften genannt, die bei der Inferenz von Reaktionsregeln in aktiven Datenbankmanagementsystemen vorhanden sein müssen:

- Terminierung
- Konfluenz
- Beobachtbarer Determinismus

„Beobachtbarer Determinismus“ (Observable Determinism [AHW95]) bedeutet, dass die unterschiedliche Ausführungsreihenfolge von Regeln nicht zu unterschiedlichen Datenbankzuständen führen darf, die beobachtet werden können. Dabei können auch Nebenläufigkeitsanomalien (siehe Abschnitt 2.2.3) ins Spiel kommen, wenn die Aktionen von Regeln schon während der Inferenz sichtbar werden. Weil bei der Inferenz auf Basis der RRML nicht auf gemeinsame Daten zugegriffen wird, sondern auf exklusive „Ergebnisvariablen“ (siehe Abschnitt 5.3), muss die Eigenschaft „Beobachtbarer Determinismus“ nicht weiter untersucht werden. Daher wird nachfolgend im Abschnitt 5.5.1 auf Terminierung und im Abschnitt 5.5.2 auf Konfluenz eingegangen. Abschließend erfolgt in Abschnitt 5.5.3 eine Bewertung der RRML und der zugehörigen Inferenz.

5.5.1. Terminierung

Terminierung bedeutet, dass die Inferenz in endlicher Zeit einen Endzustand erreicht, wobei unter Endzustand hier die Ermittlung eines geeigneten Ergebnisses verstanden wird. Aus Sicht der in dieser Arbeit vorgestellten Replikationsstrategie RegRess ist diese Eigenschaft von großer Bedeutung, weil andernfalls Zugriffe auf die Replikate dauerhaft blockiert würden, bzw. so lange blockiert sind, bis der Regelinterpreter und/oder die Regeln „korrigiert“ sind. Daher wurden in Abschnitt 4.2.4 auch Anforderungen aufgestellt, die eine Terminierung ggf. durch Zeitablauf fordern und in diesem Fall ein initiales Ergebnis setzen. Damit ist zwar gewährleistet, dass keine Blockierung auftritt, die eigentliche Ursache ist aber nicht behoben.

Nichtterminierung kann allgemein entweder dann auftreten, wenn kein Endzustand erreicht wird, oder dann, wenn Regeln sich zyklisch in Endlosschleifen aktivieren. Einen Endzustand nicht zu erreichen, ist bei der Inferenz der RRML-Regeln nicht relevant, weil nicht zwischen Zuständen unterschieden wird. Die Inferenz der RRML-Regeln operiert auf Ergebnisvariablen (siehe Abschnitte 5.3.2 bis 5.3.4), wobei jede Aktionsausführung einer RRML-Regel einen gültigen Endzustand hinterlässt.

Das endlose, zyklische Ausführen von Anweisungen wird auch als Livelock oder als Endlosschleife bezeichnet. Bei der Inferenz von Regeln treten derartige Zyklen dann auf, wenn Aktionen Ereignisse von Regeln auslösen, die bereits ausgeführt wurden, und dadurch ein endloser Kreislauf entsteht. Auch so genannte Schlingen, d.h. eine Regel aktiviert sich selbst, führt zu Livelocks, wenn keine Maßnahmen ergriffen werden. Um Livelocks zu erkennen, bedient man sich allgemein der Graphentheorie, z.B. indem man einen so genannten Triggering-Graphen [YKC99] aufstellt und nach Zyklen untersucht.

Im Regelinterpreter, Version 1.0, von Jan Stefan Addicks [Add05] wird ein solcher Triggering-Graph zur Laufzeit der Inferenz aufgestellt. Dieser Triggering-Graph wird auf Zyklen untersucht, wobei ein Algorithmus nach [MH00] verwendet wird. Wenn keine Zyklen auftreten, dann terminiert die Inferenz. Angemerkt sei, dass die Umkehrung nicht immer gilt, d.h. die Existenz von Zyklen muss nicht unbedingt Nichtterminierung nach sich ziehen. Sicherheitshalber werden derartige Ausführungen aber im Allgemeinen abgebrochen.

Die Erkennung von Nichtterminierung mittels Triggering-Graph und Zykluserkennung ist zeitaufwändig und muss bei jedem Zugriff durchgeführt werden. Ein anderes Verfahren, Terminierung bei der Inferenz sicher zu stellen, ist das einmalige Ausführen von Regeln. Dieses Verfahren kann insbesondere dann angewendet werden, wenn sich die Fakten während der Inferenz nicht ändern. Die Aktionen der RRML-Regeln ändern die Fakten, z.B. Kennzahlen der Systemlandschaft wie Performance, nicht. Wenn man voraussetzt, dass während einer Inferenz keine neuen Kennzahlen aufgenommen oder berücksichtigt werden, dann ist das einmalige Ausführen von Regeln ein adäquates Mittel. Weil in der Version 2.0 der RRML eine Liste der ausgeführten Regeln während der Inferenz geführt wird, mit der widersprüchliche Regeln erkannt werden, können hierüber auch Wiederholungen erkannt werden und somit das wiederholte Ausführen von Regeln mit dem Rückgabewert `NoAction` unterbunden werden (siehe Listing 5.6 auf Seite 133).

Angemerkt sei, dass so genannte *„Deadlocks“*, d.h. eine Verklemmung von Prozessen auf Grund gegenseitigem Blockieren von Betriebsmitteln, nicht auftreten. Die Inferenzen unterschiedlicher

Prozesse werden isoliert ausgeführt, wobei exklusive Ergebnisvariablen verwendet werden. Kennzahlen, die z.B. im Bedingungsteil benötigt werden, werden ausschließlich gelesen, sodass keine Sperren oder ähnliches nötig sind.

5.5.2. Konfluenz

Mit Konfluenz ist in aktiven Datenbanken gemeint, dass unabhängig von der Ausführungsreihenfolge der Reaktionsregeln der gleiche Endzustand der Datenbank erreicht wird. Übertragen auf den hier betrachteten Fall bedeutet das, dass die Ergebnisvariablen (siehe Abschnitte 5.3.2 bis 5.3.4), die bei der Inferenz manipuliert werden, am Ende der Inferenz unabhängig von der Ausführungsreihenfolge der RRML-Regeln den gleichen Wert speichern. Entsprechend den drei Teilen einer Reaktionsregel, also dem Ereignis-, Bedingungs- und Aktionsteil, werden nachfolgend drei Fälle betrachtet, die zu unterschiedlichen Ergebnissen in Abhängigkeit der Ausführungsreihenfolge führen könnten:

1. In Abhängigkeit des Zeitpunktes wird eine Regel, dessen Ereignis eingetreten ist, bei einer Inferenz berücksichtigt oder nicht.
2. In Abhängigkeit des Zeitpunktes wird die Bedingung einer Regel bei einer Inferenz unterschiedlich ausgewertet.
3. Die Aktionen zweier Regeln, die die Ergebnisvariable bei einer Inferenz unterschiedlich manipulieren, werden in verschiedener Reihenfolge ausgeführt.

zu 1.) Eine Regel, deren Ereignis eingetreten ist, wird bei der Inferenz entweder dann nicht berücksichtigt, wenn das Ereignis nachträglich negiert wird oder wenn die Regel erst nachträglich bekannt wird. Bei der Inferenz auf Basis der RRML werden eingetretene Ereignisse nicht zu einem späteren Zeitpunkt z.B. durch eine Aktion verändert. Daher werden bei Eintreten eines Ereignisses alle Regeln als passend identifiziert, deren Ereignisteil dem Ereignis entspricht (siehe Abschnitt 5.3.1). Die passenden Regeln werden, wie gehabt, dann ausgeführt, wenn auch die Bedingung erfüllt ist.

Regeln, die zum Zeitpunkt der Ermittlung der passenden Regeln ausgelagert waren und später mittels der Aktion `use_ruleset` geladen werden, könnten bei der Ermittlung unberücksichtigt bleiben. Im Regelinterpreter der Version 1.0 [Add05] wird diesem Umstand dadurch begegnet, dass zunächst alle zugehörigen Regeln ausgeführt werden, die Regeln nachladen, bevor die eigentliche Inferenz beginnt. Alternativ kann eine Liste der eingetretenen Ereignisse geführt werden, die auf nachgeladene Regeln ausgeführt wird. Somit spielt der Zeitpunkt des Eintretens eines Ereignisses bzw. die Ermittlung der passenden Regeln keine Rolle.

zu 2.) Eine Regel, dessen Ereignis eingetreten ist und als passend markiert wurde, wird dann ausgeführt, wenn die Bedingung zum Zeitpunkt der Ausführung erfüllt ist. Weil in einer Bedingung über die Funktionen z.B. Kennzahlen der Systemlandschaft wie Performance berücksichtigt werden, ist der Zeitpunkt der Auswertung der Bedingung relevant. Hier soll angenommen werden, dass die Inferenz wenig Zeit beansprucht, sodass Änderungen der zugehörigen Kennzahlen vernachlässigt werden können, ggf. können die benötigten Kennzahlen während der Inferenz „eingefroren“ werden.

zu 3.) Zwei Regeln, deren Aktionen eine Ergebnisvariable unterschiedlich manipulieren, stehen in Widerspruch und aktivieren die Widerspruchsbehandlung (siehe Abschnitt 5.3.5). Da die Behandlung von Widersprüchen eine der in Widerspruch stehenden Regeln als überlebende Regel bestimmt oder einen Abbruch der Inferenz nach sich zieht, ist die Ausführungsreihenfolge der beiden Regeln nicht relevant. Angemerkt sei, dass die Manipulation der Ergebnisvariablen durch die Aktion einer Regel unabhängig vom Zeitpunkt ist, weil es sich um absolute und nicht um relative Änderungen handelt.

5.5.3. Bewertung

Die RRML (Replication Rule Markup Language) bietet umfangreiche Möglichkeiten mittels Reaktionsregeln die Koordination von Zugriffen auf die Replikate einer replizierten Datenbank zu beeinflussen. Dabei können Regeln auf Grund der Parameter, die im Ereignis- und Aktionsteil zu verwenden sind, auf unterschiedlichster Granularitätsstufe spezifiziert werden. Über die Parameter kann ebenfalls unterschiedliches Zugriffsverhalten in Abhängigkeit der beteiligten Komponenten und der zugegriffenen Daten selbst gesteuert werden. Die verschiedenen Funktionen, die in den Bedingungen verwendet werden können, erlauben die Spezifikation von Regeln für fachliche und technische Konsistenzanforderungen. Allerdings kann die Auswertung der Funktionen durchaus zeitaufwändig sein.

Durch die RRML und die zugehörige Inferenz, d.h. durch die spezifizierte Semantik und die definierten Protokolle, ist gewährleistet, dass Anfragen an einen entsprechenden Regelinterpreter terminieren und eindeutige Ergebnisse geliefert werden, die brauchbar sind. Weil sich während der Inferenz die Fakten nicht wesentlich ändern, ist eine Sicherstellung der Terminierung durch einmaliges Ausführen der Regeln kostengünstiger als eine Prüfung von Zyklen durch einen Triggering-Graphen.

Als Speicherungsformat für RRML-Regeln und zugehörigen Metainformationen wurde eine XML-basierte Syntax definiert. Dadurch ist vor allem Austauschbarkeit zwischen Komponenten bzw. einem verteiltem Replikationsmanager gewährleistet. Die Spezifikation von Regeln direkt in der XML-basierten RRML-Syntax ist allerdings komplex, sodass die Nutzung eines Regeleditors, der z.B. die Formulierung von Regeln in der ON-IF-THEN-Darstellung erlaubt, vorzuziehen ist.

Teil III.

Software-Entwurf und Evaluation

6. Replikationsmanager KARMA

Der Replikationsmanager KARMA (**K**onfigurierbarer, **a**daptiver **R**eplikations**ma**nager) realisiert die Replikationsstrategie RegRess (siehe Kapitel 4) und erreicht damit eine konfigurierbare, adaptive Replikation für heterogene, autonome Informationssysteme. In diesem Kapitel wird eine Softwarearchitektur des KARMA vorgestellt, wobei eine detaillierte Spezifikation der benötigten Komponenten erfolgt. In Kapitel 7 wird anschließend eine Implementierung auf Basis der Java Enterprise Edition (Java EE [JBC+06]) beschrieben.

Bei RegRess erfolgt die Koordination der Zugriffe durch die Inferenz von Regeln, die für den gegebenen Anwendungsbereich von einem Administrator spezifiziert werden müssen. Deswegen benötigt der KARMA einen Regelinterpreter. Da die Regeln im Wesentlichen für Eigenschaften der beteiligten Systeme spezifiziert werden, muss der KARMA bei der Verarbeitung eines Zugriffs bestimmte Messwerte protokollieren, z.B. die Antwortzeiten der Systeme bei genau diesem Zugriff. Damit stehen Kennzahlen zur Verfügung, die bei der nächsten Verarbeitung eines Zugriffs herangezogen werden können, um auf Grund der Regeln z.B. einen Wechsel zwischen synchroner und asynchroner Aktualisierung zu initiieren (siehe Abschnitt 4.2).

Da der KARMA in heterogenen, autonomen Systemlandschaften zum Einsatz kommt, ist neben der Replikationsstrategie die Transaktionsverarbeitung von besonderem Interesse. Gerade eine ordnungsgemäße Verarbeitung bei der Aktualisierung der Replikate im Sinne der Transaktionsverarbeitung ist unerlässlich, um eine konsistente Verarbeitung in verteilten Systemen zu gewährleisten.

In den folgenden Abschnitten wird der Entwurf des KARMA und seine Funktionsweise vorgestellt. In Abschnitt 6.1 wird die Softwarearchitektur des KARMA gezeigt. Anschließend wird in Abschnitt 6.2 die Funktionsweise des KARMA beschrieben, in dem einzelne Komponenten spezifiziert werden. Auf die Transaktionsverarbeitung des KARMA wird im abschließenden Abschnitt 6.3 eingegangen.

6.1. Softwarearchitektur KARMA

Der konfigurierbare, adaptive Replikationsmanager KARMA ist eine Komponente (siehe Definition 32 auf Seite 60), die die in dieser Dissertation entwickelte Replikationsstrategie RegRess realisiert. In Abbildung 4.1 auf Seite 61 ist ein einfaches Beispiel einer Systemumgebung dargestellt, die einen Replikationsmanager in allgemeiner Form als externe, zentrale Komponente zeigt. Insbesondere die Schnittstellen, die ein Replikationsmanager anbietet, um z.B. einen Zugriff auf ein logisches Datenobjekt zu ermöglichen, und die ein Replikationsmanager benötigt, um z.B. die Zugriffe auf die Replikate zu koordinieren, sind von Interesse, so auch beim KARMA, bei dem weitere Schnittstellen z.B. für die Transaktionsverarbeitung hinzukommen.

An dieser Stelle wird zunächst der ARM (**A**daptiver **R**eplikations**m**anager [NHHT03]) als Vorgänger des KARMA vorgestellt, der im Rahmen dieser Arbeit in der Diplomarbeit [Hül02] von Michael Hülsmann entwickelt und implementiert wurde. Der Schwerpunkt der Diplomarbeit lag auf die Anbindung heterogener Informationssysteme mit besonderem Augenmerk auf die Transaktionsverarbeitung. Dabei wird unterschieden, ob die Komponenten mit Replikat (in der Diplomarbeit als Systeme mit Replikat bezeichnet) das XA-Protokoll (siehe Abschnitt 2.2.2) unterstützen oder nicht und ob auf die beteiligten Datenbanken direkt oder über Anwendungsschnittstellen (API) zugegriffen wird. Die in der Diplomarbeit implementierte Replikationsstrategie kann als „frühe" Version der hier vorgestellten Replikationsstrategie RegRess aufgefasst

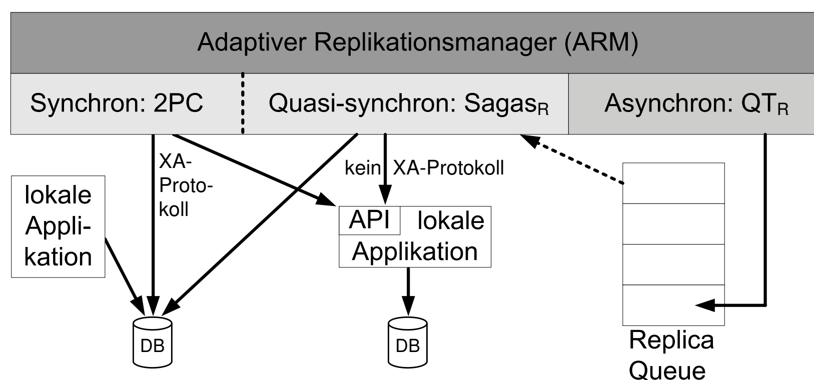

Abbildung 6.1.: Die Architektur des adaptiven Replikationsmanagers (ARM, [NHHT03])

werden. Der ARM verwendet einfache Regeln, die nicht in der RRML (siehe Kapitel 5) formuliert sind. Sie werden auch nicht in einem XML-Format gespeichert, sondern in einer relationalen Datenbank. Über den ARM werden ausschließlich Schreibzugriffe koordiniert. Lesezugriffe erfolgen grundsätzlich lokal. Die Funktionsweise der Koordination von RegRess ist aber berücksichtigt, d.h. die zu aktualisierenden Replikate werden in synchron und asynchron zu aktualisierende Replikate partitioniert.

In Abbildung 6.1 ist die Architektur des ARM dargestellt, die die Transaktionsschicht und die Zugriffsschicht zeigt. Die Komponenten mit Replikat, die synchron aktualisiert werden, müssen innerhalb einer globalen Transaktion geändert werden, um Inkonsistenzen zu vermeiden. Weil in einer heterogenen Landschaft Komponenten auftreten, die das XA-Protokoll unterstützen bzw. nicht unterstützen, verwendet der ARM die Transaktionskonzepte 2PC (siehe Abschnitt 2.2.2) bzw. Sagas_R (siehe Abschnitt 2.2.5). Gegenüber dem ursprünglichen Transaktionskonzept Sagas [GMS87] kennzeichnet der Index R in Sagas_R, dass es sich hierbei um eine spezielle Variante für den ARM handelt. Die Teiltransaktionen der Sagas_R-Transaktion sind ausschließlich so genannte *„lokale Transaktionen“* (siehe Abschnitt 2.2.1). Weiterhin werden bei Sagas_R die Teiltransaktionen nicht verkettet, sondern überlappend ausgeführt. Dabei werden die Commit-Punkte der Teiltransaktionen zeitlich nahe beieinander gelegt, wodurch im Erfolgsfall eine *„quasi-synchrone“* Verarbeitung erreicht wird (siehe auch Abschnitt 6.3.1).

Für beide Varianten der Transaktionsverarbeitung kann je nach Zielsystem entweder direkt auf die zugrunde liegende Datenbank zugegriffen werden, d.h. parallel zur lokalen Applikation, oder der Zugriff erfolgt über Schnittstellen der lokalen Applikation. Für die Systeme, die asynchron aktualisiert werden, wird ein Änderungsauftrag in eine Warteschlange (Replica Queue) eingetragen. Steht der Änderungsauftrag zur Bearbeitung an (gestrichelte Linie in Abbildung 6.1), so wird das betreffende System aktualisiert und der Auftrag aus der Replica Queue gelöscht. Hierbei handelt es sich ebenfalls um eine verteilte Transaktion, die über die *„Synchronkomponente“* verarbeitet wird, d.h. es wird entweder das Transaktionskonzept 2PC oder Sagas_R genutzt.

In diesem und dem nächsten Abschnitt liegt der Fokus mehr auf der Softwarearchitektur des eigentlichen Replikationsmanagers, also auf die Komponente, die die Replikationsstrategie RegRess realisiert, und weniger auf das Transaktionssystem. Zwar wird für eine Implementierung von RegRess eine komplexe Transaktionsverarbeitung benötigt, die über die Funktionalität einfacher Transaktionssysteme hinausgeht, dennoch wird an dieser Stelle angenommen, dass ein derartiges Transaktionssystem zur Verfügung steht. In Abschnitt 6.3 folgt eine Spezifikation der benötigten Funktionalität des Transaktionssystems sowie eine kurze Beschreibung einer Implementierung auf Basis der Java Enterprise Edition, die im Zusammenhang mit dieser Dissertation in der Diplomarbeit von Andreas Austing [Aus06] entwickelt wurde. In der Diplomarbeit wurde eine Infrastruktur für erweiterte Transaktionskonzepte (siehe Abschnitt 2.2.5) spezifiziert und implementiert.

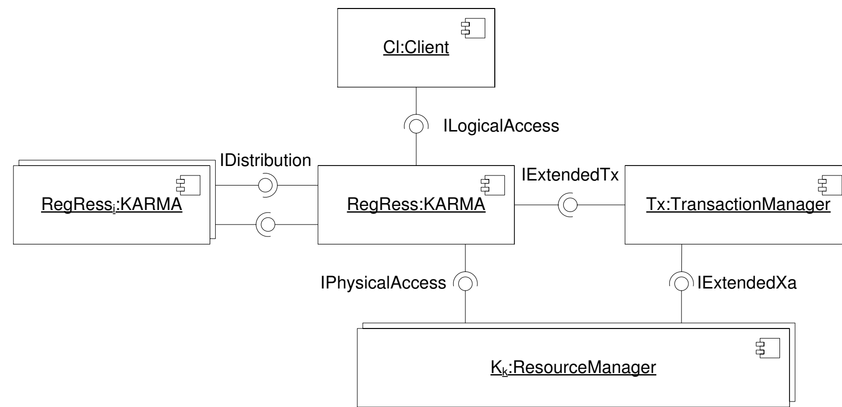

Abbildung 6.2.: Grob-Architektur des KARMA

Ein Beispiel für eine mögliche Systemumgebung mit einem zentralen Replikationsmanager ist in Abbildung 4.1 auf Seite 61 zu sehen, wobei dort auf eine Darstellung einer eigenen Transaktionskomponente verzichtet wurde. In dem Beispiel wird illustriert, wie Replikate auf verschiedenen Komponenten mit Replikat bzw. Rechnern lokalisiert sein können und wie ein Schreibzugriff eines Clients auf ein logisches Objekt über den Replikationsmanager auf die physischen Objekte, die Replikate, umgesetzt wird. Der Replikationsmanager bietet somit eine Schnittstelle an, um auf logische Objekte zugreifen zu können, und benötigt Schnittstellen auf die Komponenten mit Replikat, um die zugehörigen Replikate adressieren zu können. Neben diesen Schnittstellen wird für eine separate Transaktionsverarbeitung eine Schnittstelle zu einem Transaktionssystem und für eine Verteilung des Replikationsmanagers Schnittstellen zu den dezentralen Komponenten benötigt.

In Abbildung 6.2 ist eine grobe Softwarearchitektur des KARMA dargestellt, die insbesondere die Schnittstellen des Replikationsmanagers zeigt, der die Replikationsstrategie RegRess implementiert. Dabei ist die Architektur an das X/Open-Modell eines zentralisierten Transaktionssystems angelehnt (siehe Abbildung 2.1 auf Seite 16), wobei der KARMA nach dem X/Open-Modell ein Anwendungsprogramm ist, also die Anwendung, die auf die Ressourcen und den Transaktionsmanager zugreift. Demzufolge wird hier auf eine explizite Darstellung der Lokalisation der Replikate verzichtet und analog zu dem X/Open-Modell angenommen, dass jede Komponente mit Replikat einen Ressourcenmanager vorgeschaltet hat, der entsprechende Schnittstellen für Zugriffe auf die Replikate anbietet.

Wie in Abbildung 6.2 zu sehen ist, bietet die KARMA-Komponente bzw. eine Instanz der KARMA-Komponente mit dem Namen RegRess die Schnittstelle `ILogicalAccess` an, über die ein Client logische Zugriffe, d.h. Zugriffe auf logische Objekte, durchführt. In dieser Arbeit beginnt jeder Name einer Schnittstelle mit einem „I“ wie Interface. Der KARMA benötigt eine Schnittstelle `IExtendedTx` zu einem Transaktionsmanager, der die verteilten Transaktionen bzw. Teiltransaktionen synchronisiert. Da gegenüber dem X/Open-Modell die dortige TX-Schnittstelle erweitert wurde, wird hier von der Schnittstelle `IExtendedTx` gesprochen. Die wesentlichen Methoden der Schnittstelle sind den Beginn und das Ende von Transaktionen sowie, als Erweiterung, den Beginn und das Ende von Teiltransaktionen mitzuteilen. Der Transaktionsmanager steuert die jeweiligen lokalen Transaktionen, die auf den Komponenten mit Replikat durchgeführt werden, über die erweiterte XA-Schnittstelle, d.h. ein Ressourcenmanager einer Komponente mit Replikat muss die Schnittstelle `IExtendedXa` anbieten. Des Weiteren bietet ein Ressourcenmanager die Schnittstelle `IPhysicalAccess`, worüber der KARMA Schreib- oder Lesezugriffe auf die jeweiligen Replikate einer Komponente mit Replikat abwickelt.

Wenn es sich um einen verteilten KARMA handelt, d.h. der Replikationsmanager ist mehrfach auf verteilten Systemen vorhanden, dann müssen gemeinsame Daten verteilt werden (siehe Ab-

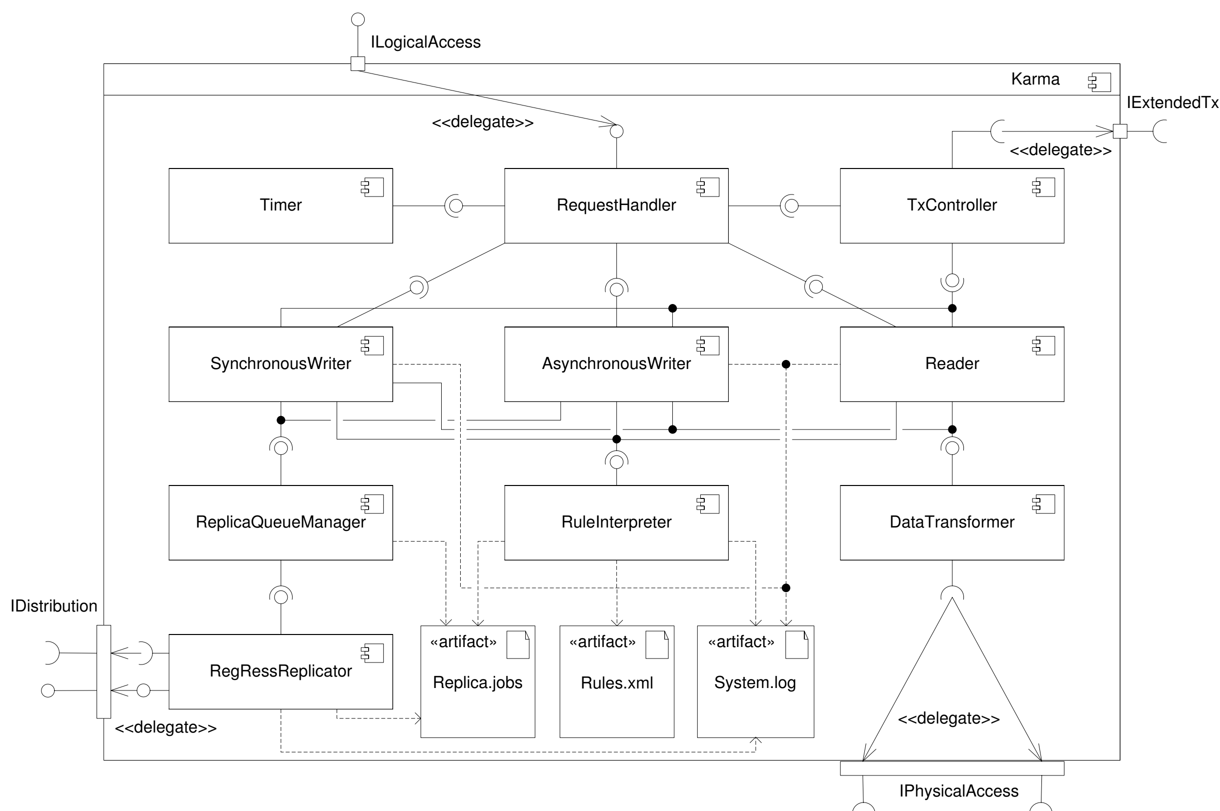

Abbildung 6.3.: Feingranulare Softwarearchitektur des KARMA

schnitt 4.2.5). Dafür benötigt der KARMA die Schnittstellen `IDistribution` zu anderen Replikationsmanagern, die die Replikationsstrategie RegRess implementieren (in der Abbildung 6.2 sind diese durch den Index i im Instanzennamen gekennzeichnet). Daher bietet ein KARMA eine entsprechende Schnittstelle an, um gemeinsame Daten entgegenzunehmen. Bevor eine detaillierte Spezifikation der Schnittstellen in Abschnitt 6.2 folgt, wird eine Dekomposition der KARMA-Komponente vorgenommen, um die benötigten Funktionen feingranularer zuordnen zu können.

In Abbildung 6.3 ist die feingranulare Softwarearchitektur der KARMA-Komponente präsentiert, wobei das UML-Komponentendiagramm die inneren Komponenten und die benötigten Artefakte zeigt. Die externen Schnittstellen (siehe Abbildung 6.2 auf Seite 155) des KARMA werden an die Schnittstellen der inneren Komponenten delegiert. Die Schnittstellen `IPhysicalAccess` und `IDistribution` sind als so genannter komplexer Port angelegt, weil es sich hierbei um Schnittstellen zu mehreren, unterschiedlichen Ressourcenmanagern bzw. zu mehreren, unterschiedlichen Replikationsmanagern handelt. In der Abbildung 6.3 wurde bei den Schnittstellen der inneren Komponenten übersichtlichkeitshalber auf eine Namensgebung verzichtet. Die inneren Komponenten des KARMA und die zugehörigen Artefakte erfüllen folgende Aufgaben:

- Der **RequestHandler** bildet eine einheitliche Schnittstelle für Clients zum KARMA gemäß dem Entwurfsmuster *„Fassade“* [GHJ04]), d.h. er nimmt Schreib- und Lesezugriffe auf logische Objekte von Clients entgegen, bearbeitet sie und liefert einen Rückgabewert.
- Der **Timer** dient als Taktgeber für den Start der asynchronen Aktualisierung, wofür eine Zeitperiode z.B. über eine Konfigurationsdatei festgelegt wird.
- Der **TxController** übernimmt die Kommunikation zu einem Transaktionsmanager (siehe Abbildung 6.2 auf Seite 155). Die Anforderungen an den Transaktionsmanager werden in Abschnitt 6.3 diskutiert.

- Der **SynchronousWriter** realisiert den synchronen Schreibzugriff auf ein logischen Objekt, wofür die Replikate des logischen Objekts in synchron und asynchron zu aktualisierende Replikate partitioniert werden.
- Der **AsynchronousWriter** realisiert den asynchronen Schreibzugriff, d.h. die in der Replica Queue anstehenden Aufträge, die die zeitversetzt zu aktualisierenden Replikate beinhalten, werden bearbeitet.
- Der **Reader** realisiert den Lesezugriff auf ein logisches Objekt, sofern ein Client einen Lesezugriff über den KARMA ausführt und nicht ein beliebiges Replikat, z.B. ein lokales Replikat, liest. Beim Lesen über den KARMA kann ein Replikat mit bestimmten Eigenschaften ausgewählt werden.
- Der **RuleInterpreter** bietet den Komponenten SynchronousWriter, AsynchronousWriter und Reader eine Schnittstelle, um mittels Inferenz auf den Replikationsregeln die betroffenen Replikate für die Zugriffe zu ermitteln (siehe Abschnitt 4.2.4).
- Der **ReplicaQueueManager** verwaltet die Aufträge der Replica Queue, d.h. die Liste der noch asynchron zu aktualisierenden Replikate.
- Der **RegRessReplicator** dient der Kommunikation zu anderen Instanzen des KARMA, sofern eine Verteilung vorliegt. Über die jeweiligen Komponenten RegRessReplicator werden die gemeinsamen Daten kommuniziert.
- Der **DataTransformer** ist für Kommunikation mit den Ressourcenmanagern der Komponenten mit Replikat verantwortlich.
- Das Artefakt **Replica.jobs** speichert die Aufträge der zeitversetzt durchzuführenden Aktualisierungen in Form der Replica Queue.
- Das Artefakt **Rules.xml** speichert die Regeln, die bei der Inferenz berücksichtigt werden.
- Das Artefakt **System.log** speichert Kennzahlen des verteilten Systems, die bei der Inferenz abgefragt werden können, z.B. die aktuelle Performance einer Komponente mit Replikat.

Weiterhin wird laut Annahme (siehe Abschnitt 4.1.1) beim Einsatz des KARMA davon ausgegangen, dass beim Start eine voll-replizierte, verteilte Datenbank in einem konsistenten Zustand vorliegt, d.h. es herrscht Replikationskonsistenz (siehe Definition 17 auf Seite 34). Daher wurde bei der Modellierung auf eine „Initialisierungskomponente“ verzichtet, die z.B. die Hinzunahme einer neuen Komponente mit Replikat in das verteilte System erlaubt. Eine derartige Initialisierung einer neuen Komponente mit Replikat könnte z.B. durch das Anlegen entsprechender Aufträge in die Replica Queue realisiert werden. Falls eine partiell-replizierte Datenbank vorliegt, dann muss bei der Koordination, d.h. bei der Inferenz, die Lokalisation von Replikaten eines logischen Objekts berücksichtigt werden. Entsprechende Mapping-Informationen wurden in der Version 1.0 der RRML spezifiziert (siehe Abschnitt 5.4.2).

Weil die hier entwickelte Replikationsstrategie RegRess Schreib-/Lesekonflikte$_R$ toleriert und Schreib-/Schreibkonflikte$_R$ vermeidet (siehe Abschnitt 4.2.5), wird kein Konfliktmanagement benötigt. Wenn z.B. in dem Protokoll für synchrone Schreibzugriffe (siehe Listing 4.1 auf Seite 71) auf den Sperrmechanismus verzichtet wird und somit Schreib-/Schreibkonflikte$_R$ auftreten können, dann wird eine Komponente für das Konfliktmanagement benötigt, die entsprechende Konflikte erkennt und behandelt (siehe Abschnitt 3.3.2).

6.2. Komponentenspezifikation KARMA

In diesem Abschnitt erfolgt die Spezifikation der „inneren“ Komponenten des KARMA. Da Komponenten ihr Verhalten im Wesentlichen über die angebotenen und die benötigten Schnittstellen definieren (siehe Definition 32 auf Seite 60), wird hier auf die Beschreibung dieser Schnittstellen fokussiert. Ein Überblick über die Komponenten sowie die zugehörigen Kommunikationswege mittels Schnittstellen bietet die feingranulare Softwarearchitektur in der Abbildung 6.3 auf Seite 156. Nachfolgend werden die einzelnen Komponenten mit ihren implementierten Schnittstel-

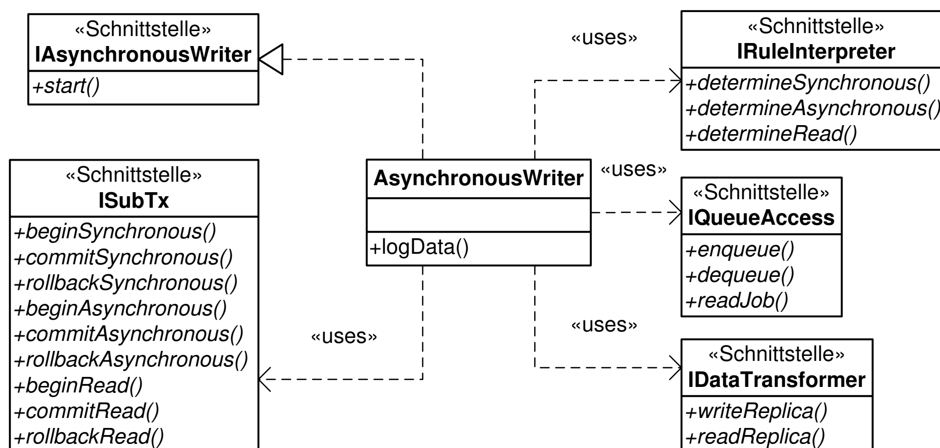

Abbildung 6.4.: Komponente AsynchronousWriter

len, also den angebotenen Schnittstellen, bzw. den extern benötigten Schnittstellen in folgender, alphabetischer Reihenfolge beschrieben:

- `AsynchronousWriter` mit `IAsynchronousWriter`
- `DataTransformer` mit `IDataTransformer` und `IPhysicalAccess`
- `Reader` mit `IReader`
- `RegRessReplicator` mit `IDistribution` und `IQueueDistribution`
- `ReplicaQueueManager` mit `IQueueAccess`
- `RequestHandler` mit `ILogicalAccess` und `IRequestHandler`
- `RuleInterpreter` mit `IRuleInterpreter`
- `SynchronousWriter` mit `ISynchronousWriter`
- `Timer`
- `TxController` mit `INestedTx`, `ISubTx` und `IExtendedTx`

Wie bereits erwähnt, beginnen die Namen von Schnittstellen mit einem „I“ für Interface. In den Abbildungen 6.4 bis 6.13 sind die Komponenten als Klassendiagramm der UML [RJB04, RHQ+05] dargestellt. Der Einfachheit halber wird jede Komponente durch eine Klasse mit gleichem Namen repräsentiert, die Schnittstellen anbietet und benötigt. Dabei werden Schnittstellen als Klassen ohne Attribute mit dem Stereotypen `<<Schnittstelle>>` gekennzeichnet. Die angebotenen Schnittstellen werden durch die Klasse implementiert (erkennbar durch den „Implementierungspfeil“) und die benötigten Schnittstellen werden von der Klasse genutzt (erkennbar durch den „Abhängigkeitspfeil“ und den Stereotypen `<<uses>>`). Gegenüber der Abbildung 6.3 auf Seite 156, in der die Schnittstellen als so genannte Lollipops dargestellt sind, wird hier die Klassendarstellung gewählt, um die zu implementierenden Methoden einer Schnittstelle zu zeigen. Auf die Parameterliste und die Rückgabewerte in den Methoden wurde hier aus Platzgründen verzichtet.

Komponente AsynchronousWriter

Der `AsynchronousWriter` in Abbildung 6.4 ist für die asynchrone Aktualisierung der Replikate zuständig, wofür er die in der Replica Queue anstehenden Aufträge abarbeitet. Dafür implementiert er die Methode `start()` der von ihm angebotenen Schnittstelle `IAsynchronousWriter`. Die Methode `start()` realisiert den asynchronen Schreibzugriff gemäß des Listings 4.2 auf Seite 74. Die Bearbeitung wird vom RequestHandler über die Methode `start()` der Schnittstelle `IAsynchronousWriter` angestoßen. Weil alle Informationen, die zum Bearbeiten eines Auftrags benötigt werden, im Auftrag selbst gespeichert sind, werden dem `AsynchronousWriter` keine weiteren Informationen in Form von Parametern in der Methode `start()` mitgegeben. Die Me-

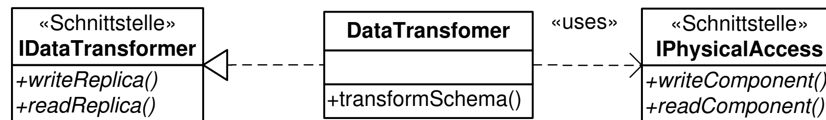

Abbildung 6.5.: Komponente DataTransformer

thode `start()` liefert als Rückgabewert einen Bearbeitungsstatus der Replica Queue, um ggf. einem Client ein Ergebnis zu liefern (siehe `RequestHandler`).

Der `AsynchronousWriter` benötigt seinerseits die Schnittstelle `IQueueAccess` zur Komponente `ReplicaQueueManager`, um einen Auftrag aus der Replica Queue mit der Methode `readJob()` zu lesen und mittels der Methode `dequeue()` bei erfolgreicher Aktualisierung zu löschen. Ob das zu aktualisierende Replikat derzeit geschrieben werden darf, wird durch Inferenz ermittelt, wofür der `AsynchronousWriter` die Methode `determineAsynchronous()` der Schnittstelle `IRuleInterpreter` nutzt. Für die Aktualisierung eines Replikats benötigt der `AsynchronousWriter` die Methode `writeReplica()` der Schnittstelle `IDataTransformer`, die von der Komponente `DataTransformer` angeboten wird.

Bei dem Schreibzugriff auf das Replikat und dem Entfernen eines Auftrags aus der Replica Queue handelt es sich um eine verteilte Transaktion, die vom `TxController` synchronisiert werden muss (siehe Abschnitt 2.2.5). Daher teilt der `AsynchronousWriter` dem `TxController` den Beginn und das Ende der Bearbeitung eines Auftrags als Transaktionsklammer über die Methoden `beginAsynchronous()` sowie `commitAsynchronous()` für das erfolgreiche Beenden bzw. `rollbackAsynchronous()` für das Zurückrollen beim Beenden mit. Diese benötigten Methoden werden von der Schnittstelle `ISubTx` bereitgestellt, die vom `TxController` angeboten wird.

Die Methode `logData()` des `AsynchronousWriter`, die explizit in Abbildung 6.4 angegeben ist, dient zum Zugriff auf das Artefakt `System.log`. Nach jeder Bearbeitung eines Auftrags werden hier entsprechende Informationen abgelegt, z.B. die Performance der betroffenen Komponente.

Komponente DataTransformer

Der `DataTransformer` in Abbildung 6.5 ist für die Kommunikation mit den Ressourcenmanagern der Komponenten mit Replikat verantwortlich und bietet dem `SynchronousWriter` sowie dem `AsynchronousWriter` die Methode `writeReplica()` zum Schreiben eines Replikats und dem `Reader` die Methode `readReplica()` zum Lesen eines Replikats. Die Methode `writeReplica()` benötigt als Parameter die Referenz zum Replikat, z.B. in Form der speichernden Komponente K_k und des logischen Objekts O^o, sowie die zu schreibenden Werte. Bei Fehlschlagen der Operation wird eine Ausnahme geworfen (siehe Listing 4.1 auf Seite 71). Gleiches gilt für die Methode `readReplica()`, wobei hier keine neuen Werte als Parameter geliefert werden müssen, sondern die Werte des gelesenen Replikats zurückgeliefert werden.

Wenn verschiedene lokale Schemata der Komponenten mit Replikat vorliegen, dann führt der `DataTransformer` auch die Datentransformation durch, wofür die Methode `transformSchema()` des `DataTransformer` dient. In diesem Fall müssen geeignete Schematransformationen [Con02] berücksichtigt werden. Ohne Beschränkung der Allgemeinheit wird in dieser Arbeit auf eine Datentransformation verzichtet (siehe Abschnitt 4.1.1), sodass die Methode `transformSchema()` nur konzeptionell angedacht ist.

Der eigentliche physikalische Zugriff auf die Replikate wird an die Ressourcenmanager delegiert, d.h. der `DataTransformer` benötigt jeweils eine Schnittstelle `IPhysicalAccess` zu einem der Ressourcenmanager, um einen Schreibzugriff mittels der Methode `writeComponent()` oder einen Lesezugriff mittels der Methode `readComponent()` durchzuführen. Weil diese Schnittstellen bzw. deren Methoden im Allgemeinen plattformabhängig sind, wird hier auf eine genaue Spezifikation verzichtet.

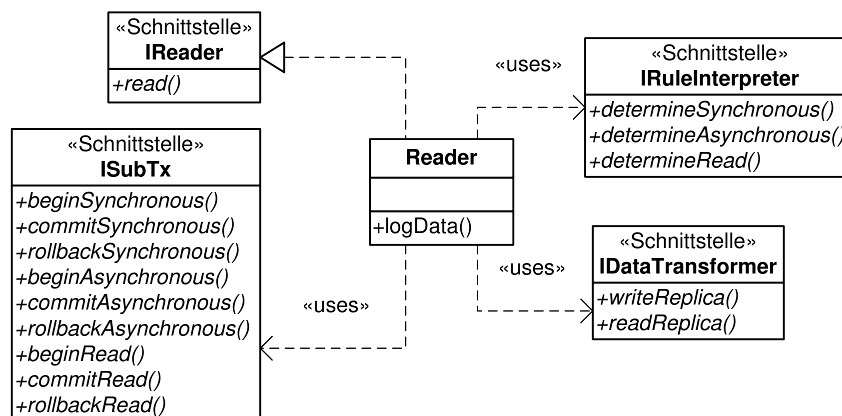

Abbildung 6.6.: Komponente Reader

Komponente Reader

Der `Reader` in Abbildung 6.6 realisiert den Lesezugriff auf ein logisches Objekt, sofern ein Client einen Lesezugriff über den KARMA ausführt und nicht ein beliebiges Replikat, z.B. ein lokales Replikat, liest (siehe Abschnitt 4.2.3). Der Lesezugriff über den KARMA gestattet die Vorgabe von Eigenschaften, die ein zu lesendes Replikat erfüllen muss. Dafür bietet der `Reader` die Schnittstelle `IReader` an, dessen Methode `read()` einen Lesezugriff gemäß des Listings 4.3 auf Seite 77 implementiert. Die Methode `read()` benötigt die im Listing genannten Parameter und liefert im Erfolgsfall das gelesene Replikat, andernfalls einen Fehlercode.

Um zu den geforderten Eigenschaften passende Replikate zu bestimmen, benötigt der `Reader` die Methode `determineRead()` der Schnittstelle `IRuleInterpreter`, die vom `RuleInterpreter` angeboten wird. Diese Methode liefert eine Liste passender Replikate bzw. eine leere Liste, falls kein Replikat passt. Der physikalische Zugriff auf eines der passenden Replikate geschieht über die Methode `readReplica()` der Schnittstelle `IDataTransformer`, die von der Komponente `DataTransformer` angeboten wird.

Ein Lesezugriff wird in einer geschachtelten, verteilten Transaktion als Teiltransaktion eingebettet (siehe `RequestHandler` bzw. `TxController`). Daher teilt der `Reader` dem `TxController` den Beginn über die Methode `beginRead()` mit und das Ende der Teiltransaktion über die Methode `commitRead()` für das erfolgreiche Beenden bzw. über die Methode `rollbackRead()` für das Zurückrollen beim Beenden. Diese benötigten Methoden werden von der Schnittstelle `ISubTx` bereitgestellt, die vom `TxController` angeboten wird.

Die Methode `logData()` des `Readers`, die explizit in Abbildung 6.6 angegeben ist, dient zum Zugriff auf das Artefakt `System.log`. Nach jedem Lesezugriff werden hier entsprechende Informationen abgelegt, z.B. die Performance der Komponente des gelesenen Replikats.

Komponente RegRessReplicator

Der `RegRessReplicator` in Abbildung 6.7 dient der Kommunikation zu anderen Instanzen des KARMA, sofern eine Verteilung vorliegt. Über die jeweiligen Komponenten `RegRessReplicator` der einzelnen Instanzen des KARMA werden die gemeinsamen Daten kommuniziert, wofür die externe Schnittstelle `IDistribution` genutzt wird: Dabei handelt es sich einerseits um die angebotene Schnittstelle, die von der Komponente selbst angeboten wird, und andererseits um die benötigten Schnittstellen zu anderen Komponenten. Jeder `RegRessReplicator` eines jeden KARMA implementiert die Methoden `sendLogInformation()` und `sendReplicaJobs` der Schnittstelle `IDistribution`.

Im Fall der angebotenen Methode `sendLogInformation()` können neue Systemkennzahlen, wie z.B. die Performance einer Komponente mit Replikat, die als Parameter übergeben werden müssen, von anderen Komponenten empfangen werden. Die empfangenen Daten werden mit der

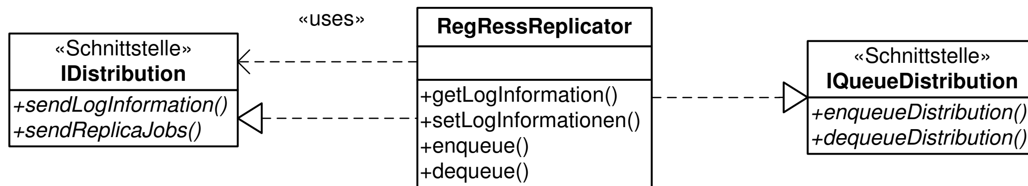

Abbildung 6.7.: Komponente RegRessReplicator

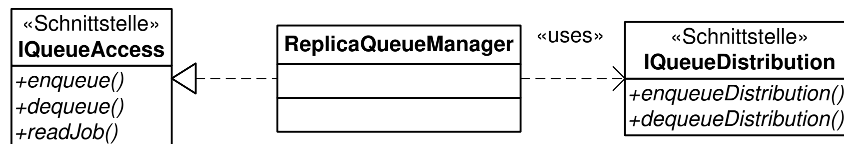

Abbildung 6.8.: Komponente ReplicaQueueManager

Methode `setLogInformation()` des `RegRessReplicator` in das Artefakt `System.log` gespeichert. Im Fall der benötigten Methode `sendLogInformation()` werden die eigenen Kennzahlen einer Komponente versendet. Die Kennzahlen werden mit der Methode `getLogInformationen()` des `RegRessReplicator` zuvor vom Artefakt `System.log` gelesen.

Analog werden mit den Methoden `sendReplicaJobs()` die Replica Queues der verschiedenen Instanzen des KARMA abgeglichen. Als Parameter wird übergeben, ob ein Auftrag einer Replica Queue anzuhängen ist oder ob ein Auftrag aus der Replica Queue zu entfernen ist. Damit der `RegRessReplicator` über Änderungen der Replica Queue informiert wird, bietet er die Schnittstelle `IQueueDistribution` an, die die Methoden `enqueueDistribution()` und `dequeueDistribution()` beinhaltet. Mittels dieser Methoden teilt ein `ReplicaQueueManager` Änderungen an der Replica Queue mit, die er kontrolliert. Wenn über die angebotene Schnittstelle `sendReplicaJobs()` Änderungen anderer Replica Queues empfangen werden, dann kann der `RegRessReplicator` diese Änderungen mittels der eigenen Methoden `enqueue()` und `dequeue()` in der lokalen Replica Queue vornehmen, d.h. im Artefakt `Replica.jobs`.

Es ist zu beachten, dass die Metadaten, insbesondere die verschiedenen Replica Queues, konsistent zueinander sind (siehe Abschnitt 4.2), damit keine Schreib-/Schreibkonflikte$_R$ auftreten können. Daher sind diese Daten mit einer synchronen Replikationsstrategie zu replizieren. In dem hier spezifizierten Fall wird die ROWA-Replikationsstrategie verwendet (siehe Abschnitt 3.3.1), d.h. alle Metadaten repräsentieren stets den gleichen Wert. Falls die Replica Queue verteilt ist, dann muss somit auch hier eine geeignete Transaktionskontrolle gewährleistet werden (siehe `TxController`).

Komponente ReplicaQueueManager

Der `ReplicaQueueManager` in Abbildung 6.8 steuert die Änderungen in der Replica Queue. Er verwaltet die Aufträge der Replica Queue und gewährleistet eine transaktionale Verarbeitung, d.h. das Anhängen und das Entfernen von Aufträgen an bzw. aus der Replica Queue wird vom `ReplicaQueueManager` unter Transaktionskontrolle gewährleistet. Dabei wird davon ausgegangen, dass der Replica Queue ein entsprechender Ressourcenmanager vorgeschaltet ist, der ebenfalls das erweiterte XA-Protokoll unterstützt (siehe `TxController`).

Die Schnittstelle `IQueueAccess`, die vom `ReplicaQueueManager` angeboten wird, beinhaltet die Methoden `enqueue()` zum Anhängen eines Auftrags, `dequeue()` zum Entfernen eines Auftrags und `readJob()` zum Lesen eines Auftrags ohne die Replica Queue zu verändern. Die Methoden erlauben das indizierte Zugreifen auf Elemente der Replica Queue, um die Performance zu steigern (siehe Abschnitt 4.2). Ein Auftrag beinhaltet folgende Informationen: Das logische

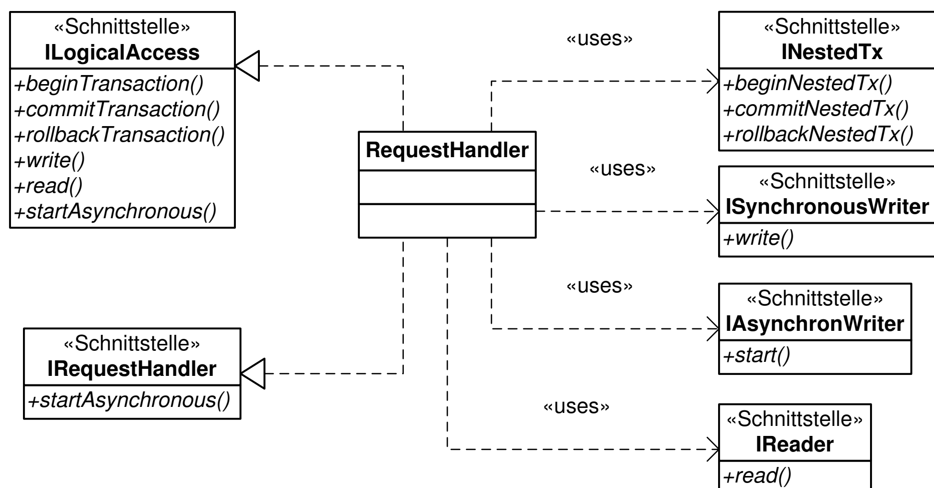

Abbildung 6.9.: Komponente RequestHandler

Objekt O^o, die initiierende Komponente K_i, die Zielkomponente K_z und die zu schreibenden Werte des Objekts.

Wenn der KARMA und damit die Replica Queue verteilt ist, dann müssen die Operationen auf der Replica Queue nicht nur lokal, sondern auch an allen Replica Queues durchgeführt werden. Deswegen benötigt der `ReplicaQueueManager` die Schnittstelle `IQueueDistribution`, die vom `RegRessReplicator` angeboten wird. Über die Methoden `enqueueDistribution()` und `dequeueDistribution()` wird das Anhängen und Entfernen von Aufträgen an bzw. aus der Replica Queue verteilt durchgeführt.

Komponente RequestHandler

Der `RequestHandler` in Abbildung 6.9 stellt gemäß dem Entwurfsmuster *„Fassade"* [GHJ04] eine einheitliche Schnittstelle für Clients zum KARMA dar, d.h. alle Anweisungen einer Transaktion eines Clients werden über den `RequestHandler` abgewickelt. Ein Ausnahme bildet das lokale Lesen von Replikaten, das die Replikationsstrategie RegRess erlaubt. Für die Zugriffe bietet der `RequestHandler` einem Client die Schnittstelle `ILogicalAccess` an, mit der die Transaktionsgrenzen und die Operationen mitgeteilt werden können. Dabei wird davon ausgegangen, dass Schreib- und Lesezugriffe in einer geschachtelten Transaktion eingebettet sind (siehe `TxController`), d.h. ein Client führt innerhalb seiner Transaktion mehrere Zugriffe durch, die jeweils eigenständige Teiltransaktionen bilden.

Mit der Methode `beginTransaction()` der Schnittstelle `ILogicalAccess` teilt der Client dem KARMA den Beginn seiner Transaktion mit, wobei die Transaktionsgrenzen über den `TxController` an einen Transaktionsmanager weitergereicht werden. Der Aufruf der Methode `beginTransaction()` kann auch implizit beim ersten Zugriff eines Clients vorgeschaltet werden. Das erfolgreiche Ende der Transaktion wird mit der Methode `commitTransaction()` und das Zurückrollen der Transaktion wird mit der Methode `rollback()` übermittelt. Mit den Methoden `write()` und `read()` der Schnittstelle `ILogicalAccess` können Schreib- bzw. Lesezugriffe auf logische Objekte durchgeführt werden. Der Methode `write()` werden die initiierende Komponente K_k, das logische Objekt O^o sowie die zu schreibenden Werte als Parameter übergeben. Der Rückgabewert ist der Status der Operation. Der Methode `read()` werden auch die initiierende Komponente K_k und das logische Objekt O^o übergeben, wobei hier neben dem Status der Operation natürlich das gelesene Replikat zurückgeliefert wird.

Die Methode `startAsynchronous()` der Schnittstelle `ILogicalAccess` spielt eine Sonderrolle: Hiermit wird die asynchrone Aktualisierung vom Client gestartet, um einen möglichst konsistenten Datenbestand zu erlangen. Diese Methode läuft nicht in der Client-Transaktion

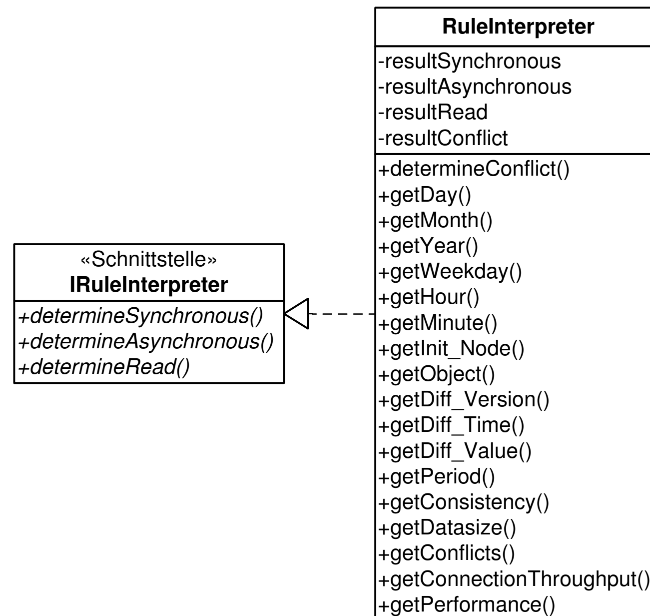

Abbildung 6.10.: Komponente RuleInterpreter

ab, weil für die asynchrone Aktualisierung eine eigene Transaktionskontrolle definiert ist (siehe `TxControlller`). Die Methode ist parameterlos und liefert einen Status. Sie ist im Wesentlichen identisch zur Methode `startAsynchronous()` der Schnittstelle `IRequestHandler`, die der `RequestHandler` dem `Timer` anbietet. Im Unterschied wird hier kein Rückgabewert geliefert.

Um die Transaktionsgrenzen, die vom Client an den `RequestHandler` überliefert werden, an einen Transaktionsmanager weiterzureichen, benötigt der `RequestHandler` die Schnittstelle `INestedTx`, die vom `TxController` angeboten wird. Mit der Methode `beginNestedTx()` wird der Beginn der geschachtelten, verteilten Transaktion eines Clients bekannt gegeben und mit der Methode `commitNestedTx()` das erfolgreiche Ende bzw. mit der Methode `rollbackNestedTx()` das Zurückrollen der geschachtelten, verteilten Transaktion.

Die Methoden des `RequestHandler`, die die Zugriffe auf die Datenobjekte anbieten, werden an die entsprechenden Komponenten weitergereicht, wobei Übergabeparameter und Rückgabewerte identisch sind. Die angebotene Methode `write()` wird an die Methode `write()` der Schnittstelle `ISynchronousWriter` weitergeleitet, die Methode `read()` an die Methode `read()` der Schnittstelle `IReader` und die Methode `startAsynchronous()` an die Methode `start()` der Schnittstelle `IAsynchronousWriter`.

Komponente RuleInterpreter

Der `RuleInterpreter` in Abbildung 6.10 führt die Inferenz auf Basis der Replikationsregeln durch, die in der RRML formuliert sind (siehe Kapitel 5). Dabei ist vom `RuleInterpreter` die Version der RRML zu beachten. Es wird unterschieden, ob die Inferenz bei der synchronen Aktualisierung, bei der asynchronen Aktualisierung bzw. bei Lesezugriffen durchgeführt wird, wofür die Begriffe InferenzSynchron, InferenzAsynchron bzw. InferenzLesen definiert wurden (siehe Abschnitt 4.2.4). Innerhalb dieser drei Inferenzarten kann die InferenzWiderspruch (siehe Abschnitt 5.3.5) aufgerufen werden, um widersprüchliche Regeln zu behandeln.

Die Methode `determineSynchronous()` der Schnittstelle `IRuleInterpreter`, die vom `RuleInterpreter` angeboten wird, führt die InferenzSynchron durch und liefert die Partitionierung in synchron und asynchron zu aktualisierende Replikate. Sie wird vom `SynchronousWriter` benötigt. Die angebotene Methode `determineAsynchronous()` ermittelt durch die InferenzAsynchron, ob derzeit ein Replikat asynchron aktualisiert werden darf. Diese Methode wird

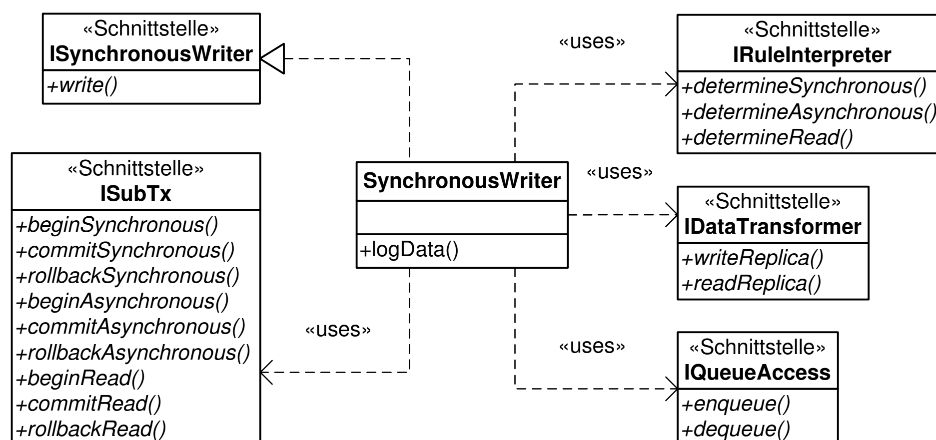

Abbildung 6.11.: Komponente SynchronousWriter

vom `AsynchronousWriter` benötigt. Die angebotene Methode `determineRead()` wickelt die InferenzLesen ab und wird vom `Reader` benötigt. Hier wird eine Liste von Replikaten geliefert, die speziellen Eigenschaften genügen. Die Methode `determineConflict()`, die in der Klasse `RuleInterpreter` definiert ist, führt die InferenzWiderspruch durch, die bei der Behandlung von widersprüchlichen Regeln von den oben genannten Inferenzarten aufgerufen wird. Der Ablauf der verschiedenen Inferenzarten sowie Parameter und Rückgabewerte sind im Abschnitt 5.3 spezifiziert. Die bei der Inferenz genutzten Regeln werden vom Artefakt `Rules.xml` gelesen.

Die Klasse `RuleInterpreter` in Abbildung 6.10 zeigt auch die Ergebnisvariablen, die bei der Inferenz manipuliert werden, um letztendlich den Rückgabewert zu bestimmen. Diese Ergebnisvariablen sind in den Abschnitten 5.3.2 bis 5.3.5 spezifiziert. Zusätzlich bietet die Klasse `RuleInterpreter` einige „Get"-Methoden, die die Funktionen einer Bedingung darstellen. Diese Funktionen sind im Abschnitt 5.3.6 spezifiziert. Sie greifen entweder auf das Artefakt `Replica.Jobs` zu, z.B. um mittels der Methode `getDiff_Version()` den Versionsabstand über die Replica Queue zu berechnen, oder auf das Artefakt `System.log`, um Systemkennzahlen wie die Performance einer Komponente mit Replikat mittels der Methode `getPerformance()` zu holen.

Komponente SynchronousWriter

Der `SynchronousWriter` in Abschnitt 6.11 deckt einen Schreibzugriff auf ein logisches Objekt ab, was die synchrone und asynchrone Aktualisierung der Replikate des logischen Objekts impliziert. Für einen derartigen Schreibzugriff bietet der `SynchronousWriter` die Schnittstelle `ISynchronousWriter` an, dessen Methode `write()` den Zugriff gemäß des Listings 4.1 auf Seite 71 ausführt. Im Wesentlichen erfolgt eine Partitionierung der Replikate, wobei die synchron zu aktualisierenden Replikate in dieser Methode selbst geschrieben werden und für die asynchron zu aktualisierenden Replikate Aufträge in die Replica Queue gestellt werden. Der Aufruf dieser Methode erfolgt durch den `RequestHandler`, der den Schreibzugriff eines Clients somit weiterreicht.

Für die Partitionierung der betroffenen Replikate benötigt der `SynchronousWriter` die Methode `determineSynchronous()` der Schnittstelle `IRuleInterpreter`, um vom `RuleInterpreter` die InferenzSynchron durchführen zu lassen. Das Schreiben eines Replikats der synchron zu aktualisierenden Replikate erfolgt mittels der benötigten Methode `writeReplica()` der Schnittstelle `IDataTransformer`, die vom `DataTransformer` angeboten wird. Um für die asynchron zu aktualisierenden Replikate entsprechende Aufträge in die Replica Queue zu stellen, benutzt der `SynchronousWriter` die Methode `enqueue()`, die vom `ReplicaQueueManager` durch die von ihm angebotene Schnittstelle `IQueueAccess` realisiert wird.

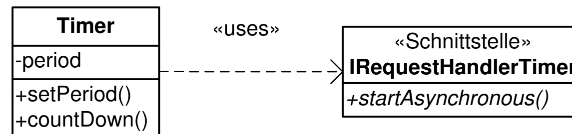

Abbildung 6.12.: Komponente Timer

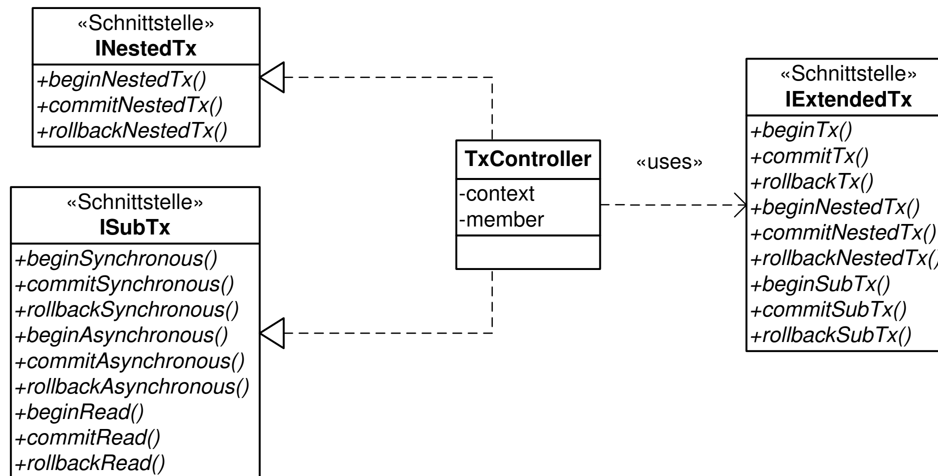

Abbildung 6.13.: Komponente TxController

Bei dem Schreibzugriff auf ein logisches Objekt handelt es sich um eine verteilte Transaktion, weil einige Replikate, die auf unterschiedlichen Komponenten (Rechnern) lokalisiert sind, synchron geschrieben werden, und für die anderen Replikate Aufträge in die transaktionale Replica Queue eingereiht werden. Diese verteilte Transaktion wird als Teiltransaktion der übergeordneten Client-Transaktion aufgefasst (siehe `TxController`). Daher teilt der `SynchronousWriter` dem `TxController` den Beginn und das Ende des Schreibzugriffs als Transaktionsklammer über die Methoden `beginSynchronous()` sowie `commitSynchronous()` für das erfolgreiche Beenden bzw. `rollbackSynchronous()` für das Zurückrollen beim Beenden mit. Diese benötigten Methoden werden von der Schnittstelle `ISubTx` bereitgestellt, die vom `TxController` angeboten wird.

Die Methode `logData()` des `SynchronousWriter`, die explizit in Abbildung 6.11 angegeben ist, dient zum Zugriff auf das Artefakt `System.log`. Nach dem Schreibzugriff auf das logische Objekt bzw. im Fehlerfall werden hier entsprechende Informationen abgelegt, z.B. die Performance einer Komponente, die ein synchron zu aktualisierendes Replikat speichert.

Komponente Timer

Der `Timer` in Abbildung 6.12 startet die asynchrone Aktualisierung entweder in periodischen Zeitabständen oder nach Abschluss eines Schreibzugriffs auf ein logisches Objekt. Es wird angenommen, dass die Periode in einer Konfigurationsdatei angegeben ist. Die Konfigurationsdatei ist nicht in der Softwarearchitektur in der Abbildung 6.3 dargestellt.

Die Methode `setPeriod()` der Klasse `Timer` liest die Periode aus der Konfigurationsdatei und initialisiert das Attribut `period`. Die Methode `countDown()` der Klasse `Timer` ruft in einer endlosen Iteration zunächst die Methode `setPeriod()` auf, um ggf. auch Änderungen der Konfigurationsdatei bei laufendem Betrieb zu berücksichtigen. Anschließend wird die Zeitspanne `period` gewartet, um dann die asynchrone Aktualisierung über die benötigte Schnittstelle `startAsynchronous()` der Schnittstelle `IRequestHandler`, die vom `RequestHandler` angeboten wird, zu aktivieren.

Komponente TxController

Der `TxController` in Abbildung 6.13 übermittelt die Transaktionsgrenzen an einen Transaktionsmanager, der die Synchronisation der beteiligten Ressourcenmanager übernimmt (siehe Abbildung 6.2 auf Seite 155). Der Transaktionsmanager muss sowohl einfache, verteilte Transaktionen als auch geschachtelte Transaktionen verarbeiten können. Auf die Transaktionsverarbeitung wird detailliert im Abschnitt 6.3 eingegangen. An dieser Stelle wird zunächst nur die von einem Transaktionsmanager anzubietende Schnittstelle `IExtendedTx` betrachtet, über die der `TxController` die Transaktionsgrenzen für einfache, verteilte Transaktionen, für geschachtelte, verteilte Transaktionen und für Teiltransaktionen, auch Subtransaktionen genannt, mitteilt.

Die Schnittstelle `IExtendedTx` bietet somit für die Synchronisation von einfachen, verteilten Transaktionen T die Methode `beginTx()`, um den Beginn von T zu kennzeichnen, die Methode `commitTx()`, um T erfolgreich zu beenden, d.h. die Änderungen gültig zu setzen, und die Methode `rollback()`, um T zurückzurollen, d.h. die Änderungen zurückzusetzen. Analog dienen die Methoden `beginNestedTx()`, `commitNestedTx()` und `rollbackNestedTx()` für die Synchronisation von geschachtelten, verteilten Transaktionen. Die Teiltransaktionen einer geschachtelten Transaktion werden mit den Methoden `beginSubTx()`, `commitSubTx()` und `rollbackSubTx()` synchronisiert.

Der `TxController` bietet zwei Schnittstellen an, die von internen Komponenten des KARMA genutzt werden. Die Schnittstelle `INestedTx` mit den Methoden `beginNestedTx()`, `commitNestedTx()` und `rollbackNestedTx()` benutzt der `RequestHandler`, um dem `TxController` den Beginn und das Ende der Client-Transaktion mitzuteilen. Diese angebotenen Methoden werden an die gleichnamigen Methoden der Schnittstelle `IExtendedTx` delegiert (siehe oben).

Schreibzugriffe eines Clients sind Teiltransaktionen der geschachtelten Transaktion (siehe Abschnitt 6.3). Daher benutzt der `SynchronousWriter` die vom `TxController` angebotenen Methoden `beginSynchronous()`, `commitSynchronous()` und `rollbackSynchronous()`, um den Beginn und das erfolgreiche Ende bzw. das Zurückrollen einer Teiltransaktion bekannt zu geben. Diese angebotenen Methoden werden an die entsprechenden Methoden `beginSubTx()`, `commitSubTx()` und `rollbackSubTx()` der Schnittstelle `IExtendedTx` delegiert. Gleiches gilt für Lesezugriffe, sodass die vom `Reader` benutzten Methoden `beginRead()`, `commitRead()` und `rollbackRead()` ebenfalls an die entsprechenden Methoden einer Teiltransaktion der Schnittstelle `IExtendedTx` delegiert werden.

Die asynchrone Aktualisierung, genauer gesagt, die Bearbeitung eines Auftrags bei der asynchronen Aktualisierung ist nicht Teiltransaktion einer übergeordneten, geschachtelten Transaktion. Zwar zerfällt auch diese Transaktion in zwei Teiltransaktionen, nämlich dem Schreibzugriff und dem Entfernen eines Auftrags aus der Replica Queue, dennoch braucht diese Transaktion nicht strukturiert zu werden. Daher kann hier die Synchronisation in einer einfachen, verteilten Transaktion erfolgen. Daher benutzt der `AsynchronousWriter` die vom `TxController` angebotenen Methoden `beginAsynchronous()`, `commitAsynchronous()` und `rollbackASynchronous()`, um den Beginn und das erfolgreiche Ende bzw. das Zurückrollen einer Transaktion mitzuteilen. Diese angebotenen Methoden werden an die entsprechenden Methoden `beginTx()`, `commitTx()` und `rollbackTx()` der Schnittstelle `IExtendedTx` delegiert.

Die Klasse `TxController` hat zwei Attribute: Das Attribut `context` speichert den Kontext einer Transaktion, z.B. die eindeutige Transaktionsnummer, die von einem Transaktionsmanager beim Beginn einer Transaktion geliefert wird. Das Attribut `member` speichert die Teilnehmer an einer Transaktion. Die Teilnehmer sind die Ressourcenmanager von Komponenten mit Replikat und/oder der Replica Queues. Entsprechende Informationen müssen den vom `TxController` angebotenen Methoden vom Aufrufer über die Parameter geliefert werden.

6.3. Transaktionsverarbeitung des KARMA

Neben der Entwicklung der Replikationsstrategie RegRess und der Definition der Regelsprache RRML, mit der die Replikationsregeln für RegRess formuliert werden können, ist die Konzeption eines Transaktionsmodells, mit dem sich die in RegRess spezifizierten Protokolle zur Koordination verwirklichen lassen, der dritte Schwerpunkt dieser Dissertation. So wurden schon in frühen Arbeiten, die im Kontext dieser Dissertation erstellt wurden, die transaktionale Anbindung von heterogenen, autonomen Informationssystemen betrachtet [NHW+02, NH02, NHHT03].

Die grobe Software-Architektur in Abbildung 6.2 auf Seite 155 zeigt, dass ein Transaktionsmanager, auch Transaktionskoordinator genannt, der ein geeignetes Transaktionsmodell realisiert, nicht Teil des Replikationsmanagers KARMA ist, sondern dass der KARMA den Transaktionsmanager über eine Schnittstelle `IExtendedTx` anspricht. Diese Schnittstelle ist in Abbildung 6.13 auf Seite 165 spezifiziert. Der Vorteil der Entkopplung des Replikationsmanagers und des Transaktionsmanagers liegt darin, dass so die einzelnen Komponenten austauschbar sind bzw. dass bei der Realisierung des KARMA ein geeigneter Transaktionsmanager wiederverwendet werden kann.

Ein Transaktionsmanager gewährleistet eine auf einem Transaktionsmodell bzw. auf einem Transaktionskonzept basierende Transaktionsverarbeitung von Transaktionen, wobei spezielle Eigenschaften, im Allgemeinen die so genannten ACID-Eigenschaften, erfüllt werden (siehe Abschnitt 2.2). Wesentlich ist, dass die Atomarität von Transaktionen durch geeignete Commit-Protokolle (siehe Abschnitt 2.2.2) realisiert wird. Für verteilte Transaktionen, um die es sich bei replizierten Datenbanken im Allgemeinen handelt, wurde das X/Open DTP Modell [Ope03] spezifiziert, das eine so genannte TX-Schnittstelle, über die eine Anwendung mit dem Transaktionsmanager kommuniziert, und eine so genannte XA-Schnittstelle, über die der Transaktionsmanager mit den Ressourcenmanagern kommuniziert, beinhaltet. Weil bei RegRess erweiterte Transaktionskonzepte benötigt werden, insbesondere geschachtelte Transaktionen, sind Erweiterungen dieser Schnittstellen erforderlich, die nachfolgend beschrieben werden. Daher werden in der Abbildung 6.2 auf Seite 155 die entsprechenden Schnittstellen als `IExtendedTx` und `IExtendedXa` bezeichnet.

Im Folgenden wird zunächst in Abschnitt 6.3.1 auf das Transaktionskonzept eingegangen, das vom KARMA bzw. einem Transaktionsmanager realisiert werden muss, um eine von der Replikationsstrategie RegRess geforderte Transaktionsverarbeitung zu unterstützen. Anschließend wird in Abschnitt 6.3.2 kurz der regelbasierte, dynamische Transaktionsmanager vorgestellt, der in der von mir betreuten Diplomarbeit von Andreas Austing [Aus06] entwickelt wurde.

6.3.1. Transaktionskonzepte

Der Replikationsmanager KARMA realisiert die Replikationsstrategie RegRess, sodass neben der Koordination der Zugriffe auf die Replikate auch eine geeignete Synchronisation der Transaktionen erfolgen muss. Die Komponente `TxController` des KARMA (siehe Abschnitt 6.2) ist für die Synchronisation von Transaktionen, die vom KARMA ausgeführt werden, verantwortlich. Dabei realisiert der KARMA nicht selbst die Synchronisation, sondern bedient sich eines geeigneten Transaktionsmanagers. Der `TxController` und der Transaktionsmanager müssen das in Abbildung 6.13 auf Seite 165 spezifizierte Verhalten realisieren, d.h. es muss ein geeignetes Transaktionskonzept implementiert sein.

Bei der Transaktionsverarbeitung im KARMA treten zwei Arten von Transaktionen auf: Entweder ist eine Transaktion von einem Client initiiert oder sie wird intern bei der asynchronen Aktualisierung ausgeführt. Wenn die Client-Transaktion Schreibzugriffe beinhaltet, dann wird in beiden Fällen auf die Replica Queue zugegriffen. Daher werden neben den elementaren Operationen, die in Abschnitt 2.2.1 definiert sind, zwei weitere Operation für das Anhängen und Entfernen an bzw. aus der Replica Queue benötigt. Somit werden folgende Operationen im Weiteren bei der Beschreibung einer Transaktion verwendet:

- BOT: Beginn einer Transaktion oder einer Teiltransaktion.
- EOT: Ende einer Transaktion. Beim Beenden können die Änderungen gültig gesetzt werden (Commit) oder zurückgerollt werden (Rollback).
- SP: Setzen eines Sicherungspunktes. Eine Transaktion bzw. Teiltransaktion kann auf einen Sicherungspunkt zurückgerollt werden, d.h. die letzten Änderungen ab dem Sicherungspunkt können zurückgesetzt werden.
- r(O^o): Lesen des o-ten Objekts.
- r(R_k^o): Lesen des Replikats, das zum o-ten Objekt gehört und auf der k-ten Komponente gespeichert ist.
- w(O^o): Schreiben des o-ten Objekts.
- w(R_k^o): Schreiben des Replikats, das zum o-ten Objekt gehört und auf der k-ten Komponente gespeichert ist.
- e(J_k^o): Anhängen eines Auftrags (Jobs) an die Replica Queue. Der Auftrag betrifft das Replikat R_k^o.
- d(J_k^o): Entfernen eines Auftrags (Jobs) aus der Replica Queue. Der Auftrag betrifft das Replikat R_k^o.

Wenn die Indize im Folgenden nicht angegeben werden, ist ein beliebiges logisches Objekt oder Replikat gemeint. Welche Werte geschrieben oder gelesen werden, ist in diesem Abschnitt nicht von Belang (vergleiche Abschnitt 4.1.2). Zunächst wird eine intern initiierte Transaktion betrachtet. Im Listing 4.2 auf Seite 74 ist ein Schreibzugriff bei der asynchronen Aktualisierung spezifiziert, wobei die Zeilen 18 bis 24 die Operationen der Transaktion zeigen. Vereinfacht mit den oben genannten Operationen handelt es sich um folgende Transaktion:

(6.1) BOT, w(R), d(J), EOT

Die Transaktion 6.1 beinhaltet also einen Schreibzugriff auf ein Replikat und ein Entfernen eines Auftrags aus der Replica Queue. In Abschnitt 2.2.5 wurde das Transaktionskonzept Queued Transactions [BHM90, BN97] vorgestellt, wobei dort eine Client-Server-Kommunikation gezeigt ist. Wesentlich ist, dass auch Zugriffe auf eine Warteschlange transaktional verarbeitet werden, d.h. dass die ACID-Eigenschaften erfüllt werden. Wenn Zugriffe auf eine Warteschlange wie in der Transaktion 6.1 innerhalb einer verteilten Transaktion erfolgen, d.h. neben dem Zugriff auf die Warteschlange wird auf eine andere Ressource zugegriffen, dann muss die Warteschlange ein entsprechendes Commit-Protokoll, im Allgemeinen das 2-Phasen-Commit-Protokoll, unterstützen (siehe Abschnitt 2.2.2). Nach dem oben genannten X/Open DTP Modell benötigt eine Warteschlange somit einen Ressourcenmanager, der das XA-Protokoll unterstützt.

Die Replica Queue des KARMA muss sich also aus Sicht des Transaktionsmanagers wie eine der anderen Ressourcen, d.h. wie die Komponenten mit Replikat, verhalten und eine Schnittstelle `IExtendedXa` anbieten, über die der Transaktionsmanager die Transaktion synchronisiert. Im Fall der Transaktion 6.1 ist keine Erweiterung des ursprünglichen TX- bzw. XA-Protokolls des X/Open DTP Modells nötig, weil es sich um eine einfache, verteilte Transaktion handelt. Deswegen beinhaltet die Schnittstelle `IExtendedTx` in Abbildung 6.13 auf Seite 165 auch Methoden, mit denen der `TxController` des KARMA dem Transaktionsmanager den Beginn bzw. das Ende einer verteilten Transaktion im ursprünglichen Sinne mitteilt. Entsprechendes gilt daher auch für die Schnittstelle `IExtendedXa` eines jeden Ressourcenmanagers.

Wenn ein Client eine Transaktion initiiert, dann wird eine Erweiterung des X/Open DTP Modells erforderlich, weil es sich dann um eine geschachtelte, verteilte Transaktion handelt. Zur Erläuterung wird zunächst eine Transaktion eines Clients betrachtet, die ausschließlich Schreibzugriffe enthält. Schreibzugriffe erfolgen auf logische Objekte, sodass die Transaktion bei m Schreibzugriffen folgendes Aussehen hat (ohne Beschränkung der Allgemeinheit wird hier angenommen, dass auf die ersten m logischen Objekte zugegriffen wird):

(6.2) BOT, w(O^1), w(O^2),..., w(O^m), EOT

Ein Schreibzugriff auf ein logisches Objekt O^o wird bei RegRess derart koordiniert, dass bei einem Replikationsgrad von n die n Replikate des logischen Objekts O^o in die Menge M_S^o der synchron zu aktualisierenden Replikate und die Menge M_A^o der asynchron zu aktualisierenden Replikate partitioniert werden. Die Mengen M_S^o und M_A^o werden für jedes logische Objekt neu bestimmt. Wenn die Mengen M_S^o und M_A^o nicht leer sind, dann setzt sich jeder Schreibzugriff der Transaktion 6.2 aus mehreren Operationen zusammen, nämlich aus Schreibzugriffen auf Replikate und aus Anhängen von Aufträgen an die Replica Queue:

(6.3) $\quad \mathrm{w}(O^i) = \mathrm{w}(R^i_{S_1}), \mathrm{w}(R^i_{S_2}), ..., \mathrm{e}(J^i_{A_1}), \mathrm{e}(J^i_{A_2}), ...$

Mit $R^i_{S_1}$ ist das erste Replikat des logischen Objekts O^i aus der Menge M^i_S gemeint, mit $R^i_{S_2}$ das zweite Replikat des logischen Objekts O^i aus der Menge M^i_S usw. Gleichermaßen bedeuten die Indize $J^i_{A_1}$ das erste Replikat des logischen Objekts O^i aus der Menge M^i_A usw. Ein Schreibzugriff auf ein logisches Objekt kann somit als Teiltransaktion aufgefasst werden, die selbst eine verteilte Transaktion ist. Dabei sind Zugriffe auf die Replica Queue involviert, falls M^i_A nicht leer ist.

Im Listing 4.1 auf Seite 71 ist ein derartiger Schreibzugriff bei RegRess erläutert, wobei die Transaktionsverarbeitung in den Zeilen 14 bis 30 spezifiziert ist. Wenn beim Zugriff auf die Replikate ein Fehler auftritt, wird bei so genannten wechselfähigen Replikaten die Teiltransaktion zurückgerollt, das fehlerverursachende Replikat den asynchron zu aktualisierenden Replikaten zugeordnet und die Teiltransaktion neu aufgesetzt. Daher ist es erforderlich, dass auf den Zustand zu Beginn einer Teiltransaktion zurückgesetzt werden kann, wofür Sicherungspunkte in die Transaktion eingeführt werden:

(6.4) $\quad \mathrm{BOT}, \mathrm{w}(O^1), \mathrm{SP}, \mathrm{w}(O^2), \mathrm{SP}, ..., \mathrm{w}(O^m), \mathrm{EOT}$

Damit handelt es sich bei der Transaktion 6.4 um eine geschachtelte Transaktion (siehe Abschnitt 2.2.5), deren Teiltransaktionen selbst verteilte Transaktionen sind. Die komplette Client-Transaktion stellt sich also wie folgt dar:

(6.5)
$$
\begin{aligned}
&\mathrm{BOT}, \mathrm{w}(R^1_{S_1}), \mathrm{w}(R^1_{S_2}), ..., \mathrm{e}(J^1_{A_1}), \mathrm{e}(J^1_{A_2}), ...,\\
&\quad \mathrm{SP}, \mathrm{w}(R^2_{S_1}), \mathrm{w}(R^2_{S_2}), ..., \mathrm{e}(J^2_{A_1}), \mathrm{e}(J^2_{A_2}), ...,\\
&\quad \mathrm{SP}, ...,\\
&\quad \mathrm{SP}, \mathrm{w}(R^m_{S_1}), \mathrm{w}(R^m_{S_2}), ..., \mathrm{e}(J^m_{A_1}), \mathrm{e}(J^m_{A_2}), ..., \mathrm{EOT}
\end{aligned}
$$

Zu Beginn jeder Teiltransaktion mit Ausnahme der ersten Teiltransaktion wird ein Sicherungspunkt gesetzt. Wenn die Client-Transaktion nur aus einem Schreibzugriff besteht, dann handelt es sich um eine einfache, verteilte Transaktion. Wenn zusätzlich Lesezugriffe auf logische Objekte vom Client durchgeführt werden, dann kann eine Teiltransaktion einen Lesezugriff auf genau einem Replikat enthalten, weil in der Version 2.0 der Replikationsstrategie RegRess kein gleichzeitiges Lesen mehrerer Replikate wie bei den Votierungsverfahren (siehe Abschnitt 3.3.1) erfolgt. Ein Beispiel einer Transaktion mit zwei Schreibzugriffen und zwei Lesezugriffen eines Clients könnte folgende Gestalt haben:

(6.6)
$$
\begin{aligned}
&\mathrm{BOT}, \mathrm{w}(R^1_{S_1}), \mathrm{w}(R^1_{S_2}), ..., \mathrm{e}(J^1_{A_1}), \mathrm{e}(J^1_{A_2}), ...,\\
&\quad \mathrm{SP}, \mathrm{r}(R^2_4),\\
&\quad \mathrm{SP}, \mathrm{w}(R^2_{S_1}), \mathrm{w}(R^2_{S_2}), ..., \mathrm{e}(J^2_{A_1}), \mathrm{e}(J^2_{A_2}), ...,\\
&\quad \mathrm{SP}, \mathrm{r}(R^6_7), \mathrm{EOT}
\end{aligned}
$$

In der Transaktion 6.6 wird beispielhaft auf die Replikate R_4^2 und R_7^6 lesend zugegriffen. Im Gegensatz zu einer Teiltransaktion bei einem Schreibzugriff, bei der es sich um eine verteilte Transaktion handelt, ist die Teiltransaktion eines Lesezugriffs eine einfache, lokale Transaktion. Im Listing 4.3 auf Seite 77 ist der Lesezugriff bei RegRess spezifiziert, wobei die Zeilen 12 bis 19 die Transaktionsverarbeitung festlegen. Weil bei einem Lesezugriff auf ein logisches Objekt mehrere passende Replikate ermittelt werden können, kann auf ein anderes Replikat ausgewichen werden, sofern ein Transaktionsfehler bei einem Lesezugriff auftritt. In diesem Fall wird, wie bei den Teiltransaktionen eines Schreibzugriffs, auch die Teiltransaktion des Lesezugriffs zurückgerollt. Zwar müssen hier keine Änderungen zurückgenommen werden, aber mögliche Sperren können wieder frei gegeben werden.

An dieser Stelle sei angemerkt, dass häufig auf Sperren bei Lesezugriffen verzichtet wird. In diesem Fall kann die Nebenläufigkeitsanomalie *„Non-Repeatable Read"* auftreten (siehe Abschnitt 2.2.3), d.h. beim wiederholten Lesen eines Replikats kann sich der Wert geändert haben, weil eine andere Transaktion das Replikat zwischenzeitlich geschrieben hat. Wie in einer Datenbank verfahren wird, kann im Allgemeinen über so genannte Isolationslevel [DD97] gesteuert werden. In der hier betrachteten Systemlandschaft bedeutet das, dass die Ressourcenmanager exklusive Sperren bei Lesezugriffen setzen können müssen (siehe Schnittstelle `IPhysicalAccess` der Komponente `Datatransformer`).

Der `TxController` des KARMA (siehe Abbildung 6.13 auf Seite 165) bietet über die Schnittstelle `INestedTx` einem Client Methoden an, mit denen der Client den Beginn und das Ende (Commit oder Rollback) einer geschachtelten Transaktion mitteilt. Des Weiteren bietet der `TxController` den eigenen Komponenten `SynchronousWriter` und `Reader` die Schnittstelle `ISubTx` an, über deren entsprechenden Methoden der Beginn und das Ende einer Teiltransaktion (Subtransaktion) mitgeteilt wird. Welche Wirkung mit diesen Methoden bei dem Transaktionsmanager ausgelöst werden, z.B. das Setzen von Sicherungspunkten, hängt von dem verwendeten Transaktionsmanager, genauer gesagt, von dessen Transaktionskonzept, ab. Das wiederum hängt davon ab, was die beteiligten Ressourcen an Transaktionsunterstützung bieten, wobei in einer heterogenen Systemlandschaft unterschiedliche Transaktionskonzepte der jeweiligen lokalen Ressourcen auftreten können. Deswegen werden nachfolgend Transaktionskonzepte aus der Literatur betrachtet, die eingesetzt werden könnten. Dabei sind ggf. Einschränkungen bei der Replikationsstrategie RegRess hinzunehmen, z.B. Einschränkungen der Konsistenz oder des Protokolls beim Schreibzugriff (siehe unten). Die folgenden Transaktionskonzepte werden betrachtet:

- Geschlossen-geschachtelte Transaktionen [Mos85]
- Offen-geschachtelte Transaktionen bzw. Sagas [GMS87]
- Verteilte Transaktionen nach dem X/Open DTP Modell [GR93, Ope03]
- Keine Unterstützung des XA-Protokolls bzw. Sagas_R [NHHT03]
- Keine Transaktionsunterstützung von Operationsfolgen

Geschlossen-geschachtelte Transaktionen

Die Transaktionen 6.5 und 6.6 zeigen Beispiele für den Idealfall des benötigten Transaktionskonzepts, nämlich der so genannten geschlossen-geschachtelten Transaktionen (siehe Abschnitt 2.2.5). Die Teiltransaktionen können isoliert zurückgerollt werden, d.h. es erfolgt ein Zurücksetzten bis zum Start der Teiltransaktion oder, hier, bis zum Sicherungspunkt. Sperren (siehe Abschnitt 2.2.3), die eine Teiltransaktion hält, werden bei einem Commit der Teiltransaktion an die übergeordnete Transaktion, also der geschlossen-geschachtelten Transaktion, übergeben, sodass eine Synchronisation der Nebenläufigkeit gewährleistet ist. Die geschlossen-geschachtelte Transaktion ist damit in der Lage, auch ein komplettes Zurückrollen aller Teiltransaktionen durchzuführen. Die Verwendung von geschlossen-geschachtelten Transaktionen gewährleisten die ACID-Eigenschaften [GR93] und sind somit ideal für den KARMA, weil keinerlei Einschränkungen der Replikationsstrategie RegRess nötig sind.

Offen-geschachtelte Transaktionen bzw. Sagas

Wenn die beteiligten Komponenten mit Replikat bzw. die Replica Queue keine geschlossen-geschachtelten Transaktionen unterstützen, was in heterogenen, autonomen Systemen durchaus häufig der Fall sein kann, dann können unter Umständen offen-geschachtelte Transaktionen, auch Sagas genannt, eingesetzt werden (siehe Abschnitt 2.2.5). In dem hier benötigten Transaktionskonzept heißt das, dass verteilte Transaktionen aneinander gehängt werden. Die Transaktion 6.6 sieht in diesem Fall wie folgt aus, wobei ein tiefer gestellter Index an der Operation BOT bzw. EOT die Nummer der Teiltransaktion zeigt:

(6.7) BOT_1, w($R^1_{S_1}$), w($R^1_{S_2}$),..., e($J^1_{A_1}$), e($J^1_{A_2}$), ..., EOT_1

BOT_2, r(R^2_4), EOT_2

BOT_3, w($R^2_{S_1}$), w($R^2_{S_2}$),..., e($J^2_{A_1}$), e($J^2_{A_2}$), ..., EOT_3

BOT_4, r(R^6_7), EOT_4

Jede Teiltransaktion der Sagas ist eine verteilte Transaktion gemäß dem X/Open DTP Modell (siehe Abschnitt 2.2.3), wobei hier auch Zugriffe auf eine Replica Queue erfolgen. Bei einem entsprechenden Ressourcenmanager der Replica Queue gibt es, wie bereits erwähnt, keine Unterschiede, d.h. die beteiligten Ressourcenmanager müssen das XA-Protokoll unterstützen. Somit kann eine Teiltransaktion auf den Zustand, der beim Beginn der Teiltransaktion vorlag, zurückgerollt werden, sodass das im Listing 4.1 auf Seite 71 spezifizierte Protokoll für einen Schreibzugriff bzw. das im Listing 4.3 auf Seite 77 spezifizierte Protokoll für einen Lesezugriff aus Sicht der Transaktionsverarbeitung gewährleistet ist.

Ein Nachteil der Verwendung von Sagas besteht darin, dass mittels Commit beendete Teiltransaktionen ihre Änderungen für andere Transaktionen sichtbar machen und die Sperren aufheben. Somit kann es zur Nebenläufigkeitsanomalie der so genannten *„Lost Updates“* kommen (siehe Abschnitt 2.2.3). Auch das Zurückrollen der kompletten Client-Transaktion, also der Sagas, ist schwierig, weil bereits abgeschlossene Teiltransaktionen nur mittels Kompensationstransaktionen zurückgerollt werden können. Das Problem dabei ist es, dass entsprechende Kompensationstransaktionen vorliegen müssen. Falls während der zurückzusetzenden Sagas andere Transaktionen schon auf die geänderten Werte zugegriffen haben, dann müssen entweder auch diese Transaktionen zurückgesetzt werden (*„kaskadierendes Zurückrollen“*) oder aber es werden *„Dirty Reads“* (siehe Abschnitt 2.2.3) in Kauf genommen.

Wenn die Kompensationstransaktionen im `TxController` definiert werden, bieten Sagas den Vorteil, dass keine erweiterten Schnittstellen zum Transaktionsmanager und zu den Ressourcenmanagern benötigt werden. Somit können die ursprünglich von der X/Open definierten Schnittstellen verwendet werden. Der Nachteil liegt darin, dass im Allgemeinen kein isoliertes Zurückrollen möglich ist und dass abgeschwächte Konsistenz wegen der Nebenläufigkeitsanomalien toleriert werden muss.

Verteilte Transaktionen nach dem X/Open DTP Modell

Wenn weder geschlossen- noch offen-geschachtelte Transaktionen verwendet werden können, sondern einfache, verteilte Transaktionen nach dem X/Open DTP Modell (siehe Abschnitt 2.2.2), dann werden ebenfalls keine erweiterten Schnittstellen des Transaktionsmanagers benötigt. Der Transaktionsmanager muss in diesem Fall das TX-Protokoll und die Ressourcenmanager das XA-Protokoll unterstützen. Die Methoden der Schnittstellen vom `TxController` des KARMA, mit denen die Transaktionsgrenzen von Teiltransaktionen kommuniziert werden, bleiben ohne Wirkung. Da nun keine Teiltransaktion isoliert zurückgerollt werden kann, muss die komplette Client-Transaktion entweder nur aus einem Zugriff bestehen, was eher unüblich ist, oder die Protokolle für Schreib- und Lesezugriffe von RegRess müssen angepasst werden.

Das Zurückrollen bei einem Schreibzugriff wird dann erforderlich, wenn ein wechselfähiges Replikat beim synchronen Schreibzugriff einen Fehler verursacht und die (Teil-)Transaktion erneut aufgesetzt wird, wobei das fehlerverursachende Replikat nun asynchron aktualisiert wird (siehe Listing 4.1 auf Seite 71). Auf diese Funktionalität muss bei einem Transaktionskonzept mit einfachen, verteilten Transaktionen verzichtet werden, d.h. während der Client-Transaktion können keine Replikate in ihrer Aktualisierungsart wechseln. Wenn ein Fehler auftritt, dann muss die komplette Client-Transaktion abgebrochen werden.

Wie bereits oben erwähnt, wird eine Teiltransaktion beim Lesen deswegen zurückgerollt, um mögliche Sperren freizugeben. Ein Fehler bei einem Lesezugriff muss jedoch nicht zwingend die komplette Transaktion ungültig setzen, sondern nur die einzelne Operation wird annulliert. Selbst wenn Sperren zunächst gesetzt bleiben, ist es effizienter, nicht die komplette Client-Transaktion abzubrechen, sondern einfach das nächste passende Replikat zu lesen (siehe Listing 4.3 auf Seite 77), d.h. bei einem Lesezugriff ist nur dann ein Abbruch erforderlich, wenn keines der passenden Replikate gelesen werden kann. Als Beispiel wird die Transaktion 6.7 erneut auf dieses Transaktionskonzept angepasst:

(6.8) $\text{BOT}, \text{w}(R^1_{S_1}), \text{w}(R^1_{S_2}),..., \text{e}(J^1_{A_1}), \text{e}(J^1_{A_2}), ...,$

$\text{r}(R^2_4), \text{r}(R^2_5),$

$\text{w}(R^2_{S_1}), \text{w}(R^2_{S_2}),..., \text{e}(J^2_{A_1}), \text{e}(J^2_{A_2}), ...,$

$\text{r}(R^6_7), \text{EOT}$

In der Transaktion 6.8 sind keine Teiltransaktionen oder Sicherungspunkte enthalten. Gegenüber der Transaktion 6.7 wird hier angenommen, dass das Lesen des Replikats R^2_4 wegen einer Sperre einer anderen Transaktion annulliert wird. Unter der Voraussetzung, dass weitere passende Replikate durch die Inferenz ermittelt wurden (siehe Abschnitt 4.2.3), wird nun das Replikat R^2_5 erfolgreich gelesen und die ursprüngliche Transaktion wird fortgesetzt. Wenn allerdings ein Schreibzugriff w(...) oder ein Anhängen an die Replica Queue e(...) fehlschlägt, dann muss die Client-Transaktion komplett zurückgerollt werden.

Keine Unterstützung des XA-Protokolls bzw. Sagas$_R$

Angemerkt sei, dass beim Replikationsmanager ARM (siehe Abbildung 6.1 auf Seite 154) ein anderes Verständnis einer Teiltransaktion vorliegt. Dort ist eine Teiltransaktion nicht eine verteilte Transaktion z.B. auf Grund eines Schreibzugriffs (siehe Operationsfolge 6.3), sondern eine Teiltransaktion ist eine lokale, flache Transaktion. Diese Teiltransaktionen werden dann benötigt, wenn die beteiligten Ressourcenmanager das XA-Protokoll nicht unterstützen, sondern nur eine lokale Transaktionsverarbeitung, d.h. die eigene lokale Transaktion erfolgreich beenden (Commit) oder zurückrollen (Rollback) können. Das folgende Beispiel illustriert diese Teiltransaktionen, wobei angenommen wird, dass zwei Replikate auf drei Komponenten lokalisiert sind und keine Aufträge in die Replica Queue geschrieben werden:

(6.9) $\text{BOT}_1, \text{w}(R^1_1), \text{w}(R^2_1), \text{EOT}_1$

$\text{BOT}_2, \text{w}(R^1_2), \text{w}(R^2_2), \text{EOT}_2$

$\text{BOT}_3, \text{w}(R^1_3), \text{w}(R^2_3), \text{EOT}_3$

In der Transaktion 6.9 werden je Teiltransaktion nur Replikate einer Komponente geschrieben. Bei dieser Anordnung der Operationen, bei denen die Teiltransaktion komplett nacheinander ausgeführt werden, ist die Gefahr größer, dass Inkonsistenzen auftreten. Daher wurde eine Überlappung der Teiltransaktionen, insbesondere der *„Commit-Zeitpunkte“* vorgeschlagen, was als

Sagas_R [NHHT03] bezeichnet wird, also eine Sagas mit Index „R“ gekennzeichnet. Die Sagas_R zur Transaktion 6.9 hat somit folgende Operationsfolge:

$$\begin{aligned}
(6.10)\qquad & \text{BOT}_1,\ \text{w}(R_1^1),\ \text{w}(R_1^2), \\
& \text{BOT}_2,\ \text{w}(R_2^1),\ \text{w}(R_2^2), \\
& \text{BOT}_3,\ \text{w}(R_3^1),\ \text{w}(R_3^2), \\
& \text{EOT}_1,\ \text{EOT}_2,\ \text{EOT}_3
\end{aligned}$$

Wenn in der Transaktion 6.10 jede Teiltransaktion erfolgreich beendet werden kann, dann wird eine *„quasi-synchrone“* Transaktionsverarbeitung erreicht. Dies gilt entsprechend für die Transaktion 6.7 bzw. 6.8, wo jede Teiltransaktion bzw. Transaktion eine verteilte Transaktion ist. Sind hier Ressourcenmanager beteiligt, die das XA-Protokoll nicht unterstützen, sollten die Commit-Zeitpunkte nahe beieinander liegen. Wenn allerdings ein Fehler auftritt und eine Teiltransaktion nicht erfolgreich beendet werden kann und somit zurückgerollt wird, so treten die bekannten Inkonsistenzen auf (siehe Abschnitt 2.2.2). Das Beheben der Inkonsistenzen kann in diesem Fall vom KARMA in der Form realisiert werden, dass die zurückgesetzten Änderungen als Aktualisierungsaufträge in die Replica Queue eingetragen werden oder die gültig gesetzten Änderungen mittels Kompensationstransaktionen zurückgenommen werden (siehe oben).

Keine Transaktionsunterstützung von Operationsfolgen

Abschließend soll die wohl schwächste Transaktionsverarbeitung einer Ressource betrachtet werden, die unter Umständen bei der Replikation von Daten in heterogenen, autonomen Informationssystemen auftreten kann. Eine Ressource unterstützt weder geschachtelte Transaktionen, noch verteilte Transaktionen inklusive des XA-Protokolls, noch lokale Transaktionen, d.h. es existiert kein Mechanismus, um eine Operationsfolge gültig zu setzen oder zurückzurollen. Die Ressource bietet nur das Schreiben oder Lesen eines Replikats an. Ein derartiges Verhalten kann als Transaktion aufgefasst werden, die nur eine Schreiboperation oder einen Lesezugriff pro Transaktion erlaubt. Hierfür müssen dann schon die ACID-Eigenschaften gewährleistet werden, um überhaupt Aussagen zum Zustand der Ressource zu treffen.

Bei nur einer Zugriffsoperation je Transaktion muss entweder die Client-Anwendung auch entsprechend angepasst werden, d.h. es ist nur ein Zugriff auf ein logisches Objekt je Client-Transaktion erlaubt (siehe oben), oder die Client-Transaktion wird derart in Teiltransaktionen zerlegt, dass jeder Zugriff auf ein Replikat eine eigene Teiltransaktion bildet. Entweder werden nun abgebrochene Teiltransaktionen über die Replica Queue so lange wiederholt, bis sie erfolgreich beendet werden, oder die übrigen Teiltransaktionen werden mittels Kompensationstransaktionen zurückgesetzt. So oder so wird ohne entsprechende Unterstützung durch einen Ressourcenmanager die Transaktionsverwaltung in replizierten Datenbanken aufwändig und im Allgemeinen ist Isoliertheit und Konsistenz nicht zu gewährleisten (siehe oben).

6.3.2. Regelbasierter, dynamischer Transaktionsmanager

Im Rahmen dieser Dissertation wurde die Diplomarbeit von Andreas Austing mit dem Titel „Erweiterte Transaktionskonzepte in der Java Enterprise Edition“ erstellt [Aus06]. Das Hauptziel der Arbeit war die Entwicklung eines regelbasierten, dynamischen Transaktionsmanagers (RD-TM), womit die Entwicklung eines transaktionalen Gesamtkonzeptes und die Spezifikation eines Entwicklungsprozesses zur Umsetzung transaktionaler Anwendungen verknüpft war. Der RD-TM wurde bei der RedDot Solutions, ein Anbieter von Enterprise Content Management Systemen, evaluiert, sodass die Anwendbarkeit in einem realen System belegt wurde. Damit kann der RD-TM auch bei der transaktionalen Anbindung von heterogenen, autonomen Infor-

mationssystemen, in denen Daten gemäß der Replikationsstrategie RegRess repliziert werden, eingesetzt werden.

In diesem Abschnitt werden die wichtigsten Konzepte des RD-TM vorgestellt, damit gezeigt wird, wie die in Abschnitt 6.3.1 diskutierten Transaktionskonzepte realisiert und kombiniert werden können. Detailliert sind die Modelle, Spezifikationen und Implementierungen in der Diplomarbeit erläutert [Aus06]. Insbesondere die Konzepte sind auch auf andere Technologien als die Java Enterprise Edition [JBC+06] übertragbar. Die wichtigsten Anforderungen an den RD-TM sind:

1. Unterstützung von Anwendungsvorgängen: Der RD-TM unterstützt Anwendungsvorgänge in Form von so genannten Workflows bzw. des zugehörigen Transaktionsmodells *„Contracts“* [WR92].
2. Unterstützung des ACID-Transaktionskonzepts: Der RD-TM unterstützt das so genannte ACID-Transaktionskonzept bzw. das X/Open DTP Modell [Ope03].
3. Unterstützung von erweiterten Transaktionskonzepten: Der RD-TM unterstützt erweiterte Transaktionskonzepte wie z.B. Sagas [GMS87] oder Queued Transactions [BN97].
4. Deklarative Bestimmung des Transaktionskonzepts: Das Transaktionskonzept, das der RD-TM für eine Aktivität eines Anwendungsvorgangs verwendet, kann deklarativ mittels eines Deskriptors bestimmt werden.
5. Dynamische Adaption des Transaktionskonzepts: Zur Laufzeit kann das Transaktionskonzept ausgetauscht werden. Dies gilt für neu gestartete Transaktionen.
6. Regelbasierter Austausch des Transaktionskonzepts: Der Austausch des Transaktionskonzepts wird durch Transaktionsregeln gesteuert, wobei der Zustand des Systems, also über Sensoren ermittelte Kennzahlen, berücksichtigt werden.

Die erste Anforderung zeigt, dass der RD-TM komplexe Anwendungsvorgänge wie Workflows unterstützen soll. Somit kann bei Verwendung des RD-TM als Transaktionsmanager des KARMA einem Client eine umfangreiche Operationsfolge gestattet werden. Die zweite und dritte Anforderung beziehen einerseits einfache, verteilte und andererseits geschachtelte, verteilte Transaktionen mit in die Transaktionsverarbeitung ein, sodass die in Abschnitt 6.3.1 genannten Transaktionen realisiert werden können. Mit der vierten Anforderung ist gemeint, dass das Transaktionskonzept deklarativ bestimmt werden kann, wofür einheitliche Schnittstellen bezüglich der Definition der Transaktionsgrenzen für die verschiedenen Transaktionskonzepte geschaffen wurden.

Der RD-TM ist in der Lage, wie in der fünften Anforderung spezifiziert, zur Laufzeit dynamisch das Transaktionskonzept anzupassen, d.h. für eine Aktivität bzw. Operationsfolge eines Anwendungsvorgangs kann das Transaktionskonzept gewechselt werden. Dies gilt für neu gestartete Transaktionen. Hierfür wurden erweiterte Benutzerschnittstellen (vgl. TX-Schnittstelle des X/Open DTP Modells) spezifiziert. Die sechste Anforderung legt fest, dass der Austausch des Transaktionskonzepts regelbasiert erfolgt. Mittels Regeln und Sensoren, die Systemkennzahlen liefern, kann ein Wechsel von einem Transaktionskonzept zu einem anderen initiiert werden. Angemerkt sei, dass hierüber sogar einfache Replikationsregeln wie der Wechsel von synchroner zu asynchroner Aktualisierung realisierbar ist.

In Abbildung 6.14 ist die Architektur des RD-TM als Schichtenmodell dargestellt. Dabei handelt es sich um eine Verallgemeinerung und Erweiterung der Architektur des J2EE Activity Service [Rob03]. Neben der Architektur auf der linken Seite sind zwei mögliche Implementierungsvarianten auf der rechten Seite abgebildet: Die erste Implementierung ist auf CORBA [COR02] ausgerichtet, die zweite auf Grund der Webservice-Unterstützung [Ley03] auf serviceorientierte Architekturen. Nachfolgend werden die sechs Schichten erläutert:

1. **Transaction Context Transport Layer**: Auf der untersten Schicht der Architektur ist das Protokoll spezifiziert, mit dem der Transaktionskontext zwischen den beteiligten Komponenten ausgetauscht wird. In CORBA wird der Transaktionskontext mittels IIOP (In-

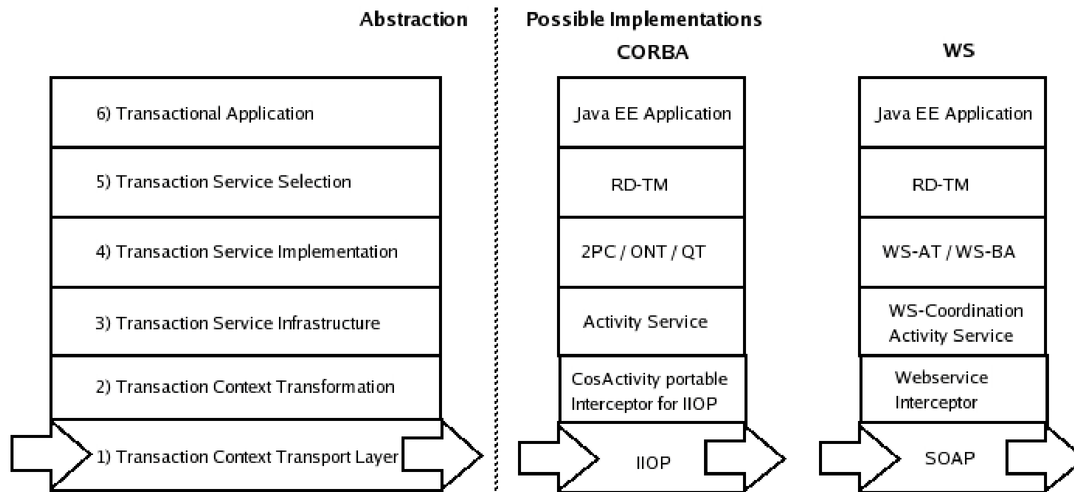

Abbildung 6.14.: Die Architektur des RD-TM [Aus06]

ternet Inter-ORB Protocol) kommuniziert, bei Webservices mittels SOAP (Simple Object Access Protocol).

2. **Transaction Context Transformation**: In dieser Schicht wird der Transaktionskontext, insbesondere die globale Transaktionsnummer, extrahiert und in eine Objektrepräsentation transformiert. Sowohl bei CORBA als auch bei Webservices werden dafür so genannte *„Interceptoren“* eingesetzt. Ein Interceptor realisiert allgemein einen Sicherheitsdienst. In diesem Fall wird der Transaktionskontext dem aktuellen Thread-Kontext hinzugefügt.
3. **Transaction Service Infrastructure**: Die Transaktionsinfrastruktur-Schicht stellt einen flexiblen Dienst dar, auf dessen Basis sich unterschiedliche Transaktionskonzepte realisieren lassen, d.h. ein Dienst, der die Konstruktion der unterschiedlichen Koordinationsprotokolle verschiedener Transaktionskonzepte unterstützt. Der Activity Service ist ein solcher Dienst, der sowohl für Java als auch für CORBA spezifiziert ist [LMP04]. Für Webservices existiert die WS-Coordination Spezifikation, die sich ebenfalls auf Basis des Activity Service realisieren lässt.
4. **Transaction Service Implementation**: In dieser Schicht werden die konkreten Transaktionskonzepte als Dienste, d.h. als Transaktionsmanager, implementiert. In CORBA können neben einfach, verteilten Transaktionen nach dem 2-Phasen-Commit-Protokoll (2PC) erweiterte Transaktionskonzepte wie die offen-geschachtelten Transaktionen (ONT) und Queued Transactions (QT) umgesetzt werden. ONT und QT wurden in der Diplomarbeit [Aus06] realisiert. Bei Webservices gibt es entsprechend die Protokolle WS-AtomicTransaction und WS-BusinessActivity.
5. **Transaction Service Selection**: Die benötigte Infrastruktur zur transparenten Selektion eines Transaktionsmanagers, der ein bestimmtes Transaktionskonzept realisiert, wird auf dieser Schicht bereitgestellt. Der in der Diplomarbeit [Aus06] entwickelte RD-TM ist eine mögliche Implementierung.
6. **Transaction Application**: Eine transaktionale Anwendung, z.B. der KARMA, greift auf den transaktionalen Selektionsdienst zu, z.B. auf den RD-TM.

In der Diplomarbeit [Aus06] wird neben der Architektur eine Spezifikation der Komponenten des RD-TM vorgenommen. Des Weiteren werden die zugehörigen Schnittstellen der Komponenten, z.B. auch zu einem Regelinterpreter und den Sensoren, definiert. Die Protokolle z.B. für den Ablauf einer Transaktion, zur Adaption unterschiedlicher Transaktionskonzepte und zur Ausführung von Anwendungsregeln werden spezifiziert. Dabei wurde auf hohe Abstraktion geachtet, um eine leichte Austauschbarkeit der verschiedenen Komponenten und Artefakte zu

ermöglichen. Wie bereits erwähnt, wurden die Transaktionskonzepte offen-geschachtelte Transaktionen (ONT) und Queued Transactions (QT) als so genannte *„High-Level-Services“* entworfen und implementiert, die auf den *„Low-Level-Service“* Activity Service aufsetzen.

Der RD-TM ist in der Diplomarbeit auf Basis des frei verfügbaren Java Applikationsservers JBoss [Com08] implementiert worden, wobei die Architektur der CORBA Implementierungsvariante entspricht (siehe Abbildung 6.14 auf Seite 175). Aus Sicht dieser Dissertation ist insbesondere die Realisierung der erweiterten Transaktionskonzepte wichtig. Interessant ist auch die Verwendung des Frameworks JBoss Rules [Rul08], womit der regelbasierte Austausch der Transaktionskonzepte realisiert wurde. JBoss Rules bietet einen Regelinterpreter, der Wenn-Dann-Regeln auswertet. Im Ausblick wird der Einsatz von JBoss Rules anstelle der RRML bzw. des eigen entwickelten Regelinterpreters diskutiert (siehe Kapitel 9).

7. Prototypische Implementierung und Evaluation

In diesem Kapitel wird die prototypische Implementierung der regelbasierten Replikationsstrategie RegRess beschrieben, mit der die erstellten Konzepte, Methoden und Protokolle hinsichtlich ihrer Realisierung überprüft wurden. Bezüglich der Replikationsstrategie RegRess bedeutet das im Wesentlichen, dass die Protokolle zur Koordination der Schreib- und Lesezugriffe implementiert werden müssen (siehe Kapitel 4), wobei ein Regelinterpreter die Replikationsregeln auswerten muss. Die Replikationsregeln, die in der Replication Rule Markup Language (RRML, siehe Kapitel 5) formuliert sind, bestimmen die Art der Koordination. Somit ist ein konfigurierbarer, adaptiver Replikationsmanager (KARMA) zu implementieren, der RegRess realisiert. Die Architektur und Komponenten des KARMA wurden in Kapitel 6 spezifiziert.

Die Prototypen, die im Zusammenhang mit dieser Dissertation entstanden sind, wurden in der Programmiersprache Java entwickelt. Als Plattform für einen KARMA dient daher die Java Platform, Enterprise Edition (Java EE, Version 5, [JBC+06]), genauer gesagt, ein Applikationsserver, der der Java EE Spezifikation entspricht. Ein Java EE Applikationsserver stellt technische Funktionalität für die in Java entwickelten Anwendungen zur Verfügung, z.B. Transaktionsmanagement, Persistenzdienste und Kommunikationsdienste.

Die Vorgängerversion der Java EE nannte sich Java 2 Platform Enterprise Edition (J2EE, Version 1.4), die die Plattform für einige der erstellten Prototypen war. Deswegen kann in dieser Dissertation sowohl von Java EE als auch von J2EE gesprochen werden, was also letztlich nur die Versionsnummer der hier verwendeten technologischen Plattform darstellt. Falls ein Prototyp nicht die vollständige Funktionalität eines Applikationsservers benötigte, wurde auch die so genannte Java Platform Standard Edition (Java SE, vormals J2SE) als Programmierumgebung genutzt.

Da leider kein reales, verteiltes System zur Verfügung stand, in der replizierte Daten verwaltet werden, konnte der KARMA nicht in einer produktiven Systemlandschaft getestet werden. Deswegen wurde auch kein einzelner Prototyp entwickelt, der die komplette Funktionalität des KARMA beinhaltet, sondern es wurden Prototypen entwickelt, die spezielle Teile der Ergebnisziele dieser Dissertation abdeckten. Nach [Flo84] handelt es sich somit um experimentelle Prototypen. Der Vorteil dieser Aufsplittung war es auch, dass diese experimentellen Prototypen im Rahmen von studentischen Arbeiten entwickelt und implementiert werden konnten. Eine Ausnahme bildet die Implementierung der eigentlichen Replikationsstrategie RegRess (siehe Abschnitt 7.3.2), die von mir selbst vorgenommen wurde.

Der Einsatz des KARMA in einem produktiven System hätte den Vorteil gehabt, RegRess unter echten Bedingungen testen und ggf. vergleichen zu können. Allerdings beinhaltet ein reales System den Nachteil, dass im Allgemeinen keine gleichen Vorbedingungen hergestellt werden können, d.h. die Reproduzierbarkeit von Experimenten ist nicht gewährleistet. Weiterhin ist es häufig schwierig, dass reale System zu modifizieren, um spezielle Eigenschaften zu testen. Deswegen bietet sich die Simulation an, bei der diese Nachteile nicht bestehen [LK91, BCN99]. Ein weiterer Vorteil der Simulation ist es, dass vorab Aussagen zur Konsistenz, z.B. in Form des Konsistenzgrades (siehe Abschnitt 4.3.3), möglich sind, d.h. der Einfluss von Änderungen z.B. der Replikationsregeln kann gemessen werden. Der Nachteil der Simulation besteht darin, dass die Aussagekraft der Experimente von der Gültigkeit des Simulationsmodells gegenüber der Realität abhängt.

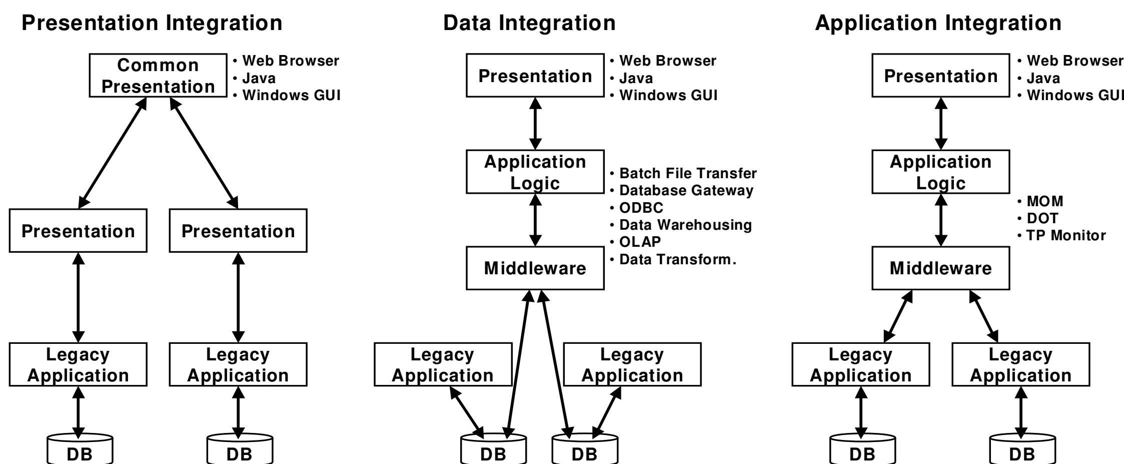

Abbildung 7.1.: Integrationsmodelle [RMB00]

Der Aufbau dieses Kapitels gliedert sich wie folgt: In Abschnitt 7.1 wird die Integration heterogener Informationssysteme mittels Java EE beschrieben, wobei auf die für den KARMA relevanten Aspekte fokussiert wird. Die im Kontext der Dissertation implementierten Prototypen, die im Rahmen studentischer Arbeiten entstanden sind, werden in Abschnitt 7.2 vorgestellt. Anschließend wird die Simulation von Replikationsstrategien thematisiert, d.h. der Abschnitt 7.3 stellt die Implementierung des KARMA in einem Simulationsframework vor. Abschließend erfolgt eine Evaluation der Simulationsexperimente in Abschnitt 7.3.4.

7.1. Integration heterogener Informationssysteme mittels der Java EE

Die Kopplung einzelner Informationssysteme zur gemeinsamen Nutzung von Ressourcen wird als *„Integration von Informationssystemen“* [Has00] oder als *„Enterprise Application Integration“* (EAI, [CHK05]) bezeichnet. Weil in dieser Dissertation die Replikation von Daten betrachtet wird, wird davon ausgegangen, dass die Integration die Daten als Ressource betrifft, d.h. die Daten der Informationssysteme müssen abgeglichen, also repliziert, werden. Laut Annahme (siehe Abschnitt 4.1.1) wird von einer voll-replizierten Datenbank ausgegangen, d.h. alle beteiligten Informationssysteme speichern jedes logische Objekt als Replikat. In dieser Dissertation wird anstelle der Informationssysteme, die Replikate speichern, die Bezeichnung Komponente mit Replikat gewählt (siehe Definition 34 auf Seite 61).

In Abbildung 7.1 sind Integrationsmodelle abgebildet, wobei nach [RMB00] zwischen Präsentations-, Datenbank- und Applikationsintegration unterschieden wird. Ein Replikationsmanager stellt eine Middleware [BN96] dar (siehe Abbildung 4.1 auf Seite 61), die letztlich auf die Replikate der Komponenten mit Replikat zugreift. Nach den Integrationsmodellen in Abbildung 7.1 kann der Zugriff entweder wie bei der Datenbankintegration direkt auf die Datenbank parallel zu Altanwendungen erfolgen oder wie bei der Applikationsintegration über die Altanwendung bzw. dessen Schnittstellen (API).

Unter Verwendung der Java EE Technologie kann bei direktem Datenbankzugriff die Java Database Connectivity (JDBC [SS00]) verwendet werden. Falls über Schnittstellen der Altanwendungen auf die Daten zugegriffen wird, kann dies z.B. per Java Remote Methode Invocation (RMI [Gro01]) erfolgen. Eine Standardisierung beim Zugriff über Schnittstellen liefert die Java Connector Architecture (JCA [SSN02]). In diesem Fall muss für eine Altanwendung ein Ressourcenadapter zur Verfügung stehen, über den die Zugriffe durchgeführt werden.

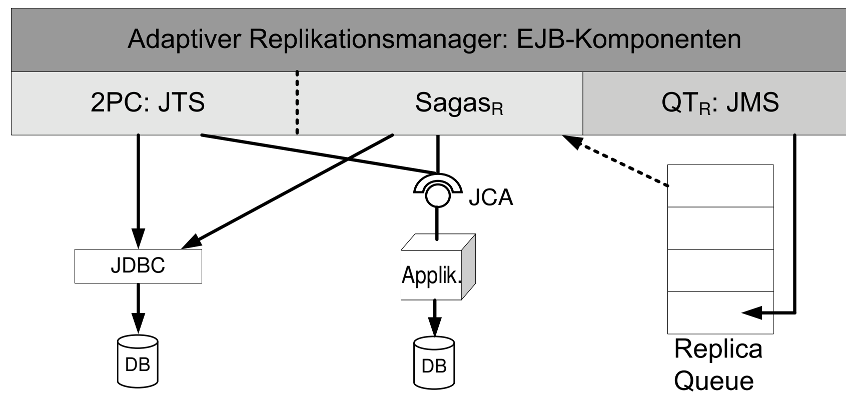

Abbildung 7.2.: Realisierung des ARM mittels der Java EE Technologie [NHHT03]

Insbesondere zu Beginn dieses Dissertationsvorhabens wurde die transaktionale Anbindung der Komponenten mit Replikat betrachtet. Hierbei wurde die zuvor genannte Unterscheidung, d.h. ein Zugriff erfolgt direkt auf die Datenbank oder auf Schnittstellen der Altanwendung bzw. der lokalen Anwendung, berücksichtigt (siehe Abbildung 6.1 auf Seite 154). Die Realisierung des ARM (**A**daptiver **R**eplikations**m**anager), wie gesagt, eine frühe Version des KARMA, mittels der Java EE Technologie ist in Abbildung 7.2 veranschaulicht, wobei die oben genannten Schnittstellen JDBC und JCA als Schnittstellen für den Datenbankzugriff bzw. den Zugriff über eine API dienen. Bei der Grob-Architektur des KARMA (siehe Abbildung 6.2 auf Seite 155) wurde hingegen derart abstrahiert, dass diese Schnittstellen zur Schnittstelle `IPhysicalAccess` zusammengefasst wurden, die von einem Ressourcenmanager, der der Komponente mit Replikat vorgeschaltet ist, angeboten werden muss.

Des Weiteren zeigt die Abbildung 7.2 die Transaktionsschicht. Die Umsetzung der Transaktionsverarbeitung mittels der Java EE Technologie erfolgt im synchronen Fall über den Java Transaction Service (JTS [LMP04]), wobei das 2PC in Form des TX- bzw. XA-Protokolls von den Java EE Servern unterstützt wird. Die beteiligten Komponenten mit Replikat bzw. die zugehörigen Ressourcenmanager müssen somit eine XA-Schnittstelle anbieten. Falls das Sagas$_R$-Konzept im ARM für die quasi-synchrone Replikation verwendet werden muss, war eine eigene Implementierung nötig.

Das QT$_R$-Konzept für den asynchronen Fall wird über den Nachrichtendienst Java Message Service (JMS) realisiert, wobei die Replica Queue als Topic (Publish/Subscribe-Verfahren) oder Queue (Point-To-Point-Verfahren) gemäß der JMS-Terminologie gestaltet wird [MHC01]. Angemerkt sei, dass in dieser Dissertation von einer Replica Queue ausgegangen wird, die einen indizierten Zugriff erlaubt, um die Performance zu steigern. Eine andere Möglichkeit der Performancesteigerung wäre z.B. durch den Einsatz mehrerer Replica Queues für die einzelnen Komponenten mit Replikat zu erreichen.

Im KARMA ist die Transaktionsverarbeitung durch die Komponente `TxController` modelliert, die die Schnittstelle `IExtendedTx` eines Transaktionsmanagers benötigt. Ein möglicher Transaktionsmanager in der Java EE Technologie ist der RD-TM, der in Abschnitt 6.3.2 vorgestellt wurde. Der RD-TM verwendet die oben genannten Java Dienste JTS und JMS sowie eine Implementierung der offen-geschachtelten Transaktionen (ONT) als so genannte High-Level-Services.

Abschließend kann gesagt werden, dass die Java EE Technologie ideale Voraussetzungen für die transaktionale Anbindung von Informationssystemen (Komponenten mit Replikat) bietet. Es existieren Schnittstellen sowohl für den Datenbankzugriff mit JDBC als auch für den Zugriff über Schnittstellen mit JCA. Mit JTS und JMS werden Transaktionsdienste für einfache, verteilte Transaktionen und transaktionale Warteschlangen angeboten. Der RD-TM erweitert diese

Dienste um offen-geschachtelte Transaktionen. Die Verwendung dieser Java EE Technologien wurde durch Prototypen implementiert und evaluiert (siehe Abschnitt 7.2).

7.2. Implementierte Prototypen durch studentische Arbeiten

In diesem Abschnitt werden die Protoytpen vorgestellt, die im Zusammenhang mit dieser Dissertation durch studentische Arbeiten erstellt wurden. Dabei wurde der Umfang der Arbeiten derart gewählt, dass sie innerhalb des vorgegebenen Zeitrahmens, im Allgemeinen sechs Monate, zu bewältigen war. Neben der Einarbeitung und Literaturrecherche beinhaltet eine studentische Arbeit vor allem den Entwurf mit prototypischer Implementierung und Evaluation eines Teils der Ergebnisziele dieser Dissertation. Damit dient einer der so entwickelten Prototypen der Klärung der technischen Machbarkeit spezieller Funktionalität des KARMA. Es handelt sich somit um experimentelle Prototypen [Flo84].

Wie bereits erwähnt, wurden die Prototypen mittels der Java EE Technologie bzw. dessen Vorgänger, der J2EE Technologie, realisiert (siehe oben). Die Dokumentationen und Implementierungen stehen in der Abteilung Software Engineering des Department Informatik, Fakultät II der Carl von Ossietzky Universität in Oldenburg zur Verfügung (siehe Quellenangaben). Bei der nachfolgenden Vorstellung der Prototypen wird folgender Aufbau gewählt:

- **Transaktionsverarbeitung**

  P1 MeReB (**Me**ssage and **Re**plication **B**enchmark) in der Diplomarbeit von Thomas Wolf, Juni 2002: *Benchmark für EJB-Transaction und Message-Services* [Wol02]

  P2 RD-TM (**R**egelbasierter, **d**ynamischer **T**ransaktions**m**anager) in der Diplomarbeit von Andreras Austing, Juli 2006: *Erweiterte Transaktionskonzepte in der Java Enterprise Edition* [Aus06]

- **Replikationsmanager und Regelsprache**

  P3 SAPAdapter in der Diplomarbeit von Ludger Bischofs, Juni 2002: *Anbindung von SAP R/3 über die J2EE Connector-Architektur* [Bis02]

  P4 ARepMan (**A**daptiver **Rep**likations**man**ager) in der Diplomarbeit von Michael Hülsmann, Oktober 2002: *Entwicklung eines adaptiven Replikationsmanagers auf Basis von EJB-JTS/-JMS* [Hül02]

  P5 RIM (**R**ule **I**nferencing **M**achine) in der Diplomarbeit von Jan Stefan Addicks, März 2005: *Entwicklung eines XML-basierten Regelsprachensystems für Replikationsstrategien* [Add05]

  P6 rulEdit (**rul**e **Edit**or) ebenfalls wie P5 in der Diplomarbeit von Jan Stefan Addicks, März 2005: *Entwicklung eines XML-basierten Regelsprachensystems für Replikationsstrategien* [Add05]

- **Simulation von Replikationsstrategien**

  P7 F4SR (**F**ramework **4** (for) **S**imulation of **R**eplication Strategies) in der Diplomarbeit von Markus Fromme, Mai 2006: *Framework zur Simulation und Evaluation von Replikationsstrategien* [Fro06]

  P8 Erweiterung des F4SR (siehe P7) im individuellen Projekt von Nicolai Kalisch, August 2007: *Simulation von Konfliktmanagementstrategien für die asynchrone Replikation* [Kal07]

Angemerkt sei, dass im Zusammenhang mit dieser Dissertation auch die Diplomarbeit von Michael Weers „Eine UML-basierte Abbildungssprache zwischen Datenschemata“ [Wee03] entstanden ist. Eine grafische Abbildungssprache, die in der Diplomarbeit entwickelt wurde, ermöglicht

die Modellierung bidirektionaler Schemaabbildungen. Hierdurch wird die Datentransformation (siehe Komponente `DatenTransformer` in Abschnitt 6.2) erleichtert. Da aber die Schematransformation (siehe z.B. [Con02]) in dieser Dissertation laut Annahme nicht betrachtet wird (siehe Abschnitt 4.1.1), wird auf diese studentische Arbeit nicht weiter eingegangen.

Transaktionsverarbeitung

Zum Thema Transaktionsverarbeitung wurden zwei Prototypen durch studentische Arbeiten erstellt. Der MeReB (siehe oben, P1) diente zur Überprüfung der Dienste Java Transaction Service (JTS [LMP04]) und Java Message Service (JMS [MHC01]). Als Datenbank diente der Microsoft SQL-Server 2000, der mittels der JDBC-Treiber von Microsoft angebunden wurde. Als J2EE Server wurden BEA Weblogic 6.1, JBoss 2.4.4 und der Sun J2EE Referenzserver verwendet. Als Anwendungsszenario wurde die Datenreplikation gewählt, wobei die Replikationsstrategien ROWA als synchrone Replikationsstrategie und Peer-To-Peer-Replikation als asynchrone Replikationsstrategie implementiert wurden (siehe Abschnitt 3.3). Für die Messungen wurde ein eigener Benchmark entwickelt, ebenfalls MeReB genannt. Der Prototyp MeReB zeigt die Verwendung der genannten Java Dienste für verschiedene J2EE Server und erlaubt die Bewertung an Hand geeigneter Messergebnisse des Benchmarks.

Der RD-TM (P2) ist ein Transaktionsmanager bzw. ein Transaktionsdienst, der die regelbasierte Auswahl geeigneter Transaktionskonzepte bzw. der zugehörigen Transaktionsmanager als High-Level-Services erlaubt. Eine ausführliche Beschreibung des RD-TM befindet sich in Abschnitt 6.3.2. Der RD-TM wurde auf dem Java EE Server JBoss entwickelt. Die Evaluation im Industrieunternehmen Red Dot Solutions zeigte seine Praxistauglichkeit und ist somit für den KARMA zur Abwicklung der benötigten transaktionalen Funktionalität geeignet.

Replikationsmanager und Regelsprache

Unter dem Punkt Replikationsmanager und Regelsprache sind die Prototypen zusammengefasst, die die Replikationsstrategie RegRess und deren Kontext, wie z.B. die Anbindung von Komponenten mit Replikat, betreffen. Dabei basiert die Entwicklung auf früheren Versionen oder einfacheren Protokollen als die in dieser Dissertation spezifizierten Protokolle. Nichtsdestotrotz zeigen die Prototypen die technische Machbarkeit der Konzepte, weil die ursprünglichen Konzepte sukzessive erweitert wurden.

Der SAPAdapter (P3) diente zur Klärung der Frage, wie eine Altanwendung (Legacy System) über die J2EE Connector-Architektur (JCA [SSN02]) angebunden werden kann. Als Altanwendung wurde das Enterprise Resource Planning System SAP R/3 gewählt, das weit verbreitet bei Unternehmensanwendungen ist. Der SAPAdapter ist eine exemplarische Entwicklung eines Ressourcenadapters nach der JCA-Technologie, wobei auch die transaktionale Anbindung berücksichtigt wird. Die Transaktionsverarbeitung hängt natürlich von den Schnittstellen ab, die die Altanwendung bereitstellt. Angemerkt sei, dass mittlerweile für die meisten Altanwendungen entsprechende Adapter nach der JCA-Technologie vom Hersteller der Altanwendung oder von Drittanbietern kommerziell angeboten werden.

Ein erster Prototyp eines Replikationsmanagers, der Regeln zur Koordination verwendet, ist der ARepMan (P4), der in Publikationen auch ARM (**A**daptiver **R**eplikations**m**anager) genannt wird (z.B. [NHHT03], siehe auch Abbildung 6.1 auf Seite 154 bzw. Abbildung 7.2 auf Seite 179). Der ARepMan wurde unter Verwendung des J2EE Applikationsservers Bea WebLogic Server Release 6.1 entwickelt, wobei der enthaltene JMS Server als Replica Queue genutzt wurde. Die Replikate wurden auf die relationalen Datenbanken Cloudscape von IBM, Oracle Server 8.1.7 und Microsofts SQL-Server 2000 gespeichert, die alle das XA-Protokoll unterstützen. Die Anbindung erfolgte über XA-fähige JDBC-Treiber. Bei der Implementierung wurden eher einfache Replikationsregeln herangezogen, die in einer Datenbank gespeichert wurden. Der ARepMan beinhaltet einen `SyncUpdater` für die synchrone Aktualisierung und einen `AsyncUpdater` für die asynchrone Aktualisierung sowie einen einfachen Regelinterpreter. Mit dem ARepMan wur-

de die transaktionale Anbindung über JDBC der Komponenten mit Replikat gezeigt sowie die Basisfunktionalität des KARMA, wobei auf komplexe Regeln verzichtet wurde.

In der Diplomarbeit von Jan Stefan Addicks [Add05] wurden zwei Prototypen implementiert und insbesondere die Regelsprache RRML (**R**eplication **R**ule **M**arkup **L**anguage) in der Version 1.0 entwickelt. Der RIM (P5) ist ein Regelinterpreter, der die Inferenz der Regeln durchführt, die in der RRML Version 1.0 formuliert sind. Der RIM stellt somit eine Komponente dar, die der KARMA als Komponente `RuleInterpreter` nutzen kann (siehe Abschnitt 6.2), wobei für die Version 2.0 der RRML die in Abschnitt 7.3.2 genannten Anpassungen erforderlich sind. In der Diplomarbeit diente der RIM vor allem zur Evaluierung der RRML, d.h. es wurde geprüft, ob sinnvolle Ergebnisse für eine Menge von Regeln durch die Inferenz ermittelt werden. Dabei bietet der RIM unter anderem die syntaktische Prüfung der XML-Datei sowie die Prüfung auf Nicht-Terminierung der Inferenz.

Der Prototyp rulEdit (P6), der ebenfalls in der Diplomarbeit von Jan Stefan Addicks [Add05] entwickelt wurde, stellt ein Hilfsmittel für die Formulierung und Verwaltung der Replikationsregeln in der RRML dar. Die Regeln, die im XML-Format gespeichert werden, können in der ON-IF-THEN-Darstellung (siehe Abschnitt 5.1) bearbeitet werden. Der Editor bietet die üblichen Funktionen eines Editors wie Einfügen, Löschen und Editieren von Regeln. Des Weiteren existieren Funktionen zur Verwaltung der XML-Dateien und der zugehörigen Mapping-Dateien. Außerdem bietet der rulEdit die Möglichkeit, Simulationsläufe auf den Replikationsregeln durchzuführen, d.h. unter Verwendung des Prototypen RIM (P5) wird eine Inferenz auf den Regeln durchgeführt.

Simulation von Replikationsstrategien

Um Replikationsstrategien untereinander vergleichen zu können oder die Auswirkungen von Konfigurationsänderungen einer Replikationsstrategie bemessen zu können, z.B. die Änderung von Regeln bei RegRess, bietet sich die Simulation an. Die Simulation hat gegenüber realen Systemen den Vorteil, dass gleiche Bedingungen für die einzelnen Experimente hergestellt werden können.

Der F4SR (P7) ist ein Simulator, mit dem Replikationsstrategien evaluiert werden können. Er bietet einen Plugin-Mechanismus, der das Einbinden von Replikationsstrategien erlaubt. Auf das Simulationsmodell und den Simulator wird detailliert in Abschnitt 7.3.1 eingegangen. In der Diplomarbeit von Markus Fromme [Fro06] wurde der F4SR evaluiert, indem die Replikationsstrategien ROWA, Majority Consensus und Peer-To-Peer (siehe Abschnitt 3.3) implementiert wurden.

Eine Erweiterung des F4SR (P8) wurde im individuellen Projekt von Nicolai Kalisch [Kal07] um die Replikationsstrategien CODA und ROAM (siehe Abschnitt 3.3.2) vorgenommen, die zu den komplexen asynchronen Replikationsstrategien zählen. Dafür war es nötig, den F4SR derart zu erweitern, dass Konflikte erkannt und automatisch behandelt werden können. Außerdem wurde das Analysewerkzeug (siehe Abschnitt 7.3.1) um Kennzahlen erweitert, die gerade bei der asynchronen Aktualisierung von Interesse sind, z.B. Anzahl Konflikte oder Anzahl gelöster Konflikte.

Bewertung der Prototypen

Abschließend kann gesagt werden, dass durch die Prototypen der studentischen Arbeiten die technische Funktionalität und Machbarkeit einer regelbasierten Replikationsstrategie geklärt werden konnte. Die transaktionale Anbindung von Komponenten mit Replikat mittels JDBC und JCA sowie den zugehörigen Transaktions- bzw. Nachrichtendiensten JTS und JMS wurden in den Prototypen MeReB (P1), SAPAdapter (P3) und ARepMan (P4) untersucht. Der RD-TM (P2) bietet die vom KARMA benötigten Transaktionskonzepte, insbesondere die offengeschachtelten Transaktionen.

Der ARepMan (P4) zeigt, dass eine regelbasierte Partitionierung der Replikate in synchron und asynchron zu aktualisierende Replikate durchführbar ist, wobei hier einfache Regeln mit ei-

nem einfachen Regelinterpreter verwendet wurden. Die erste Version der RRML mit zugehörigem Regelinterpreter RIM (P5) demonstriert, dass auch komplexe Regeln zu einer geeigneten Partitionierung führen. Um Experimente mit Replikationsstrategien durchführen zu können, bietet der F4SR (P7, P8) einen geeigneten Plugin-Mechanismus. Somit verbleibt, die Replikationsstrategie RegRess mit den in dieser Dissertation erweiterten Protokollen und Sprachelementen, d.h. Version 2.0 der RRML, im Simulator zu evaluieren (siehe Abschnitt 7.3).

7.3. Simulation der Replikationsstrategie RegRess

Die Implementierung der Replikationsstrategie RegRess zum Zweck der Simulation wird in diesem Abschnitt vorgestellt. Nach [Gra92] ist *„Simulation“* wie folgt definiert: *„Simulation ist die Durchführung von Experimenten mit Modellen dynamischer Systeme“*, wobei mit Experiment ausdrücklich keine analytischen Verfahren gemeint sind. Die Simulation sollte natürlich eine Auswertung und Analyse der Daten beinhalten, die durch die Durchführung der Experimente gewonnen werden [BCN99]. Eine Simulation kann auf verschiedene Arten durchgeführt werden (siehe z.B. [LK91]). In dieser Dissertation wird die *„diskrete, ereignisgesteuerte Simulation“* verwendet, d.h. Zustandsänderungen des Modells erfolgen durch interne oder externe Ereignisse, wobei die Zeitabstände zwischen den Ereignissen unterschiedlich sind.

Das Modell, genauer das *„Simulationsmodell“*, ist eine Abstraktion der realen Welt bzw. eines realen Systems. Im vorliegenden Fall ist das Simulationsmodell eine Abstraktion einer replizierten Datenbank, d.h. von Komponenten mit oder ohne Replikaten und einem Replikationsmanager. Ein Simulationsmodell ist ein formales Modell, sodass es von einem Computer „verstanden“ wird. Ein *„Simulator“* ist ein Programm, dass ein Simulationsmodell abarbeitet. Bei der diskreten, ereignisgesteuerten Simulation besteht ein Simulator aus einer Ereignisliste, Ereignisroutinen, einem Zustandsraum und einer Uhr. Ein Ereignis, genauer ein *„Simulationsereignis“*, bewirkt eine Zustandsänderung, z.B. ein Schreibzugriff auf Replikate, wobei im Simulator wie folgt verfahren wird:

Die Ereignisliste speichert alle Simulationsereignisse in einer festen Reihenfolge. Der Simulator arbeitet die Ereignisliste ab, wobei die Uhr um die Simulationszeit weitergeschaltet wird, das Simulationsereignis entfernt wird und die Ereignisroutinen an Hand des Simulationsereignisses den Zustandsraum verändern und/oder neue Simulationsereignisse auslösen, die an die Ereignisliste angefügt werden. Diese Abarbeitung wiederholt sich, bis die Ereignisliste leer ist oder eine vorher definierte Abbruchbedingung, z.B. eine Endzeit, erfüllt ist.

Der Prototyp F4SR [Fro06], der in der Diplomarbeit von Markus Fromme entwickelt wurde (siehe auch Abschnitt 7.2), ist ein Simulator, der eine diskrete, ereignisgesteuerte Simulation durchführt, um Replikationsstrategien zu untersuchen. Hierauf wird in Abschnitt 7.3.1 eingegangen. Die Implementierung des Replikationsmanagers KARMA, der die Replikationsstrategie RegRess umsetzt, wird in Abschnitt 7.3.2 vorgestellt. Darauf folgt in Abschnitt 7.3.3 die Erläuterung des KARMA-Plugins für den F4SR. In Abschnitt 7.3.4 werden die Simulationsergebnisse abschließend evaluiert.

7.3.1. Framework für die Simulation von Replikationsstrategien

Der F4SR (**F**ramework **4** (for) **S**imulation of **R**eplication Strategies) ist ein Simulator, der mittels diskreter, ereignisgesteuerter Simulation die Evaluation von Replikationsstrategien erlaubt. Er wurde in der von mir betreuten Diplomarbeit von Markus Fromme [Fro06] entwickelt und wird in diesem Abschnitt kurz vorgestellt. Die Bezeichnung „Framework“ beruht darauf, dass Anwendungsentwickler den F4SR erweitern oder anpassen können, z.B. indem neue Replikationsstrategien hinzugefügt werden, wie im individuellen Projekt von Nicolai Kalisch (siehe Abschnitt 7.2). Neben den Anwendungsentwicklern nutzen Analytiker den F4SR als Simulator, um implementierte Replikationsstrategien zu simulieren und zu analysieren.

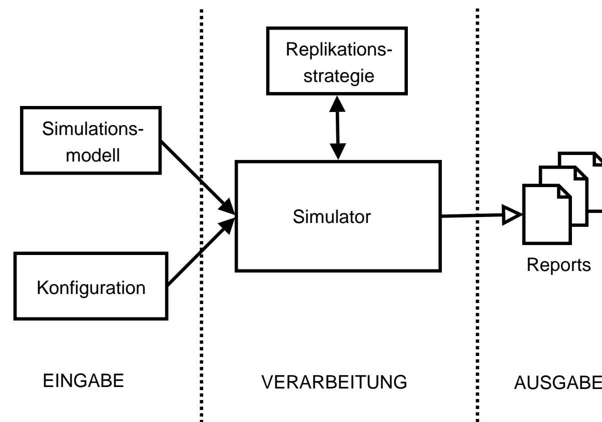

Abbildung 7.3.: Ein-/Ausgabemodell des F4SR [Fro06]

Das Simulationsmodell des F4SR bildet eine replizierte Datenbank ab. Es werden mehrere Komponenten mit Replikat (siehe Definition 34 auf Seite 61) modelliert. Angemerkt sei, dass der F4SR auch die partielle Replikation erlaubt, d.h. eine Komponente mit Replikat muss nicht für jedes logische Objekt ein Replikat speichern, wo hingegen hier laut Annahme von voll-replizierten Datenbanken ausgegangen wird (siehe Abschnitt 4.1.1). Das Simulationsmodell unterstützt Schreib- und Lesezugriffe auf die Replikate und somit auf die Komponenten mit Replikat. Die Komponenten mit Replikat können sich in der Verarbeitungsgeschwindigkeit z.B. eines Schreibzugriffs unterscheiden. Im Simulationsmodell lassen sich auch Lastspitzen abbilden, die für Verzögerungen bei der Ausführung von Zugriffen verantwortlich sind. Des Weiteren können Komponenten mit Replikat ausfallen, d.h. eine Komponente mit Replikat ist entweder temporär oder dauerhaft während der Simulation nicht verfügbar. Dabei wird nicht zwischen Ausfall der Komponente und Netzwerkproblemen unterschieden.

In Abbildung 7.3 ist ein Ein-/Ausgabemodell des F4SR nach [Fro06] dargestellt. Der F4SR benötigt zur Durchführung eines Simulationslaufs ein wie oben beschriebenes Simulationsmodell. Das Simulationsmodell wird vom Analytiker mittels Parametereingabe generiert. Als weitere Eingabe ist eine Konfiguration erforderlich, die alle Einstellungen enthält, die für einen Simulationslauf benötigt werden. Hiermit sind Einstellungen z.B. über die maximale Simulationszeit und für das Verhältnis von Schreib- zu Lesezugriffen gemeint. In der Konfiguration wird auch das Zeitverhalten aller Simulationsereignisse festgelegt.

Während der Simulation wird eine implementierte Replikationsstrategie verwendet, die vor Beginn der Simulation vom Analytiker ausgewählt wurde, wobei je nach Replikationsstrategie spezielle Einstellungen vom Analytiker konfiguriert werden. Beim Ablauf der Simulation werden vom F4SR alle Aktionen und deren Auswirkungen protokolliert, sodass am Ende der Simulation entsprechende Reports ausgegeben werden können. Hierfür wurden entsprechende Metriken definiert: Antwortzeit eines Datenzugriffs, Anzahl Schreib- bzw. Lesezugriffe, Anzahl veralteter Lesezugriffe, Anzahl erfolgreicher Schreib- bzw. Lesezugriffe, Anzahl abgebrochener Schreib- bzw. Lesezugriffe und Alter der Replikate.

Mögliche Simulationsereignisse, die während eines Simulationslaufs eintreten können, sind in der Abbildung 7.4 zu sehen. Die Simulationsereignisse betreffen entweder „Lastspitzen“, „Knotenausfall“ (hier: Ausfall einer Komponente) oder „Datenzugriff“. Das Simulationsereignis Lastspitzen simuliert eine hohe Auslastung bei einem Knoten (Komponente mit Replikat), d.h. die Antwortzeiten bei einem Zugriff verzögern sich gegenüber der normalen Antwortzeit der Komponente mit Replikat. Angemerkt sei, dass die Komponenten mit Replikat je nach Konfiguration über unterschiedliche normale Antwortzeiten verfügen. Ob bei Eintreten dieses Simulationsereignisses eine Überlastung einer Komponente mit Replikat gesetzt wird, hängt zusätzlich von einer Zufallszahl ab.

Beim Ausfall einer Komponente mit Replikat kann unterschieden werden, ob die Komponente mit Replikat dauerhaft während der Simulation ausfällt (Totalausfall) oder temporär nicht erreichbar ist. Eine temporär ausgefallene Komponente mit Replikat kann nach Eintreten des entsprechenden Ereignisses wieder verfügbar sein. Auch bei Eintreten dieser Simulationsereignisse wird zusätzlich eine Zufallszahl geprüft, bevor eine Komponente mit Replikat entsprechend gekennzeichnet wird. Wenn eine Komponente mit Replikat nicht verfügbar ist, ob dauerhaft oder temporär, dann können keine Schreib- oder Lesezugriffe auf die Replikate dieser Komponente durchgeführt werden.

Das Simulationsereignis Datenzugriff kann einen Schreib- oder Lesezugriff betreffen. Für diese Simulationsereignisse wird einerseits das Schreib-/Leseverhältnis benötigt, andererseits die durchschnittliche Zugriffszeit eines solchen Datenzugriffs. Weiterhin wird zwischen globalen und lokalen Schreib- bzw. Lesezugriffen unterschieden. Lokale Zugriffe erfolgen auf die initiierende Komponente mit Replikat. Globale Zugriffe erfolgen auf andere Komponenten mit Replikat, je nach Replikationsstrategie im Allgemeinen auf eine Menge von Replikaten. Bei der Ausführung eines Zugriffs wird geprüft, ob die Komponente mit Replikat verfügbar ist und ob keine Sperre im Zuge einer Transaktionsverarbeitung eines anderen Zugriffs vorliegt.

Die Abbildung 7.5 zeigt die grobe Architektur des F4SR, der in drei Komponenten aufgeteilt ist: Simulator, Simulationsmodell und Analysator. Die Simulatorkomponente im F4SR beinhaltet einen Generator, der während der Simulation Ereignisse erzeugt. Die Ereignisse werden von dem Scheduler verwaltet. Ereignisroutinen des Simulators, die die Ereignisse bearbeiten, lösen Aktionen innerhalb des Simulationsmodells aus. Zur Verwaltung der Simulationszeit wird die Subkomponente Uhr verwendet. Die Aktualisierung der Simulationszeit erfolgt in Abhängigkeit der ausgeführten Aktion.

Das Simulationsmodell bildet die Struktur des verteilten Systems ab, d.h. die Komponenten mit Replikat, und modelliert die internen Vorgänge. Dazu sind interne Subkomponenten definiert: Der Replikationsmanager verwaltet die existierenden Datenobjekte und verarbeitet alle Schreib- und Lesezugriffe. Die Koordination der Schreib- und Lesezugriffe wird von der Replikationsstrategie vorgegeben, die der Replikationsmanager benutzt. Zusätzliche Replikationsstrategien können dem F4SR per Plugin-Mechanismus hinzugefügt werden (siehe Abschnitt 7.3.3). Der Statusmanager im Simulationsmodell ist für Statusänderungen der Komponenten mit Replikat verantwortlich, z.B. Ausfall einer Komponente oder Lastspitze einer Komponente.

Der Transaktionsmanager im Simulationsmodell merkt sich die an einer globalen Transaktion beteiligten Replikate und verwaltet Schreib- und Lesesperren. Jeder Zugriff auf ein logisches Objekt wird in einer eigenständigen Transaktion durchgeführt, d.h. eine Transaktion besteht aus Zugriffen auf die Replikate eines einzigen logischen Datenobjekts. Jede Transaktion kann per „Commit“ erfolgreich beendet werden oder mittels Rollback zurückgerollt werden. Das Zurückrollen ist dabei mit dem Entwurfsmuster *„Memento“* [GHJ04] realisiert. Angemerkt sei, dass

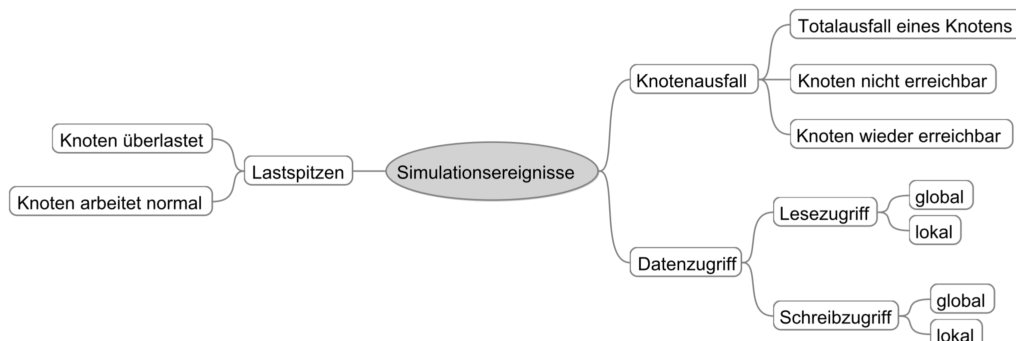

Abbildung 7.4.: Simulationsereignisse des F4SR [Fro06]

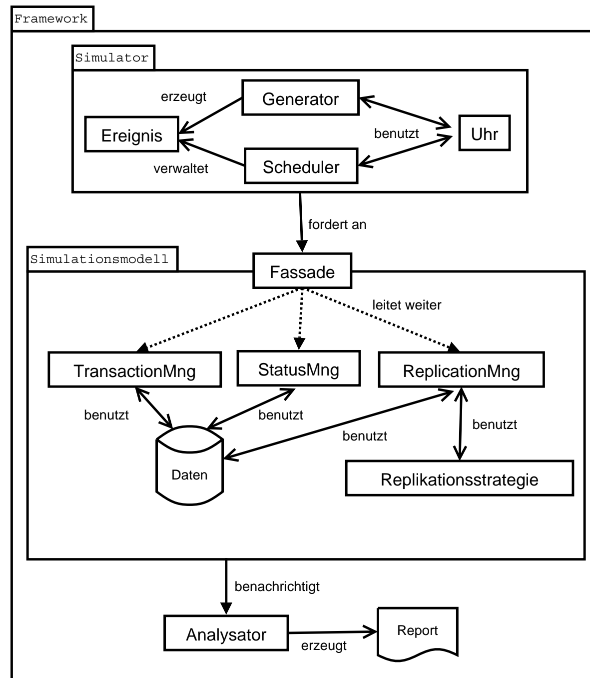

Abbildung 7.5.: Grobe Architektur des F4SR [Fro06]

keine echte Datenbank verwendet wird, sondern dass die Datenbank und ebenfalls die Transaktionsverarbeitung simuliert werden. Nebenläufigkeit wird dadurch simuliert, dass Sperren auf Replikate erst nach dem Eintreten eines Ereignisses für ein Transaktionsende freigegeben werden, sodass zwischenzeitlich andere Transaktionen auf gesperrte Replikate zugreifen könnten und somit einen Fehler verursachen.

Der Analysator sammelt alle Ereignisse und Statusänderungen des Simulationsmodells zum Zweck der Analyse. Er bietet eine grafische Benutzeroberfläche, mit der Kennzahlen der Simulationsexperimente abgerufen werden können. Auch die Darstellung der Ergebnisse in Diagrammen ist möglich. Bei der Analyse werden die oben genannten Metriken verwendet.

Bei der Implementierung des F4SR wurden verschiedene Frameworks genutzt: Das Framework Desmo-J (www.desmoj.de) dient zur Realisierung der Simulationskomponente. Desmo-J stellt Konstrukte zur Durchführung einer ereignisgesteuerten Simulation bereit, wie z.B. das gesamte Scheduling. Das Framework JFreeChart (www.jfree.org) bietet Möglichkeiten zur Visualisierung von Daten, d.h. es eignet sich zur Aufbereitung der Analyseergebnisse. Das Framework Log4J (logging.apache.org) erlaubt eine differenzierbare Protokollierung von Kennzahlen. Mit dem Framework JDOM (www.jdom.org) werden XML-Dateien verarbeitet, die hier zur Persistierung des Simulationsmodells dienen.

Der Test des F4SR wurde von Markus Fromme [Fro06] dadurch vorgenommen, dass die einfachen Replikationsstrategien ROWA, Majority Consensus und Peer-To-Peer (siehe Abschnitt 3.3) implementiert wurden. Weil das Verhalten dieser Replikationsstrategien bekannt ist, konnten die Ergebnisse des Analysators mit entsprechenden Aussagen aus der Literatur (siehe z.B. [DGMD85, Len97, WSP+00, SS05]) abgeglichen werden. Im individuellen Projekt von Nicolai Kalisch [Kal07] zeigte sich, dass der F4SR leicht zu erweitern ist. Auch die Einbindung weiterer Replikationsstrategien, wie CODA und ROAM (siehe Abschnitt 3.3.2), stellte kein Problem dar. Daher bietet sich der F4SR für die Evaluation der Replikationsstrategie RegRess an.

7.3.2. Implementierung des Prototypen KARMA

Bevor auf die Simulation der Replikationsstrategie RegRess eingegangen wird, wird in diesem Abschnitt der Prototyp KARMA vorgestellt. Der Prototyp KARMA dient der Überprüfung der in RegRess spezifizierten Protokolle für Schreib- und Lesezugriffe (siehe Abschnitt 4.2). Dafür beinhaltet der KARMA unter anderem einen Regelinterpreter, der eine Inferenz von Regeln der RRML, Version 2.0, gemäß der Spezifikation des Kapitels 5 durchführen kann. Das Ergebnisziel des experimentellen Prototypen KARMA ist somit, die korrekte Koordination von Zugriffen der Replikationsstrategie RegRess zu zeigen. Daher wird die persistente Datenhaltung der Replikate sowie die Transaktionsverarbeitung im Hauptspeicher simuliert.

In Abbildung 7.6 ist ein vereinfachtes UML-Implementierungsmodell des KARMA dargestellt, in dem die Klassen und die Schnittstellen nur die wichtigsten Methoden ohne Parameter und Rückgabewerte beinhalten. Das vollständige UML-Implementierungsmodell wird im Anhang C gezeigt. Der KARMA wurde mit der Java-Entwicklungsumgebung NetBeans, Version 6.5, entwickelt (NetBeans IDE, http://www.NetBeans.org), die auch eine UML-Modellierung mit Code-Generierung erlaubt. Die Abbildung 7.6 zeigt neben der KARMA-Komponente (Paket `Karma`) auch einen Client (Klasse `KarmaClient`), der Zugriffe auf die Replikate durchführt, sowie einen Transaktionsmanager (Klasse `RDTM`) und die Ressourcenmanager (Klasse `ResourceManager`) mit den zugehörigen Schnittstellen.

Das UML-Implementierungsmodell entspricht der in Abbildung 6.2 auf Seite 155 gezeigten Grob-Architektur des KARMA mit der angebotenen Schnittstelle `ILogicalAccess` für einen Client, der Zugriffe auf logische Objekte durchführt, und den benötigten Schnittstellen `IExtendedTx` eines entsprechenden Transaktionsmanagers und `IPhysicalAccess` eines entsprechenden Ressourcenmanagers. Die Schnittstelle `IExtendedXa`, über die der Transaktionsmanager mit den Ressourcenmanagern kommuniziert, ist ebenfalls berücksichtigt. Die innere Struktur des KARMA ist gemäß der Softwarearchitektur des KARMA realisiert (siehe Abbildung 6.3 auf Seite 156), wobei vier Abweichungen zu nennen sind:

1. Die Artefakte `Replica.Jobs` bzw. `System.log` sind nicht persistiert, sondern werden im Hauptspeicher durch die Komponenten `ReplicaQueueManager` bzw. `DataTransformer` realisiert. Weil der Regelinterpreter `RuleInterpreter` zur Auswertung einer Regelbedingung, die entsprechende Funktionen beinhaltet, einen Zugriff auf diese Daten benötigt, wurden die Schnittstellen `IConsistency` bzw. `ISystemValues` eingefügt, über die der Regelinterpreter benötigte Werte abrufen kann.
2. Das Artefakt `Rules.xml` wurde nicht als XML-Dokument realisiert, sondern als einfache Textdatei (siehe unten).
3. Die Komponente `Timer` und die von der Komponente `RequestHandler` angebotene Schnittstelle `IRequestHandler`, über die der `Timer` die asynchrone Aktualisierung startet, wurden nicht realisiert. Das Starten der asynchronen Aktualisierung wird ausschließlich vom Client initiiert, um die Abläufe besser kontrollieren zu können.
4. Auf die Komponente `RegRessReplicator` wurde verzichtet, d.h. der KARMA wurde als zentrale Komponente implementiert. Auf eine Erweiterung für den verteilten Fall wird am Ende dieses Abschnitts eingegangen.

Der Prototyp KARMA wurde derart implementiert, dass die Anwendung in einem Thread (siehe z.B. [CDK00]) abläuft, d.h. Nebenläufigkeit wird nicht betrachtet. Somit liegen die Transaktionen für Schreib- und Lesezugriffe (siehe unten) in serieller Form vor, weil die Transaktionen nacheinander ausgeführt werden. Damit können keine Inkonsistenzen auftreten, die auf Grund so genannter Nebenläufigkeitsanomalien (siehe Abschnitt 2.2.3) entstehen. Nebenläufigkeitsanomalien treten bei der nebenläufigen Ausführung von Transaktionen auch dann nicht auf, wenn die Ausführung der nebenläufigen Transaktionen äquivalent zu einer seriellen Ausführung der Transaktionen ist (siehe z.B. [TvS02]). Worauf auf eine Erweiterung von nebenläufigen Transaktionen zu achten ist, wird am Ende dieses Abschnitts diskutiert.

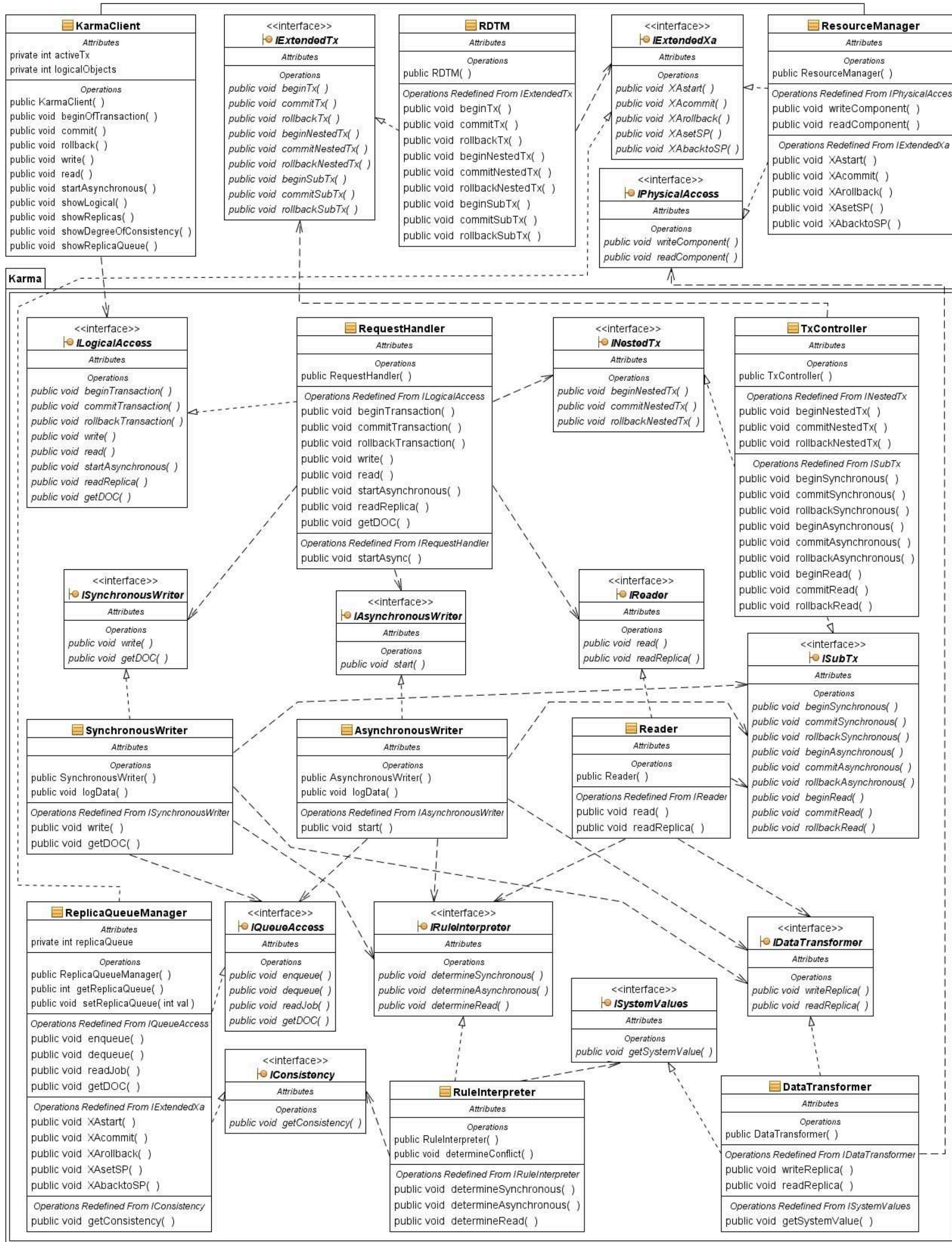

Abbildung 7.6.: Vereinfachtes UML-Implementierungsmodell des KARMA

Der `KarmaClient`, der als Client unter anderem Schreib- und Lesezugriffe auf logische Objekte durchführt (siehe unten), instanziiert zunächst den `Karma`, der seinerseits die `ResourcenManager` und den Transaktionsmanager `RDTM` instanziiert. Im `KarmaClient` wird über Konstanten die Anzahl an Komponenten mit Replikat, d.h. die Anzahl an Ressourcenmanagern, und die Anzahl an logischen Objekten und damit die Anzahl an Replikaten einer Komponente mit Replikat

festgelegt. Weil die Regeln auf diese Konfiguration abgestimmt werden müssen, wurde bei der Implementierung eine konstante Vorgabe gewählt: Es werden 10 Ressourcenmanager mit den Namen „RM-i“, $i = 0, 1, .., 9$ instanziiert, die jeweils 15 Replikate speichern. Der Einfachheit halber wurden die logischen Objekte und damit die Replikate als Integer-Werte definiert, die initial mit dem Wert Null belegt werden. Die Replikate werden nicht in einer Datenbank persistiert, sondern in einem Feld (Array) im Hauptspeicher gespeichert. Zusätzlich speichert ein Ressourcenmanager je Replikat eine Versionsnummer und einen Zeitstempel, der als Sekundenzähler ab Beginn der Anwendung realisiert ist.

Damit der `KarmaClient` die aktuellen Werte der logischen Objekte mit den Replikaten vergleichen kann, benötigt er eine den Replikaten entsprechende Datenstruktur, die ebenfalls einer Transaktionskontrolle unterliegt, um z.B. ein Rücksetzen bei einer fehlgeschlagenen Transaktion zu ermöglichen. Daher instanziiert der `KarmaClient` einen Ressourcenmanager, der die logischen Objekte widerspiegelt (siehe Assoziation vom `KarmaClient` zum `ResourceManager` in Abbildung 7.6). Der Name des Ressourcenmanagers für die logischen Objekte lautet „LogO“.

Neben den Ressourcenmanagern wird als transaktionale Einheit die Replica Queue benötigt (siehe Abschnitt 4.2), die ebenfalls nicht persistent angelegt ist, sondern im Hauptspeicher verwaltet wird. Bei der Instanziierung des `Karma`, genauer gesagt des `ReplicaQueueManager`, wird eine verkette Liste angelegt, deren Elemente die initiierende Komponente, die Komponente mit Replikat als Index, die Nummer des logischen Objekts, den zu schreibenden Wert und einen Zeitstempel speichert. Zum einfachen Navigieren in der Liste wird im `ReplicaQueueManager` ein Iterator gehalten, der einen Durchlauf durch die Liste ermöglicht, sowie eine Referenz auf das letzte Element, die ein schnelles Anhängen eines neuen Elements gestattet.

Die Transaktionsverarbeitung im Prototyp KARMA erlaubt geschlossen-geschachtelte und einfach verteilte Transaktionen, wobei die Verarbeitung simuliert wird. Für die Transaktionsverarbeitung registriert sich jeder der 10 Ressourcenmanager „RM-i“, der Replikate speichert, der Ressourcenmanager „LogO“ für die logischen Objekte und der `ReplicaQueueManager` für die ReplicaQueue beim Transaktionsmanager `RDTM`. Beim `RDTM` handelt es sich nicht um den Prototypen, der in Abschnitt 6.3.2 vorgestellt wurde, sondern um eine einfache Variante, die die entsprechenden Aufrufe des erweiterten XA-Protokolls an die Ressourcenmanager weiterleitet, ohne z.B. eine Synchronisation gemäß dem 2-Phasen-Commit-Protokoll durchzuführen. Dies liegt darin begründet, dass im Prototyp KARMA keine Transaktionsverarbeitung untersucht werden soll und daher keine „echten“ Ressourcen angebunden wurden.

Ein im KARMA implementierter `ResourcenManager` sowie der `ReplicaQueueManager` bieten somit die Schnittstelle `IExtendedXa` an, d.h. sie implementieren die entsprechenden Methoden. Beim Start einer Transaktion, ob einfach verteilt oder geschlossen-geschachtelt, wird eine Kopie der gespeicherten Datenstruktur angelegt, also des Integer-Feldes der Replikate bzw. der logischen Objekte und der Liste für die Replica Queue. Beim Setzen eines Sicherungspunktes (Savepoint) wird analog eine zweite Kopie angelegt. Beim erfolgreichen Ende (Commit) ist dann keine Aktion erforderlich, während beim Zurückrollen (Rollback), ob zum Sicherungspunkt oder komplett, die zuvor angelegte Kopie wieder auf die Originaldaten kopiert werden. Angemerkt sei, dass diese Vorgehensweise nicht bei der Simulation von nebenläufigen Transaktionen funktioniert (siehe unten).

Bei der Instanziierung des Regelinterpreters `RuleInterpreter` im KARMA wird die Regeldatei `rules.txt` eingelesen. Bei der Datei `rules.txt` handelt es sich nicht um ein XML-basiertes Dokument (siehe Abschnitt 5.4), sondern um eine einfache Text-Datei, die die Regeln und die Zuordnung von Objekten zu Replikationseinheiten enthält. Des Weiteren kann ein Grenzwert für fehlerfreie Transaktionen angegeben werden (siehe unten). Beim Prototypen KARMA wurde auf eine XML-basierte Darstellung der Regeln verzichtet, weil kein Austausch zwischen Replikationsmanagern erforderlich ist und die Darstellung der Regeln in der ON-IF-THEN-Darstellung ein einfacheres Testen erlaubt. Ein Beispiel einer Regeldatei ist im Listing 7.1 abgebildet. Dabei ist folgendes Format zu verwenden:

Listing 7.1: Datei rules.txt

```
LimitOfTxError:004

S001 ON update(E00) IF (weekday > 5) THEN async_update(E00)
S001 ON update(E01) IF (weekday > 6) THEN async_update(E01)
S099 ON async_update(K02) IF (true)  THEN sync_update(K02)

A001 ON write(D00) IF (responsetime > 1200) THEN later(D00)

R001 ON read(D00) IF (true) THEN getConsistency(T)
R001 ON read(D00) IF (true) THEN getPerformance(R)

E00
0
1
2
3
4
-1
E01
8
9
-1
```

- Mit dem optionalen Schlüsselwort `LimitofTxError` kann eine Grenze für fehlerfreie Transaktionen bei der asynchronen Aktualisierung gesetzt werden. Wenn die Anzahl fehlerfreier Transaktionen erreicht ist, wird der Transaktionsfehler `XTransactionError` geworfen (siehe unten). Hiermit kann das Protokoll der asynchronen Aktualisierung beim Auftreten von Transaktionsfehlern getestet werden (siehe Zeile 23 des Listings 4.2 auf Seite 74).
- Eine Regel wird in einer Zeile angegeben (siehe z.B. Zeilen 3 bis 10 des Listings 7.1). Das erste Zeichen kodiert die Regelart: „S“ steht für eine Regel, die bei der synchronen Aktualisierung herangezogen wird, „A“ dementsprechend für Regeln der asynchronen Aktualisierung und „R“ für Regeln von Lesezugriffen. Darauf folgt eine dreistellige Zahl, die die Priorität angibt. Abschließend folgt eine Regel gemäß der RRML-Syntax, Version 2.0 (siehe Listing 5.1 auf Seite 117). Der laufende Index im Parameter einer Ereignis- bzw. Aktionsart ist grundsätzlich zweistellig anzugeben.
- Eine Regeleinheit beginnt mit dem Zeichen „E“ gefolgt vom zweistelligen Index. Anschließend können logische Objekte der Replikationseinheit zugeordnet werden, wobei in jeder Zeile ein Index eines logischen Objekts steht (wie bereits erwähnt, wird angenommen, dass die logischen Objekte und Replikate durchnummeriert sind). Das Ende der Zuordnung der logischen Objekte zu einer Replikationseinheit wird durch den nummerischen Wert -1 gekennzeichnet.

Durch die Regeln im Beispiel des Listings 7.1 wird festgelegt, dass logische Objekte der Replikationseinheit `E00` an Wochenenden bzw. der Replikationseinheit `E01` an Sonntagen asynchron aktualisiert werden. Die Komponente `K02` wird jedoch synchron aktualisiert, weil die Regel eine höhere Priorität hat als die beiden anderen Regeln der synchronen Aktualisierung. Anmerkung: In diesem Prototyp wird auf Widerspruchsregeln verzichtet. Beim Erkennen von widersprüchlichen Regeln wird grundsätzlich das Verfahren „PFG“ als Konfliktlösungsstrategie geliefert (siehe Abschnitt 5.3.5), d.h. die Rangfolge der Regeln wird durch Angabe von Prioritäten festgelegt.

Die asynchrone Aktualisierung (siehe Zeile 7 des Listings 7.1) wird nicht durchgeführt, wenn die Komponente, auf der das Replikat gespeichert ist, aktuell eine größere Antwortzeit als 1200 hat (siehe unten). Für Lesezugriffe werden Replikate ausgewählt, die als Parameter übergebene Eigenschaften für Konsistenz, genauer gesagt Zeitabstand, und Performance, genauer gesagt

Listing 7.2: Ablauf der Inferenz beim Regelinterpreter, Version 2.0

```
public Object ermittleInferenzErgebnis(Object Parameterliste) {

  initialisiereErgebnisvariable();

  setzeAuszufuehrendeRegelnAnhandStartereignisse();

  boolean regelAusgefuehrt := true;
  while (regelAusgefuehrt) {
    regelAusgefuehrt := false;
    for (int i = 0; i < anzahlRegeln; i++ {
      Object aktuelleRegel := gibRegel(Regelliste,i);
      if (regelMussNichtAusgefuehrtWerden(aktuelleRegel)) continue;
      if (regelWurdeBereitsAusgefuehrt(aktuelleRegel)) continue;

      setzeRegelAusgefuehrt(aktuelleRegel);
      regelAusgefuehrt := true;
      if (not pruefeBedingung(aktuelleRegel)) continue;

      fuehreAktionAus(aktuelleRegel);
      setzeAuszufuehrendeRegelnAnhandAktionsereignis(aktuelleRegel);
    }
  }
  return Ergebnisvariable;
}
```

Antwortzeit, erfüllen. Abschließend werden im Beispiel die logischen Objekte 0 bis 4 der Replikationseinheit `E00` und die logischen Objekte 8 und 9 der Replikationseinheit `E01` zugeordnet.

Gegenüber dem Regelinterpreter der Version 1.0 wurden die im Abschnitt 5.5 diskutierten Vereinfachungen beim Regelinterpreter der Version 2.0 berücksichtigt. Der hier implementierte `RuleInterpreter` gewährleistet Terminierung dadurch, dass jede Regel höchstens einmal ausgeführt wird. In Listing 7.2 ist der Ablauf der Inferenz dargestellt, der im `RuleInterpreter` der Version 2.0 implementiert ist. Je nach Inferenzart gibt es kleine Abweichungen, aber der hier dargestellte Ablauf ist grundsätzlich bei jeder Inferenzart gleich. Unterscheidungen gibt es auch bei den übergebenen Parametern, sodass im Listing 7.2 allgemein von `Parameterliste` gesprochen wird, die zumindest die initiierende Komponente und das logische Objekt beinhaltet, ggf. noch die Zielkomponente und Eigenschaften bei Lesezugriffen. Nachfolgend werden die einzelnen Aktivitäten des Ablaufs erläutert:

Zeile 3 Entsprechend der Inferenzart wird eine Ergebnisvariable initialisiert (siehe Abschnitte 5.3.2 bis 5.3.5).

Zeile 5 An Hand der Startereignisse (siehe Abschnitte 5.3.2 bis 5.3.5) wird die boolsche Variable `mussAusgefuehrtWerden` jeder Regel entweder mit `true` oder `false` initialisiert. Die boolsche Variable `wurdeAusgefuehrt` wird bei allen Regeln auf `false` gesetzt.

Zeile 7 Die boolsche Variable `regelAusgefuehrt` dient als Kennzeichen, ob eine Regel bei einem erneuten Durchlauf ausgeführt wurde und wird mit `true` initialisiert.

Zeile 8 Der nachfolgende Ablauf wird solange wiederholt, bis keine Regel mehr ausgeführt wurde.

Zeile 9 Das Kennzeichen `regelAusgefuehrt` wird auf `false` gesetzt, d.h. bisher wurde im Durchlauf keine Regel ausgeführt.

Zeile 10 Für jede Regel, die in der Regelliste der entsprechenden Inferenzart aufgeführt ist, wird der nachfolgende Ablauf ausgeführt.

Zeile 11 Die i-te Regel wird als aktuelle Regel gesetzt.

Zeile 12 Wenn die aktuelle Regel nicht ausgeführt werden muss, dann wird mittels `continue` zum nächsten Schleifendurchlauf, d.h. zur nächsten Regel, gesprungen.

Zeile 13 Wenn die aktuelle Regel bereits ausgeführt wurde, dann wird mittels `continue` zur nächsten Regel gesprungen.

Zeile 15 Die boolsche Variable `wurdeAusgefuehrt` der aktuellen Regel wird auf `true` gesetzt, d.h. die aktuelle Regel wird als ausgeführt gekennzeichnet. Das Setzen erfolgt unabhängig von der tatsächlichen Ausführung, die z.B. auf Grund einer negierten Bedingung nicht erfolgt, um Endlosschleifen zu verhindern.

Zeile 16 Das Kennzeichen `regelAusgefuehrt` wird analog auf `true` gesetzt (siehe Zeile 15).

Zeile 17 Die Bedingung der Regel wird ausgewertet. Wenn die Bedingung nicht erfüllt ist, dann wird mittels `continue` zur nächsten Regel gesprungen.

Zeile 19 Die Aktion der aktuellen Regel wird gemäß dem Listing 5.2 (Aktion der synchronen Aktualisierung), dem Listing 5.3 (Aktion der asynchronen Aktualisierung) bzw. dem Listing 5.5 (Aktion eines Lesezugriffs) durchgeführt. Bei der Aktion wird nicht der eigentliche Zugriff durchgeführt, sondern die Ergebnisvariable entsprechend der Aktionsart manipuliert (siehe Abschnitte 5.3.2 bis 5.3.4). Die Manipulation wird nur dann durchgeführt, wenn die Konfliktbehandlung von widersprüchlichen Regeln eine Ausführung erlaubt (siehe Abschnitt 5.3.5), z.B. durch Priorisierung.

Zeile 20 Analog zu den Startereignissen (siehe Zeile 5) werden die Regeln als „auszuführend“ gesetzt, die durch das Aktionsereignis betroffen sind. Somit könnten beim erneuten Durchlauf weitere Regeln ausgeführt werden.

Zeile 23 Das Ergebnis der Inferenz wird zurückgeliefert.

Damit der Regelinterpreter die Bedingungen von Regeln auswerten kann, wurden die Funktionen implementiert, die im Abschnitt 5.3.6 spezifiziert wurden. Die Funktionen für Datum und Uhrzeit verwenden die Systemuhr des Rechners, der die Anwendung ausführt. Die Funktionen zur Identifikation nutzen die an den KARMA übergebenen Parameter, um die initiierende Komponente, die Zielkomponente oder das logische Objekt zu bestimmen. Bei den Funktionen der fachlichen Konsistenzbedingungen wird entweder auf die Schnittstelle des `ReplicaQueueManager` oder des `DataTransformer` zugegriffen, wobei letzterer die Anfrage an die entsprechenden Ressourcenmanager bzw. dessen Replikate delegiert. Der Versionsabstand `diff_version` und der Konsistenzgrad `consistency` kann über die Replica Queue berechnet werden (siehe Abschnitt 4.3.3). Der Zeitabstand `diff_time` wird als Differenz des Zeitstempels des logischen Objekts und des Zeitstempels eines Replikats (siehe oben) berechnet, während für die periodische Aktualisierung `period` die aktuelle Zeit und der Zeitstempel der Replikate herangezogen wird. Die Wertdifferenz `diff_value` basiert auf dem Wert eines aktuellen Replikats und dem Wert des zugegriffenen Replikats.

Die Funktionen für die technischen Konsistenzbedingungen ermitteln ihre Werte auf Basis willkürlich erzeugter Kennzahlen, weil keine echten Daten im Prototyp vorliegen. Die Datengröße, die die Funktion `datasize` liefert, wird als Produkt der Objektnummer mit 100 berechnet. Die Anzahl der Schreib-/Lesekonflikte$_R$ (siehe Definition 15 auf Seite 33) für die Funktion `conflicts` ergibt sich als Summe der Konflikte, die jeder Ressourcenmanager speichert. Die Anzahl Konflikte eines Ressourcenmanagers wird dabei per Zufallszahl inkrementiert, wenn ein neuer Zeitstempel erzeugt wird, und wird auf Null zurückgesetzt, wenn die asynchrone Aktualisierung bei leerer Replica Queue gestartet wird. Die Funktion für die Antwortzeit `responsetime`

wird ebenfalls vom Ressourcenmanager angefordert. Die Antwortzeit eines Ressourcenmanagers wird mit $900+30\cdot i$, $i=$Komponentennummer, initialisiert und bei jedem neuen Zeitstempel um die Sekundenzahl des Zeitstempels erhöht. Die Antwortzeit wird entweder beim Überschreiten des Wertes 1500 zurückgesetzt oder bei der asynchronen Aktualisierung, wenn ein Zugriff auf den Ressourcenmanager durch die Aktion `later` verschoben wird. Durch die genannten Berechnungsvorschriften steigen und fallen die Werte für die Anzahl Konflikte und für die Antwortzeit, sodass mit dem Prototypen KARMA entsprechende Regeln getestet werden können.

Die Komponenten `SynchronousWriter`, `AsynchronousWriter` und `Reader` des KARMA implementieren die im Abschnitt 4.2 spezifizierten Protokolle für Schreib- und Lesezugriffe sowie Hilfsmethoden, um z.B. auf die Replikate oder die Replica Queue direkt lesend zugreifen zu können. Der `DataTransformer` reicht im Wesentlichen die Schreib- und Lesezugriffe auf ein Replikat an die entsprechenden Ressourcenmanager weiter. Da laut Annahme (siehe Abschnitt 4.1.1) keine Schematransformation benötigt wird, erfolgt keine Transformation von Daten. Dafür werden im `DataTransformer` folgende Ausnahmen geworfen, um die oben genannten Protokolle bei Transaktionsfehlern zu testen:

- Ausnahmen bei Schreibzugriffen der synchronen Aktualisierung:
  - Bei der initiierenden Komponente 11 und der Zielkomponente 1 wird ein Transaktionsfehler geworfen.
  - Bei der initiierenden Komponente 12 und der Zielkomponente 2 wird ein Schreibfehler geworfen, der bei wechselfähigen Replikaten einen erneuten Schreibversuch ohne das fehlerverursachende Replikat ermöglicht.
  - Bei der initiierenden Komponente 13 und der Zielkomponente 3 wird auch ein Schreibfehler geworfen, um z.B. die Reaktion bei nicht wechselfähigen Replikaten zu testen.
- Ausnahme bei Schreibzugriffen der asynchronen Aktualisierung: Es kann eine Grenze für fehlerfreie Transaktionen in der Regeldatei gesetzt werden (siehe oben). Beim Erreichen der Grenze wird ein Transaktionsfehler geworfen, sodass laut Protokoll die Bearbeitung des Auftrags der Replica Queue zurückgerollt wird und der nächste Auftrag bearbeitet wird. Die Ausnahme erlaubt unter anderem einen Test auf Einhaltung der chronologischen Verarbeitung von Aufträgen.
- Ausnahme bei Lesezugriffen: Bei der initiierenden Komponente 11 und der Zielkomponente 0 wird ein Transaktionsfehler geworfen, sodass ein anderes Replikat gelesen werden muss, sofern noch Replikate ermittelt wurden, die den gegebenen Eigenschaften genügen.

Der `KarmaClient` ermöglicht einem Anwender die Interaktion mit dem Prototyp KARMA über die Schnittstelle `ILogicalAccess`. Dafür listet der `KarmaClient` die Aufträge der Replica Queue auf, zeigt die aktuellen Werte der logischen Objekte und der Replikate und bietet ein Menü zur Auswahl spezieller Funktionen (siehe Abbildung 7.7). Um Zugriffe auf logische Objekte durchzuführen, muss zunächst eine Transaktion gestartet werden. Bei der Transaktion handelt es sich um eine geschlossen-geschachtelte Transaktion (siehe Abschnitt 6.3.1), die sowohl erfolgreich beendet (Commit) als auch zurückgerollt werden kann (Rollback). Wenn eine Transaktion aktiv ist, können Schreib- oder Lesezugriffe durchgeführt werden. Dazu wird zuerst die initiierende Komponente und das logische Objekt, auf das zugegriffen werden soll, abgefragt. Im Fall eines Lesezugriffs ist zusätzlich die Eingabe von Eigenschaften erforderlich, die die Replikate erfüllen müssen.

Bei einem Schreibzugriff wird als zu schreibender Wert der nächst höhere Integerwert des logischen Objekts gesetzt. Damit entspricht der Wert eines Replikats seiner Versionsnummer. Während der Wert des Replikats jedoch durch Schreibzugriffe geändert wird, wird die Versionsnummer bei jedem Schreibzugriff durch den Ressourcenmanager inkrementiert. Diese Vorgehensweise erlaubt eine Prüfung des Protokolls für die synchronen und asynchronen Aktualisierungen, weil eine Differenz zwischen Wert und Versionsnummer eines Replikats somit eine Verletzung der Chronologie bedeutet und zu einem Systemabbruch führt. Zusätzlich kann auf diese Weise einfach der Versionsabstand im `KarmaClient` abgelesen werden (siehe Abbildung 7.7).

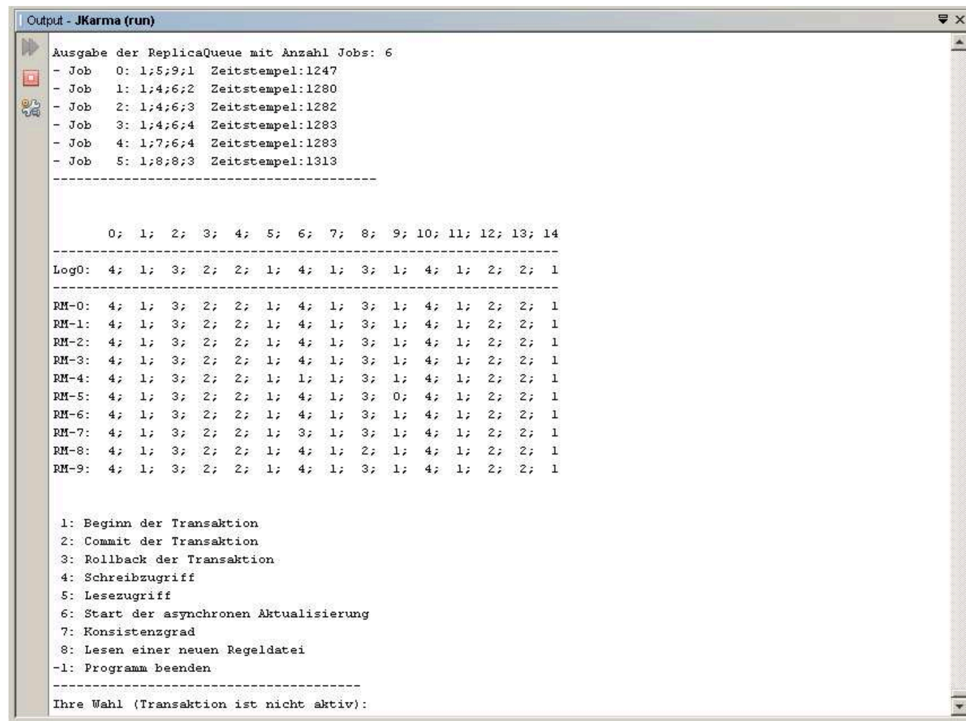

Abbildung 7.7.: Screenshot des Menüs im KARMA-Prototypen

Wenn keine Transaktion aktiv ist, dann kann die asynchrone Aktualisierung gestartet werden. Dadurch werden, sofern möglich, Aufträge aus der Replica Queue abgearbeitet. Wie bereits erwähnt, wird die Anzahl der Konflikte eines Ressourcenmanagers zurückgesetzt, wenn diese Funktion bei leerer Replica Queue aufgerufen wird. Unabhängig davon, ob eine Transaktion aktiv ist oder nicht, kann der Konsistenzgrad abgerufen werden oder die Regeldatei erneut gelesen werden. Der Konsistenzgrad wird letztlich vom `ReplicaQueueManager` geliefert und wird gemäß Definition 40 auf Seite 93 berechnet. Beim Lesen einer neuen Regeldatei wird wiederum die Datei `rules.txt` gelesen (siehe oben). Weil die Datei `rules.txt` zwischenzeitlich geändert werden kann, ist dadurch eine Anpassung der Replikationsstrategie RegRess zur Laufzeit möglich.

Beim Prototyp KARMA werden umfangreiche Log-Daten während der Verarbeitung ausgegeben, um insbesondere die Protokolle für Schreib- und Lesezugriffe und den Regelinterpreter überwachen zu können. In Abbildung 7.8 werden Log-Daten gezeigt, die vornehmlich den Regelinterpreter betreffen. Insbesondere für die Ereignis- und Aktionsarten werden Nummerncodes verwendet, die bei der internen Abspeicherung der Regeln verwendet werden. Diese Nummerncodes sind in der Klasse `KarmaErrors` als Konstanten definiert (siehe Anhang C). Im Beispiel ist eine Transaktion aktiv und es soll ein Schreibzugriff durchgeführt werden, wobei die initiierende Komponente die Komponente 12 ist und das logische Objekt 3 mit dem Wert 1 geschrieben werden soll. Das Beispiel beruht auf folgender Regel:

$$\text{ON} \quad \textit{update(D)} \quad \text{IF} \quad \textit{hour} > 13 \quad \text{THEN} \quad \textit{changeableTrue(D)}$$

Die Log-Daten in Abbildung 7.8 zeigen, dass der `RequestHandler` den Zugriff vom `KarmaClient` an den `SynchronousWriter` weiterreicht. Der `SynchronousWriter` erlangt eine Sperre auf das logische Objekt 3 und ruft anschließend den `RuleInterpreter` auf, der eine Partitionierung der betroffenen Replikate ermittelt. An den Log-Daten des `RuleInterpreter` spiegelt sich der im Listing 7.2 auf Seite 191 spezifizierte Ablauf wider. Zusätzlich zeigen die Log-Daten detaillierte Informationen, z.B. wurde durch die Startereignisse die oben genannte Regel als „aus-

Output - JKarma (run)

```
Eingabe initiierende Komponente im Bereich 0 bis 14: 12
Eingabe logisches Objekt im Bereich 0 bis 14: 3
Initiierende Komponente: 12  Logisches Objekt: 3   alter Wert: 0  zu schreibender Wert:1
RequestHandler: write
SynchronousWriter: write
SynchronousWriter: setLock auf logisches Objekt: 3
RuleInterpreter: determineSynchronous: (Komponente/Objekt): 12;3
RuleInterpreter: setStartEvents: 100
RuleInterpreter: setRuleMustBeExecuted: 100 D 0
Gesetzte Regel:0:S 1 ON 100 D 0 IF (hour>13) THEN 103 E 0
RuleInterpreter: setRuleMustBeExecuted: 100 E 0
RuleInterpreter: setRuleMustBeExecuted: 100 O 3
RuleInterpreter: setRuleMustBeExecuted: 100 K 12
Auszuführende Regel:0:S 1 ON 100 D 0 IF (hour>13) THEN 103 E 0
RuleInterpreter: checkCondition: (hour>13) für (InitKomponente;Komponente;Objekt: 12;-1;3
RuleInterpreter: computeArith: hour>13: für Komponente;Objekt: -1;3
RuleInterpreter: hour
RuleInterpreter: systemTimeValue: HH:15
Arithmetischer Ausdruck: 15.0 OP(3) 13
RuleInterpreter: computeAffectedReplicas:
RuleInterpreter: handleConflictSynchronous:
RuleInterpreter: determineConflict: liefert PFG: Priorität, dann feinste Granularitaet
RuleInterpreter: handleConflictSynchronous: liefert: Action
RuleInterpreter: setPartitionAttributes: 103   betroffene Replikate: -2
RuleInterpreter: setRuleMustBeExecuted: 103 E 0
RuleInterpreter: setRuleMustBeExecuted: 103 O 3
RuleInterpreter: setRuleMustBeExecuted: 103 K 0
RuleInterpreter: setRuleMustBeExecuted: 103 K 1
RuleInterpreter: setRuleMustBeExecuted: 103 K 2
RuleInterpreter: setRuleMustBeExecuted: 103 K 3
RuleInterpreter: setRuleMustBeExecuted: 103 K 4
RuleInterpreter: setRuleMustBeExecuted: 103 K 5
RuleInterpreter: setRuleMustBeExecuted: 103 K 6
RuleInterpreter: setRuleMustBeExecuted: 103 K 7
RuleInterpreter: setRuleMustBeExecuted: 103 K 8
RuleInterpreter: setRuleMustBeExecuted: 103 K 9

      0;  1;  2;  3;  4;  5;  6;  7;  8;  9
-------------------------------------------
Sync:  T;  T;  T;  T;  T;  T;  T;  T;  T;  T
Chan:  T;  T;  T;  T;  T;  T;  T;  T;  T;  T
-------------------------------------------
SynchronousWriter: checkChronologyError
ReplicaQueueManager: checkInReplicaQueue für Komponente: 0  ObjektNr: 3  FALSE
ReplicaQueueManager: checkInReplicaQueue für Komponente: 1  ObjektNr: 3  FALSE
```

Abbildung 7.8.: Screenshot der Log-Daten des KARMA-Prototypen: Regelinterpreter

zuführend“ gesetzt. Nach Abschluss der Inferenz wird das Ergebnis als Log-Datum angezeigt: Alle Replikate müssen synchron aktualisiert werden, sind aber wechselfähig. Danach kann der `SynchronousWriter` seine Verarbeitung fortsetzen, indem zunächst geprüft wird, ob die synchron zu aktualisierenden Replikate die Chronologie verletzen.

In Abbildung 7.9 ist ein weiterer Teil der Log-Daten zum oben genannten Beispiel abgebildet, wobei hier wesentliche Teile des Protokolls eines synchronen Schreibzugriffs zu sehen sind (siehe Listing 4.1 auf Seite 71). Wenn der Schreibzugriff erfolgen kann, wird zunächst ein Sicherungspunkt (Savepoint, `XAsetSP`) gesetzt, wobei in der Abbildung 7.9 nur die letzten Ressourcenmanager der Operation zu sehen sind. Der `SynchronousWriter` ruft für jedes synchron zu aktualisierende Replikat den `DataTransformer` auf, der seinerseits den Aufruf an den entsprechenden `ResourceManager` delegiert. Weil die Komponente 12 die initiierende Komponente ist, wird ein Schreibfehler bei der zweiten Komponente mit Replikat geworfen (siehe oben). Infolge der Wechselfähigkeit der Replikate kann das Replikat R_2^3 asynchron aktualisiert werden.

Laut Protokoll für synchrone Schreibzugriffe wird nun die Teiltransaktion bis zum Sicherungspunkt zurückgerollt. Der Sicherungspunkt wird erneut gesetzt (Anmerkung: hier nicht zwingend erforderlich, aber insbesondere bei offen-geschachtelten Transaktionen). Anschließend werden die synchron zu aktualisierenden Replikate geschrieben und ein Auftrag an die Replica Queue angehängt. Ein Auftrag enthält die initiierende Komponente (12), die Zielkomponente (2), die Objektnummer (3) und den Wert (1). Zusätzlich erhält der Auftrag einen Zeitstempel. Abschließend sei angemerkt, dass sich die Log-Daten ausblenden lassen, indem in der Methode `message` der Klasse `IO` die entsprechende Anweisung auskommentiert wird.

Der experimentelle Prototyp KARMA zeigt, dass die Spezifikation des KARMA (siehe Kapitel 6) sich technisch umsetzen lässt. Die regelbasierte Koordination, genauer gesagt, die Bestim-

```
Output - JKarma (run)
ResourceManager RM-8: XAsetSP
ResourceManager RM-9: XAsetSP
ResourceManager Log0: XAsetSP
DataTransformer: writeReplica
ResourceManager RM-0: writeComponent   Objekt-Nummer: 3  Wert: 1
DataTransformer: writeReplica
ResourceManager RM-1: writeComponent   Objekt-Nummer: 3  Wert: 1
DataTransformer: writeReplica
--> bei der initiierenden Komponente 12 und ZielKomponente 2 wird immer ein Schreibfehler geworfen
--> Komponente: 2 wird nun wegen Nicht-Verfügbarkeit asynchron aktualisiert!
TxController: rollbackSynchronous
RDTM: rollbackSubTx
ReplicaQueueManager: XAbacktoSP
ReplicaQueueManager: getSize
ResourceManager RM-0: XAbacktoSP
ResourceManager RM-1: XAbacktoSP
ResourceManager RM-2: XAbacktoSP
ResourceManager RM-3: XAbacktoSP
ResourceManager RM-4: XAbacktoSP
ResourceManager RM-5: XAbacktoSP
ResourceManager RM-6: XAbacktoSP
ResourceManager RM-7: XAbacktoSP
ResourceManager RM-8: XAbacktoSP
ResourceManager RM-9: XAbacktoSP
ResourceManager Log0: XAbacktoSP
TxController: beginSynchronous
RDTM: beginSubTx
ReplicaQueueManager: XAsetSP
ResourceManager RM-0: XAsetSP
ResourceManager RM-1: XAsetSP
ResourceManager RM-2: XAsetSP
ResourceManager RM-3: XAsetSP
ResourceManager RM-4: XAsetSP
ResourceManager RM-5: XAsetSP
ResourceManager RM-6: XAsetSP
ResourceManager RM-7: XAsetSP
ResourceManager RM-8: XAsetSP
ResourceManager RM-9: XAsetSP
ResourceManager Log0: XAsetSP
DataTransformer: writeReplica
ResourceManager RM-0: writeComponent   Objekt-Nummer: 3  Wert: 1
DataTransformer: writeReplica
ResourceManager RM-1: writeComponent   Objekt-Nummer: 3  Wert: 1
ReplicaQueueManager: enqueue: 12;2;3;1
DataTransformer: writeReplica
ResourceManager RM-3: writeComponent   Objekt-Nummer: 3  Wert: 1
DataTransformer: writeReplica
```

Abbildung 7.9.: Screenshot der Log-Daten des KARMA-Prototypen: Protokoll Schreibzugriff

mung der Menge der betroffenen Replikate bei einem Zugriff auf Basis von Replikationsregeln, konnte überprüft werden. Die Ausdrucksmächtigkeit der RRML (siehe Kapitel 5) konnte untersucht werden, wobei festgestellt werden kann, dass mit den Regeln der RRML eine hohe Flexibilität gegeben ist. Es können unterschiedliche Replikationsstrategien nachgebildet oder erweitert werden (siehe Abschnitt 7.3.4). Es hat sich gezeigt, dass der Regelinterpreter `RuleInterpreter` in der Version 2.0 effizienter arbeitet als der Regelinterpreter der Version 1.0, weil die Terminierung sich durch das einmalige Ausführen von Regeln ergibt. Durch diese Arbeitsweise wurden keine Nachteile während der Tests erkennbar. Letztlich konnte gezeigt werden, dass die Protokolle für Schreib- und Lesezugriffe (siehe Abschnitt 4.2) technisch realisierbar sind. Sofern entsprechende Regeln spezifiziert werden, ergibt sich eine höhere Anzahl an erfolgreichen Transaktionen, weil Teiltransaktionen neu aufgesetzt werden können (siehe Beispiel oben). Allerdings ist hierfür eine aufwändigere Transaktionsverarbeitung nötig.

Weil im Prototyp KARMA der Client `KarmaClient` die Transaktionen sequentiell nacheinander ausführt, werden serielle Schedules [GR93] erzeugt. Die nebenläufige Ausführung von Transaktionen muss gewährleisten, dass die so entstehenden Schedules zu einem seriellen Schedule äquivalent sind. Ein bewährtes Mittel, mit dem nebenläufige Transaktionen derart synchronisiert werden können, dass serielle Schedules entstehen, sind Sperrverfahren (siehe Abschnitt 2.2.3). Im Wesentlichen ist im KARMA eine Stelle bei Nebenläufigkeit betroffen, nämlich die Sperren auf logische Objekte (siehe Zeile 7 des Listings 4.1 auf Seite 71), um die chronologische Verarbeitung von Zugriffen zu garantieren. Die Verwaltung der Sperren für logische Objekte kann z.B. auf eine entsprechende Datenbank ausgelagert werden. Eine andere Möglichkeit in der Programmiersprache Java bieten so genannte synchronisierte Blöcke (Schlüsselwort `synchronized`,

[Krü00]). In diesen synchronisierten Blöcken muss dann das Setzen und Freigeben der Sperren erfolgen.

Die Ressourcenmanager der Komponenten mit Replikat, genauer gesagt, der zu Grunde liegenden Datenbanken, und die Ressourcenmanager von transaktionalen Warteschlangen (siehe Abschnitt 2.2.5) verwalten im allgemeinen ebenfalls Sperren zur Synchronisation, sodass hierauf nicht eingegangen werden muss. Liefert ein Ressourcenmanager z.B. einen Schreibfehler auf Grund einer Sperre, dann ist im Prototypen KARMA eine entsprechende Reaktion implementiert (siehe oben). Wenn ein derartiges Sperrverfahren im Prototypen KARMA implementiert werden soll, dann muss wie bei den Sperren für logische Objekte mit synchronisierten Blöcken gearbeitet werden. Weiterhin ist dann kein komplettes Kopieren der Felder möglich, die die Replikate speichern, sondern es müssen einzelne Werte zurückgesetzt werden. Dafür könnte z.B. das Entwurfsmuster *„Memento“* [GHJ04] eingesetzt werden.

Weiterhin wurde in dem hier vorgestellten Prototypen auf eine Verteilung des KARMA verzichtet, d.h. die Komponente `RegRessReplicator` wurde nicht implementiert (siehe Abschnitt 6.2). Allgemein bedeutet eine derartige Erweiterung, dass die benötigten Daten des `ReplicaQueueManagers` repliziert werden müssen. Wenn hierfür z.B. das ROWA-Verfahren (siehe Abschnitt 3.3.1) verwendet wird, dann müssen die entsprechenden Daten in jeder Instanz des KARMA geschrieben werden, z.B. das Anhängen von Aufträgen in die jeweiligen Replica Queues. Demzufolge müssen auch bei den weiteren Instanzen entsprechende Ressourcenmanager implementiert sein, die sich bei einem Transaktionsmanager registrieren. Bezogen auf den hier implementierten Prototypen bedeutet das, dass sich jeder `ReplicaQueueManager`, der die Schnittstelle `IExtendedXa` implementieren muss, beim `RDTM` registriert.

7.3.3. Das KARMA-Plugin für das Simulationsframework F4SR

Das Simulationsframework F4SR (siehe Abschnitt 7.3.1) bietet die Möglichkeit, per Plugin-Mechanismus weitere Replikationsstrategien einzubinden, die dann mit dem Simulator analysiert werden können. Bisher stehen die Replikationsstrategien ROWA, Majority Consensus, Peer-To-Peer, CODA und ROAM zur Verfügung (siehe Abschnitt 3.3). In diesem Abschnitt wird das Plugin beschrieben, mit dem der in Abschnitt 7.3.2 beschriebene KARMA, der die Replikationsstrategie RegRess implementiert, eingebunden werden kann. Die grundsätzliche Vorgehensweise beim Einbinden von weiteren Replikationsstrategien in den F4SR ist in der Diplomarbeit von Markus Fromme beschrieben [Fro06].

Damit das KARMA-Plugin im Simulator aufgerufen werden kann, muss die neue Replikationsstrategie in der Benutzeroberfläche des F4SR bekannt gegeben werden. Dazu bietet die Komponente `Simulator` im Paket `f4sr.simulator.gui` das Aufzählungsobjekt `STRATEGIES`, dem der Name „RegRess“ hinzugefügt wird. Beim Generieren des Simulationsmodells, das in der Benutzeroberfläche angestoßen wird, wird die ausgewählte Replikationsstrategie in der Methode `createReplicationStrategy()` der Komponente `Simulator` instanziiert. Vor der Instanziierung können an dieser Stelle weitere Eingaben abgefragt werden, die von der ausgewählten Replikationsstrategie benötigt werden.

Im Falle der Replikationsstrategie RegRess bzw. des KARMA ist die Eingabe eines Zeitintervalls nötig, dass die Zeitperiode für den Start der asynchronen Aktualisierung bestimmt (siehe Komponente `Timer` im Abschnitt 6.2). Derartige Konfigurationsdaten der Replikationsstrategie werden dem Konstruktor der Komponente, die die Replikationsstrategie implementiert, als Parameter mitgegeben. Angemerkt sei, dass im Gegensatz zu den bisher implementierten Plugins, die die jeweilige Replikationsstrategie in einer Komponente bzw. Klasse bereitstellen, das KARMA-Plugin aus mehreren Komponenten bzw. Klassen besteht. Das KARMA-Plugin entspricht im Wesentlichen dem in Abschnitt 7.3.2 vorgestellten Replikationsmanager KARMA, d.h. das KARMA-Plugin basiert auf dem in Abbildung 7.6 auf Seite 188 dargestellten Paket `KARMA`, dessen Komponenten für das Plugin im Paket `f4sr.simulator.model.control.strategy` un-

tergebracht sind. Die Anpassungen, die für ein Plugin des F4SR notwendig waren, werden nachfolgend beschrieben.

Die Komponente `RequestHandler` des `KARMA`-Pakets dient als Fassade zur Benutzung der Komponenten des KARMA. Daher wird dem F4SR der `RequestHandler` in der oben genannten Methode `createReplicationStrategy()` als Referenz auf die Replikationsstrategie RegRess bzw. auf den KARMA bekannt gegeben. Bei der Instanziierung des `RequestHandler` werden analog zur Beschreibung in Abschnitt 7.3.2 die weiteren Komponenten des KARMA instanziiert. Weil der F4SR über eine eigene, vereinfachte Transaktionsverarbeitung verfügt (siehe Abschnitt 7.3.1), die kein Registrieren von XA-fähigen Ressourcen erlaubt, kann sich der `ReplicaQueueManager` nicht bei einem Transaktionsmanager registrieren, d.h. die Zugriffe auf die Replica Queue unterliegen keiner Transaktionskontrolle. Demzufolge kann auch der `TxController` des KARMA-Plugins die entsprechenden Funktionen zur Transaktionsverarbeitung nicht an den Transaktionsmanager des Simulationsmodells im Sinne der Schnittstelle `IExtendedTx` (siehe Abbildung 7.6 auf Seite 188) weiterreichen. Die Methoden des `TxController` werden zwar weiterhin aufgerufen, haben aber keine Funktionalität.

Damit der Replikationsmanager des Simulationsmodells eine Replikationsstrategie nutzen kann, muss die Schnittstelle `IReplicationStrategy` des F4SR implementiert werden. Somit bietet der `RequestHandler` anstelle der Schnittstelle `ILogicalAccess` (siehe Abbildung 7.6 auf Seite 188) im KARMA-Plugin die Schnittstelle `IReplicationStrategy` an, d.h. folgende Methoden sind im `RequestHandler` implementiert worden (bei den nachfolgenden Methoden wurde der Einfachheit halber auf die Parameterliste verzichtet):

- `write()`: Als Simulationsereignis ist ein Schreibzugriff auf ein logisches Objekt aufgetreten. Der Replikationsmanager des Simulationsmodells leitet dementsprechend einen Schreibzugriff an die Replikationsstrategie, d.h. im Falle des KARMA-Plugins an den `RequestHandler`. Die Methode `write()` des `RequestHandler` ruft für den Schreibzugriff die Methode `write()` der Komponente `SynchronousWriter` auf. Nach der Durchführung der Operation werden dem Analysator des F4SR vom `RequestHandler` entsprechende Protokollinformationen sowohl im erfolgreichen Fall als auch im Fehlerfall übermittelt.
- `read()`: Wie bei der Methode `write()`, jedoch wird der Aufruf vom `RequestHandler` nicht an die Komponente `Reader` des KARMA-Plugins weitergeleitet. Der Aufruf wird als lokaler Aufruf simuliert bzw. als Aufruf auf ein beliebiges Replikat (siehe Abschnitt 4.2.3). Somit können während der Simulation Lesezugriffe auf veraltete Replikate als Schreib-/Lesekonflikte$_R$ (siehe Definition 15 auf Seite 33) gezählt werden. Auch in der Methode `read()` werden dem Analysator des Simulationsmodells entsprechende Protokollinformationen geliefert.
- `getLastAccessPeriod()`: Diese Methode liefert die Antwortzeit des letzten Zugriffs und wird vom Analysator des F4SR benötigt.
- `initialize()`: Beim KARMA-Plugin dient diese Methode dazu, ein *„RegRess“*-Ereignis in die Ereignisliste des Simulators einzutragen, wobei das Zeitintervall für die Ausführungsperiode der asynchronen Aktualisierung verwendet wird. Beim Auslösen des *„RegRess“*-Ereignisses durch den Simulator wird die Methode `onRegRessEvent()` des `RequestHandler` aufgerufen. Die Methode `onRegRessEvent()` startet die asynchrone Aktualisierung mittels der Methode `startAsynchronous()` des `RequestHandler` und erzeugt abschließend ein neues *„RegRess“*-Ereignis, damit die asynchrone Aktualisierung nach Ablauf des Zeitintervalls erneut aufgerufen wird. Hierdurch wird die `Timer`-Komponente realisiert (siehe Abschnitt 6.2).
- `isUsingReadLocks()` und `isSynchronous()`: Im KARMA-Plugin sind diese Methoden der Schnittstelle `IReplicationStrategy` nicht von Bedeutung.

Gegenüber der KARMA-Implementierung im Abschnitt 7.3.2, bei der der `DataTransformer` die Referenzen zu den Schnittstellen der Ressourcenmanager (`IPhysicalAccess`) speichert, werden diese Referenzen beim KARMA-Plugin als Parameter an die Methoden `write()` und `read()` des `RequestHandler` übergeben, der sie seinerseits an die entsprechenden Komponenten durchreicht. Neben der Liste der Komponenten mit Replikat wird die initiierende Komponente übergeben.

Eine Komponente, die im F4SR `DBNode` genannt wird, speichert eine Liste von Replikaten. Ein Replikat speichert einen Wert als String-Typ und kann eine Lese- und Schreibsperre setzen. Weil sowohl die Komponenten als auch die Replikate im F4SR nummeriert sind, können sie, wie im Abschnitt 7.3.2 vorausgesetzt, über einen Index identifiziert werden.

Damit das Konsistenzverhalten von RegRess während der Simulation beobachtet werden kann, wird im `RequestHandler` nach jeder synchronen Aktualisierung (`write()`-Methode) und nach jeder asynchronen Aktualisierung (`startAsynchronous()`-Methode) der Konsistenzgrad (siehe Definition 40 auf Seite 93) der replizierten Datenbank berechnet. Neben dem aktuellen Konsistenzgrad, der auf Basis der Replica Queue gemäß Gleichung 4.12 auf Seite 94 berechnet wird, wird der gemittelte Konsistenzgrad gemäß Gleichung 4.6 auf Seite 93 ermittelt und laufend im Konsolenfenster ausgegeben. Hierdurch kann die Entwicklung des Konsistenzgrades im eingeschwungenen Zustand der Simulation beobachtet werden. Die Ausgabe kann in der Methode `computeDOC()` abgeschaltet werden. Auch die Log-Informationen, die während der Ausführung des KARMA-Plugins analog zu der Implementierung, die in Abschnitt 7.3.2 beschrieben wurde, ausgegeben werden, können über die `message()`-Methode ein- oder abgestellt werden.

Die Methode `write()` der Komponente `SynchronousWriter` bleibt im KARMA-Plugin unverändert und implementiert das Protokoll für den synchronen Schreibzugriff (siehe Listing 4.1 auf Seite 71). Jedoch muss hier streng darauf geachtet werden, dass zunächst die synchron zu aktualisierenden Replikate geschrieben werden und dann erst die Aufträge in die Replica Queue eingetragen werden. Der Grund hierfür liegt in der eingeschränkten Transaktionsverarbeitung des F4SR, die keine Registrierung der Replica Queue als XA-fähige Ressource erlaubt. Daher werden Transaktionsfehler nur bei den Komponenten mit Replikat simuliert und die Manipulation der Replica Queue wird als korrekt angenommen, d.h. das Zurückrollen einer Transaktion betrifft bei Einhaltung oben genannter Reihenfolge nur die Komponenten mit Replikat.

Die Simulation von Nebenläufigkeit wird im F4SR, wie bereits in Abschnitt 7.3.1 erwähnt, wie folgt realisiert: Der Replikationsmanager des Simulationsmodells startet eine Transaktion, die durch eine eindeutige Identifikationsnummer identifiziert wird. Dann wird der Zugriff auf ein logisches Objekt an die Komponente delegiert, die die gewählte Replikationsstrategie implementiert, z.B. bei RegRess an den `RequestHandler`. Durch den Zugriff werden Sperren auf die Objekte gesetzt. Abschließend stellt der Replikationsmanager des Simulationsmodells ein *„Transaktionsende“*-Ereignis in die Ereignisliste, wodurch ein Zeitintervall für die Verarbeitung der Transaktion simuliert wird, weil die Sperren bis zur Abarbeitung des Ereignisses gehalten werden. Wenn vor diesem *„Transaktionsende“*-Ereignis eine andere Transaktion gestartet wird, dann kann es somit zu simulierten Konflikten kommen, weil möglicherweise auf gesperrte Replikate zugegriffen wird.

In der Methode `start()` der Komponente `AsynchronousWriter`, die das Protokoll für die asynchrone Aktualisierung (siehe Listing 4.2 auf Seite 74) implementiert, wird auf die Simulation von Nebenläufigkeit verzichtet, weil ansonsten die Abarbeitung der Replica Queue behindert würde. Bei der asynchronen Aktualisierung wird gemäß Protokoll jeder Auftrag der Replica Queue in einer eigenen Transaktion bearbeitet. Wenn nun Sperren auf Replikate gesetzt werden, die erst nach dem Auslösen eines weiteren Ereignisses, also nach der aktuellen Bearbeitung der asynchronen Aktualisierung, freigegeben werden, dann würden weitere Aufträge, die das gleiche Replikat schreiben, blockiert werden, d.h. je Start der aktuellen Aktualisierung würde jedes Replikat höchstens einmal geschrieben werden können. Daher wird bei der asynchronen Aktualisierung die oben genannte Simulation von Nebenläufigkeit nicht verwendet. Es wird kein *„Transaktionsende“*-Ereignis erzeugt, sondern nach jedem Zugriff wird die Transaktion entweder erfolgreich beendet oder zurückgerollt, sodass die Sperren wieder freigegeben werden. Auch in diesem Fall koordiniert nicht der `TxController` die Transaktion (siehe oben), sondern der Transaktionsmanager des Simulationsmodells.

Der Regelinterpreter `RuleInterpreter` bleibt im KARMA-Plugin im Wesentlichen gegenüber der Beschreibung in Abschnitt 7.3.2 unverändert, so wird z.B. auch hier die Regeldatei mit

dem Namen `rules.txt` gelesen. Weil während der Simulation Antwortzeiten erzeugt werden, die von Simulationsereignissen abhängen (siehe Abbildung 7.4 auf Seite 185), und Konflikte gezählt werden, können die entsprechenden Funktionen von Bedingungen (siehe Listing 5.1 auf Seite 117) auf diese Werte zugreifen, d.h. die Funktion `responsetime` bezieht die Antwortzeit von den Komponenten mit Replikat (im F4SR `DBNode` genannt) und die Funktion `conflicts` erhält die Anzahl Konflikte von dem `Analysator`, der diesen Wert neben anderen Kennzahlen zur Analyse des Simulationslaufs speichert. Die Funktion `diff_value` ist im KARMA-Plugin auskommentiert, weil im Simulationsmodell die Replikate keine nummerischen Werte, sondern Zeichenketten speichern. Sinnvolle Werte für die Funktion `diff_time` bzw. `period` ergeben sich nur dann, wenn der Simulationslauf „gebremst" abläuft, z.B. dadurch, dass Log-Informationen ausgeben werden, die zusätzlich die Eingabe von `<Return>` zur Fortsetzung erfordern.

Gegenüber dem Prototypen KARMA (siehe Abschnitt 7.3.2), der über gezielte Schreib- und Lesezugriffe das Testen der implementierten Protokolle und der Funktionalität des Regelinterpreters erlaubt, dient das KARMA-Plugin eher dafür, mittels des F4SR die Auswirkung von unterschiedlichen Regeln auf das Konsistenzverhalten zu analysieren. Zwar können auch im KARMA-Plugin Log-Informationen ausgegeben werden, die die korrekte Implementierung demonstrieren, aber bei der hier eher großen Anzahl an Zugriffen wird eine große Menge an Log-Informationen erzeugt. Während jedoch im Prototypen KARMA das Konsistenzverhalten, insbesondere der Konsistenzgrad (siehe Abschnitt 4.3.3), auf Grund der einzeln einzugebenden Zugriffe schwerlich zu untersuchen ist, zeigt der F4SR mit dem KARMA-Plugin hier seine Stärke, weil durch die Generierung von Ereignissen und die umfangreichen Konfigurationsmöglichkeiten verschiedene Szenarien simuliert werden können. Hierauf wird im folgenden Abschnitt eingegangen.

7.3.4. Evaluation der Simulationsexperimente

In diesem Abschnitt folgt die Evaluation der Replikationsstrategie RegRess bzw. dem Replikationsmanager KARMA, der die Replikationsstrategie RegRess mit den zugehörigen Protokollen für Schreib- und Lesezugriffe und insbesondere einen Regelinterpreter für die Regelsprache RRML implementiert. Dabei wird auf Simulationsläufe des F4SR mit dem KARMA-Plugin (siehe Abschnitt 7.3.3) eingegangen. Die Bewertung einer Replikationsstrategie erfolgt häufig hinsichtlich der Ziele der Replikation (siehe Abschnitt 3.1): Konsistenz, Performance und Verfügbarkeit. Im Allgemeinen kann gesagt werden, dass die Verbesserung eines Ziels eine Verschlechterung der anderen Ziele nach sich zieht, sodass von Zielkonflikten der Replikation gesprochen wird (siehe Abbildung 3.1 auf Seite 28).

Die Koordination von Zugriffen erfolgt bei RegRess regelbasiert, d.h. je nachdem, welche Regeln spezifiziert wurden, ergibt sich im Allgemeinen ein verändertes Verhalten von RegRess hinsichtlich Konsistenz, Performance und Verfügbarkeit. Weil in den Regeln neben Zeitangaben auch Systemkennzahlen berücksichtigt werden, ergibt sich ein dynamisches, adaptives Verhalten. Diese Flexibilität von RegRess, die durch die Sprachmächtigkeit der RRML gegeben ist, erlaubt unterschiedlichste Priorisierung der Ziele Konsistenz, Performance und Verfügbarkeit. Daher sind konkrete Aussagen zu RegRess nur dann möglich, wenn auf die Regelmenge Bezug genommen wird, die der aktuellen Konfiguration zu Grunde liegt. Daher folgt in diesem Abschnitt ein Vergleich zu anderen Replikationsstrategien bzw. den verwandten Arbeiten (siehe Abschnitt 3.4), indem gezeigt wird, mit welchen Regeln die entsprechenden Replikationsstrategien nachgebildet bzw. erweitert werden können. Zuvor wird kurz auf die Analysemöglichkeiten des F4SR unter Berücksichtigung des KARMA-Plugins eingegangen.

Der F4SR bietet als Benutzeroberfläche (GUI) drei Menüs: Settings, System und Simulation. Während in den Menüs Settings und System Eingaben erforderlich sind (siehe Abbildung 7.10), dient das Menü Simulation lediglich zum Starten und Abbrechen eines Simulationslaufs. Im Menü Settings wird die zu analysierende Replikationsstrategie ausgewählt. Weiterhin kann die Simulationszeit, das Schreib-/Leseverhältnis, Zeiten für verschiedene Simulationsereignisse und

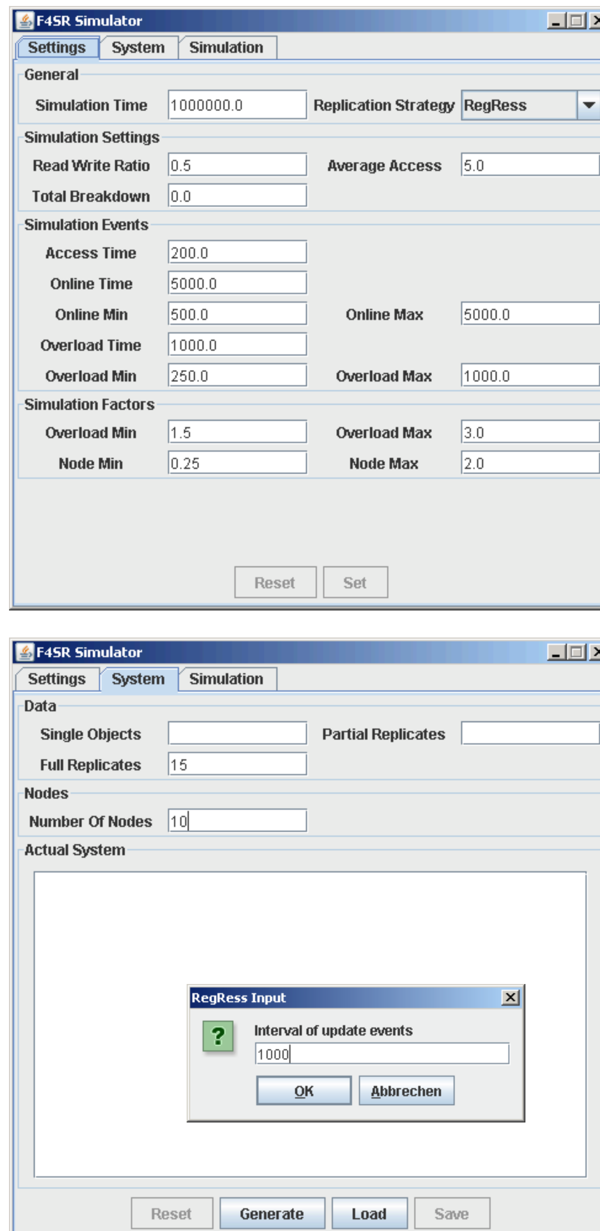

Abbildung 7.10.: F4SR-Menüs „Settings“ und „System“

Faktoren für die Antwortzeit von Knoten bzw. deren Überlastungswahrscheinlichkeit gesetzt werden. Die Zeit wird im F4SR in Zeiteinheiten eingegeben und gemessen. Für eine detaillierte Beschreibung der Eingabemöglichkeiten sei auf die Diplomarbeit von Markus Fromme [Fro06] verwiesen.

Im F4SR-Menü System wird das verteilte System konfiguriert, indem die Anzahl Replikate und Komponenten mit Replikat (im F4SR Nodes genannt) festgelegt werden. Weil in dieser Dissertation laut Annahme von einer voll-replizierten Datenbank ausgegangen wird (siehe Abschnitt 4.1.1), sind nur Werte in den Feldern „Full Replicates“ und „Number of Nodes“ einzugeben. Hierbei ist zu beachten, dass diese Werte bei der Replikationsstrategie RegRess auf die Regeldatei `rules.txt` und den Konstanten `NumberOfObjects` bzw. `NumberOfComponents` im KARMA-Plugin abgestimmt werden müssen (Anmerkung: Auf eine Parametrisierung dieser Werte wurde der Einfachheit halber verzichtet). Nach der Eingabe der Werte wird über die Schaltfläche „Generate“ das verteilte System für die Simulation generiert, wobei ggf. zusätzliche

Listing 7.3: Regeln, die einen Versionsabstand von höchstens 4 zulassen

```
S001 ON update(D00) IF (true) THEN async_update(D00)
S002 ON async_update(D00) IF (diff_version >= 4) THEN sync_update(D00)

A001 ON write(D00) IF (diff_version < 4) THEN later(D00)
```

Listing 7.4: Regeln, die einen Versionsabstand von höchstens 10 zulassen

```
S001 ON update(E00) IF (true) THEN async_update(E00)
S002 ON async_update(E00) IF (diff_version >= 10) THEN sync_update(E00)

A001 ON write(E00) IF (diff_version < 10) THEN later(E00)

E00
0
1
2
3
4
−1
```

Kennzahlen abgefragt werden. Im Fall der Replikationsstrategie muss ein Zeitintervall eingegeben werden, das die Periode für die asynchrone Aktualisierung festlegt (siehe Abbildung 7.10). Anschließend kann der Simulationslauf im F4SR-Menü Simulation gestartet werden.

Wenn die Simulation mit der Replikationsstrategie RegRess durchgeführt wird, dann wird während eines Simulationslaufs im Konsolenfenster der aktuelle und der gemittelte Konsistenzgrad ausgegeben (siehe Abschnitt 7.3.3). Die fortlaufende Ausgabe ermöglicht die Beobachtung des Konsistenzgrades im eingeschwungenen Zustand der Simulation. In Abbildung 7.11 sind diese Werte dargestellt, die auf den Regeln basieren, die durch Listing 7.3 vorgegeben sind: Die Regel in Zeile 1 besagt, dass alle Replikate asynchron aktualisiert werden. Durch die Regel in Zeile 2, die eine höhere Priorität als die erste Regel hat, wird jedoch eine synchrone Aktualisierung gefordert, sofern der Versionsabstand größer als vier ist. Weil die synchrone Aktualisierung von Replikaten gemäß Protokoll (siehe Listing 4.1 auf Seite 71) nicht möglich ist, wenn noch Aufträge der Replikate in der Replica Queue anstehen, wird der Schreibzugriff abgewiesen, d.h. der Versionsabstand kann nicht größer als vier werden. Die Regel in Zeile 4 besagt, dass die asynchrone Aktualisierung erst bei einem Versionsabstand von vier aktiv wird.

Falls nun die Periode für die asynchrone Aktualisierung niedrig gesetzt wird, dann werden Replikate, die einen Versionsabstand von vier erreicht haben, zeitnah aktualisiert, d.h. die entsprechenden Aufträge aus der Replica Queue werden abgearbeitet, sodass der Versionsabstand dieser Replikate wieder null ist. Die Erwartungswerte für dieses Beispiel wurden in Gleichung 4.17 auf Seite 103 bzw. Gleichung 4.18 auf Seite 103 berechnet: $V_{Prob} = 1,488$ bzw. $KG_{Prob} = 0,511$.

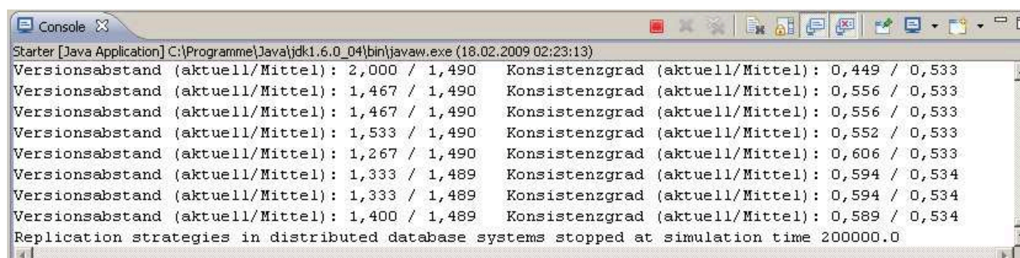

Abbildung 7.11.: Konsistenzgrad bzgl. der Regeln im Listing 7.3

```
Console
Starter [Java Application] C:\Programme\Java\jdk1.6.0_04\bin\javaw.exe (18.02.2009 02:37:46)
Versionsabstand (aktuell/Mittel): 1,600 / 1,484   Konsistenzgrad (aktuell/Mittel): 0,745 / 0,776
Versionsabstand (aktuell/Mittel): 1,600 / 1,485   Konsistenzgrad (aktuell/Mittel): 0,745 / 0,776
Versionsabstand (aktuell/Mittel): 1,600 / 1,485   Konsistenzgrad (aktuell/Mittel): 0,745 / 0,776
Versionsabstand (aktuell/Mittel): 1,667 / 1,485   Konsistenzgrad (aktuell/Mittel): 0,742 / 0,776
Versionsabstand (aktuell/Mittel): 1,667 / 1,485   Konsistenzgrad (aktuell/Mittel): 0,742 / 0,776
Versionsabstand (aktuell/Mittel): 1,667 / 1,485   Konsistenzgrad (aktuell/Mittel): 0,742 / 0,775
Versionsabstand (aktuell/Mittel): 1,667 / 1,485   Konsistenzgrad (aktuell/Mittel): 0,742 / 0,775
Versionsabstand (aktuell/Mittel): 1,667 / 1,486   Konsistenzgrad (aktuell/Mittel): 0,742 / 0,775
Replication strategies in distributed database systems stopped at simulation time 200000.0
```

Abbildung 7.12.: Konsistenzgrad bzgl. der Regeln im Listing 7.4

Die Abbildung 7.11 zeigt, dass diese Erwartungswerte durch die Simulation im Wesentlichen erreicht werden: Der mittlere Versionsabstand hat den Wert $V_{Sim} = 1,489$ und der mittlere Konsistenzgrad den Wert $KG_{Sim} = 0,534$.

An dieser Stelle soll nochmals auf den Versionsabstand (siehe Definition 41 auf Seite 93) und den Konsistenzgrad (siehe Definition 40 auf Seite 93) eingegangen werden, die während der Simulation angezeigt werden. Der Vorteil der Normierung beim Konsistenzgrad ist es, dass ein hoher Versionsabstand nicht so stark ins Gewicht fällt, der Nachteil, dass der Wert schwerer zu interpretieren ist. Wenn nun das Beispiel wie im Listing 7.4 abgeändert wird, wobei davon ausgegangen wird, dass 15 logische Objekte existieren, dann wird nur noch ein Drittel der Replikate asynchron aktualisiert, denen in diesem Fall ein Versionsabstand von zehn erlaubt ist. Damit berechnen sich die Erwartungswerte für dieses zweite Beispiel wie folgt: $V_{Prob} = 1,488$ bzw. $KG_{Prob} = 0,764$.

In der Simulation werden die Erwartungswerte wie folgt angezeigt (siehe Abbildung 7.12): Der mittlere Versionsabstand hat den Wert $V_{Sim} = 1,486$ und der mittlere Konsistenzgrad den Wert $KG_{Sim} = 0,775$. Zunächst kann gesagt werden, dass für beide Beispiele die Simulation die berechneten Konsistenzgrade mit kleinen Abweichungen ermittelt. Verschiedene Simulationsläufe zeigen, dass je höher die Simulationszeit ist, desto näher liegen die durch die Simulation ermittelten Werte bei den berechneten Erwartungswerten.

Weiterhin zeigen die beiden Beispiele den Unterschied des Versionsabstands und des Konsistenzgrades. In beiden Beispielen ist der Versionsabstand der replizierten Datenbank gleich, der Konsistenzgrad der replizierten Datenbank weist jedoch im zweiten Beispiel einen um ca. 50% höheren Wert auf. Dies liegt daran, dass beim Konsistenzgrad der erlaubte höhere Versionsabstand der Replikationseinheit $E00$ nicht so stark gewichtet ist und somit die konsistenten Replikate aufgewertet werden. Damit kann geschlossen werden, dass beide Berechnungsarten für die Analyse hilfreich sind.

Der Analysator des F4SR gibt die Ergebnisse eines Simulationslaufs in Form von Kreis- bzw. Tortendiagrammen, die das Zugriffsverhalten wiedergeben, einem Antwortzeit-Diagramm und Log-Informationen aus. In Abbildung 7.13 sind die Kreisdiagramme für das oben genannte, zweite Beispiel dargestellt, d.h. die Kreisdiagramme zeigen die Ergebnisse eines Simulationslauf mit der Replikationsstrategie RegRess unter Verwendung der Regeln in Listing 7.4. Das linke, obere Kreisdiagramm zeigt das Schreib-/Leseverhältnis, das im Wesentlichen den Vorgaben entspricht. Das rechte, obere Kreisdiagramm zeigt das Verhältnis von Lesezugriffen auf aktuelle Replikate zu veralteten Replikaten. Weil durch die Regeln die Replikate der Replikationseinheit $E00$ einen Versionsabstand von zehn haben dürfen und $E00$ ein Drittel der Replikate abdeckt, zeigt das Kreisdiagramm das Verhältnis von eins zu zwei für Lesezugriffe auf veraltete zu aktuellen Replikaten. Das linke, untere Kreisdiagramm zeigt das Verhältnis von erfolgreichen zu ungültigen Lesezugriffen. Weil durch Schreibzugriffe Replikate gesperrt sein können, treten auch ungültige Lesezugriffe auf. Das rechte, untere Kreisdiagramm zeigt das Verhältnis von erfolgreichen zu ungültigen Schreibzugriffen. Ungültige Schreibzugriffe treten dann auf, wenn eines der

betroffenen Replikate bei der synchronen Aktualisierung gesperrt ist oder eine der entsprechenden Komponenten mit Replikat nicht verfügbar ist.

Das Antwortzeit-Diagramm in Abbildung 7.14 basiert ebenfalls auf dem zweiten Beispiel. Die Antwortzeit von Lesezugriffen, die durch rote Punkte dargestellt sind, variiert in Abhängigkeit der zugegriffenen Komponente mit Replikat, hängt also von der Konfiguration des verteilten Systems ab. Die Antwortzeit von Schreibzugriffen (blaue Punkte) ist zweigeteilt: Bei Schreibzugriffen auf Replikate der Replikationseinheit $E00$ werden alle Aufträge in der Replica Queue gepuffert, wofür aus Clientsicht eine geringe Zeit benötigt wird. Die zeitversetzte Aktualisierung bzw. Bearbeitung der Aufträge der Replica Queue zählt nicht zur Antwortzeit. Bei Replikaten, die nicht zur Replikationseinheit $E00$ gehören, erfolgt ein synchronisierter Zugriff auf alle Komponenten. Daher hängt die Antwortzeit von der langsamsten Komponente ab. Neben den Antwortzeiten wird im Diagramm die Anzahl verfügbarer Komponenten (Online Nodes, grüne Kennlinie) und die Lastspitzen von Komponenten (Overload Nodes, braune Kennlinie) angezeigt.

Mit den Diagrammen des F4SR-Analysators sowie den berechneten Werten für den Versionsabstand und Konsistenzgrad ist eine Beurteilung der Replikationsstrategie RegRess bzw. den aktuell spezifizierten Regeln möglich. Nichtsdestotrotz sind weitere Auswertungen denkbar, die im Ausblick diskutiert werden (siehe Kapitel 9). Weil durch die Sprachmächtigkeit der RRML unterschiedlichstes Verhalten von RegRess konfiguriert werden kann, werden nachfolgend keine weiteren Diagramme oder Beispiele gezeigt. Abschließend in dieser Evaluation wird RegRess mit bekannten Replikationsstrategien, insbesondere den verwandten Arbeiten, verglichen. Dabei wird gezeigt, wie mit RegRess die jeweiligen Replikationsstrategien nachgebildet bzw. erweitert werden, in dem entsprechende Regeln angegeben werden. Dabei wird folgender Aufbau gewählt:

- Vergleich zu synchronen Replikationsstrategien (siehe Abschnitt 3.3.1)
- Vergleich zu asynchronen Replikationsstrategien (siehe Abschnitt 3.3.2)
- Vergleich zu adaptiven Replikationsstrategien (siehe Abschnitt 3.4.2)

Vergleich zu synchronen Replikationsstrategien

Die Replikationsstrategie ROWA (Read One Write All) ist im Grunde die Voreinstellung von RegRess, weil bei keiner Angabe von Regeln jedes Replikat eines logischen Objekts bei einem Schreibzugriff synchron aktualisiert wird. Damit kann ein lokales oder beliebiges Replikat gelesen werden, um stets einen Lesezugriff auf ein aktuelles Replikat zu erhalten. Durch RegRess ist z.B. eine Erweiterung möglich, bei der die logischen Objekte nach wichtigen Daten, die stets konsistent sein müssen, und weniger wichtigen Daten, die temporär inkonsistent sein dürfen,

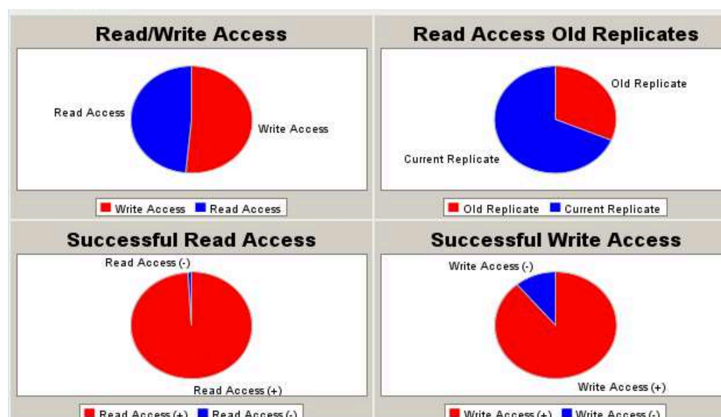

Abbildung 7.13.: Kreisdiagramme des F4SR

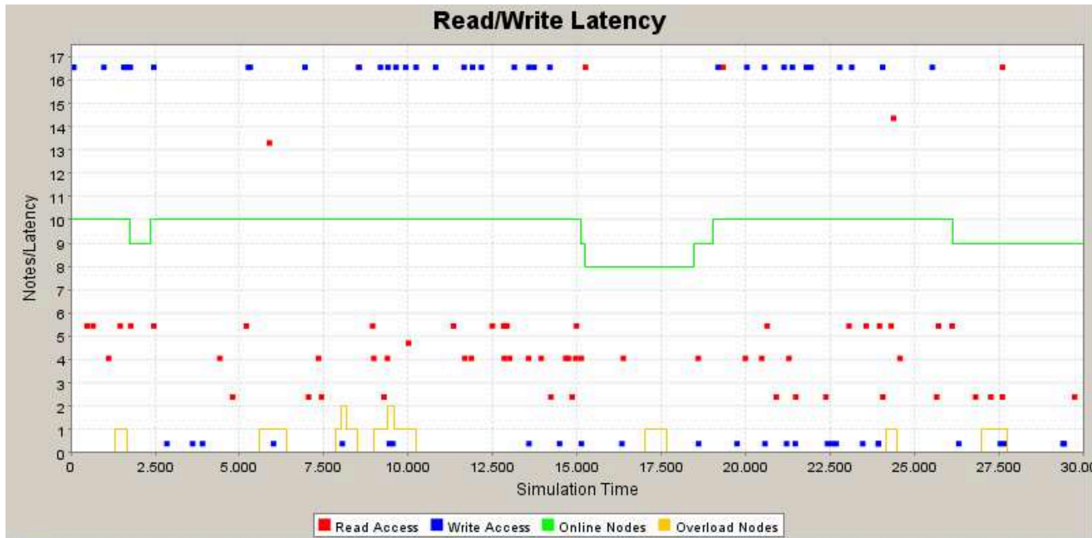

Abbildung 7.14.: Antwortzeit-Diagramm des F4SR

klassifiziert werden. Wenn die weniger wichtigen, logischen Objekte der Replikationseinheit E_1 zugeordnet werden, dann wird oben genanntes Verhalten durch folgende Regel erreicht:

(1) ON *update*(E_1) IF *true* THEN *async_update*(E_1)

Durch die Regel (1) werden Schreibzugriffe auf logische Objekte der Replikationseinheit E_1 grundsätzlich asynchron aktualisiert. Hierdurch werden im Allgemeinen weniger Transaktionen abgebrochen, als bei der synchronen Aktualisierung. Auch die Performance wird durch die Regel (1) erhöht, weil der Client nicht auf die Bearbeitung des synchronisierten Zugriffs warten muss. Die Replikationsstrategie ROWA-Available erweitert ROWA dadurch, dass nicht-verfügbare Komponenten zeitversetzt aktualisiert werden, d.h. nicht-verfügbare Komponenten bedingen keinen Abbruch der Transaktion. ROWA-Available wird durch folgende Regel nachgebildet.

(2) ON *update*(D) IF *true* THEN *changeableTrue*(D)

Die Regel (2) besagt, dass alle Replikate wechselfähig sind, sodass bei Nicht-Verfügbarkeit von Komponenten die entsprechenden Replikate asynchron aktualisiert werden. Weil im Allgemeinen Nicht-Verfügbarkeit nicht von Nicht-Erreichbarkeit durch Netzpartitionierung unterschieden werden kann, können sowohl bei ROWA-Available als auch bei RegRess Inkonsistenzen beim Lesen beliebiger Replikate auftreten. Bei RegRess können jedoch konsistente Lesezugriffe dadurch erzwungen werden, dass über den Replikationsmanager KARMA zugegriffen wird und geeignete Leseregeln definiert werden.

Weil bei RegRess nicht vorgesehen ist, Sperren auf Replikate zu setzen, deren Aktualisierungsauftrag noch in der Replica Queue ansteht, kann die Replikationsstrategie Primary Copy nur in der Variante nachgebildet werden, bei dem Lesezugriffe auch die Primärkopie bzw. den Replikationsmanager KARMA kontaktieren. Um konsistente Lesezugriffe durchzuführen, muss im Falle von RegRess der Lesezugriff über den KARMA erfolgen. Eine Zuordnung von logischen Objekten zu Primärkopien bzw. eine Verteilung der Primärkopien auf verschiedene Komponenten ist mit RegRess einfach durch die Definition entsprechender Replikationseinheiten und entsprechenden Regeln möglich. Sei angenommen, dass die logischen Objekte in drei Replikationseinheiten E_1, E_2 und E_3 partitioniert werden und die Komponente K_i die Primärkopie der Replikationseinheit E_i speichern soll. Dann wird eine derartige Verteilung durch folgende Regeln bewirkt:

(3a)	ON	*update(D)*	IF	*true*	THEN	*async_update(D)*
(3b)	ON	*async_update(E_1)*	IF	*true*	THEN	*sync_update(K_1)*
(3c)	ON	*async_update(E_2)*	IF	*true*	THEN	*sync_update(K_2)*
(3d)	ON	*async_update(E_3)*	IF	*true*	THEN	*sync_update(K_3)*

Die Regeln (3b), (3c) und (3d) müssen eine höhere Priorität als die Regel (3a) erhalten, damit die zuvor gesetzte, asynchrone Aktualisierungsart derart überschrieben wird, dass für die jeweilige Replikationseinheit die entsprechende Komponente synchron aktualisiert wird. Auch hier sind verschiedene Erweiterungen denkbar, z.B. kann anstatt einer Primärkopie eine Menge von Primärkopien angelegt werden, indem die entsprechende Regel kopiert wird und weitere Komponenten synchron aktualisiert werden.

Votierungsverfahren können mit RegRess in der Version 2.0 nicht nachgebildet werden. Zwar kann bei RegRess auch eine Menge von Replikaten synchron aktualisiert werden, sodass anschließende Lesezugriffe verfügbare, konsistente Replikate identifizieren. Bei Lesezugriffen über den KARMA bietet RegRess sogar höhere Flexibilität als feste Quorengrößen, benötigt aber höheren Kommunikationsaufwand, um konsistente Replikate zu bestimmen. Aber andererseits können bei RegRess keine Quorenmengen zusammengestellt werden, es ist keine Gewichtung und keine Strukturierung von Komponenten möglich.

Vergleich zu asynchronen Replikationsstrategien

Die unidirektionale, asynchrone Replikationsstrategie Master/Slave-Replikation gleicht der Replikationsstrategie Primary Copy, wobei dann ohne Sperren gearbeitet wird und somit inkonsistente Lesezugriffe möglich sind. Damit kann die Master/Slave-Replikation durch folgende Regeln nachgebildet werden, wenn angenommen wird, dass genau ein Master existiert und die Komponente K_0 dieser Master ist:

(4a)	ON	*update(D)*	IF	*true*	THEN	*async_update(D)*
(4b)	ON	*async_update(K_0)*	IF	*true*	THEN	*sync_update(K_0)*

Die Regel (4a) sorgt zunächst für die asynchrone Aktualisierung aller Replikate und die Regel (4b), die eine höhere Priorität als die Regel (4a) bekommt, bestimmt die Komponente K_0 dadurch zum Master, dass alle Replikate von K_0 synchron aktualisiert werden, d.h. in der Client-Transaktion aktualisiert werden. Wie bei der Master/Slave-Replikation können auch bei RegRess Lesezugriffe auf lokale oder beliebige Replikate inkonsistent, genauer gesagt, veraltet sein. Im Allgemeinen wird bei der Master/Slave-Replikation die zeitversetzte Aktualisierung unmittelbar nach Änderung des Masters angestoßen. Bei RegRess geschieht dies ebenfalls dadurch, dass keine Regel für die asynchrone Aktualisierung definiert wird.

Andere unidirektionale Replikationsstrategien begrenzen das Alter von Replikaten bzw. tolerieren ein vorgegebenes Alter der Replikate durch abgeschwächte Konsistenzbedingungen (siehe Abschnitt 3.2.3). Die Replikationsstrategie Quasi Copies verwendet z.B. Kohärenzbedingungen. RegRess bildet Quasi Copies durch die Funktionen `diff_version`, `diff_time` und `diff_value` in Bedingungen nach, wobei bei RegRess durch die feingranularen Definitionsmöglichkeiten der Regeln eine größere Flexibilität gegeben ist. Gleiches gilt für die Replikationsstrategie Snapshots, die durch die Funktion `period` nachgeahmt wird. Erweiterungen sind hier z.B. in RegRess möglich, indem die Abstände abhängig von der Tageszeit variieren:

(5a)	ON	*update(D)*	IF	*true*	THEN	*async_update(D)*
(5b)	ON	*write(D)*	IF	*diff_time* $<$ *300 and hour* $<$ *13*	THEN	*later(D)*
(5c)	ON	*write(D)*	IF	*diff_time* $<$ *900 and hour* $>$ *12*	THEN	*later(D)*

Durch die Regel (5a) wird festgelegt, dass alle Replikate asynchron aktualisiert werden. Während die Regel (5b) vormittags einen Zeitverzug von 5 Minuten gestattet, darf nachmittags der Zeit-

verzug auf Grund der Regel (5c) 15 Minuten betragen. Bei RegRess kann neben periodischen Zeitintervallen und Kohärenzbedingungen auch der Konsistenzgrad in Regeln verwendet werden und erlaubt damit, unterschiedliche abgeschwächte Konsistenzbedingungen für verschiedene Replikationseinheiten festzulegen.

Die Peer-To-Peer-Replikation als bidirektionale Replikationsstrategie gestattet Schreibzugriffe auf beliebige Replikate mit anschließender asynchroner Propagierung. Häufig wird der Schreibzugriff auf die lokale Komponente mit Replikat durchgeführt und anschließend wird die Änderung den anderen Komponenten mit Replikat mitgeteilt. Wenn angenommen wird, dass die initiierende Komponente die lokale Komponente mit Replikat ist, dann kann RegRess dieses Verhalten durch folgende Regeln nachstellen:

(6a)	ON	*update(D)*	IF	*true*	THEN	*async_update(D)*
(6b)	ON	*async_update(D)*	IF	$init_node = 0$	THEN	$sync_update(K_0)$
:						
(6c)	ON	*async_update(D)*	IF	$init_node = i$	THEN	$sync_update(K_i)$
:						
(6d)	ON	*async_update(D)*	IF	$init_node = n$	THEN	$sync_update(K_n)$

Wiederum wird durch die erste Regel (6a) die asynchrone Aktualisierung für alle Replikate gewählt. Die folgenden Regeln müssen eine höhere Priorität als (6a) besitzen, um berücksichtigt zu werden. Für jede initiierende Komponente muss eine Regel definiert werden, mit der die synchron zu aktualisierende Komponente bestimmt wird, d.h. in den Regeln (6c) und (6d) sind entsprechende Werte einzusetzen, weil die RRML in der Version 2.0 keine Variablen gestattet.

Andere bidirektionale Replikationsstrategien wie Independent Updates oder Data Patch haben Verfahren konzipiert, mit denen Schreib-/Schreibkonflikte$_R$ (siehe Definition 16 auf Seite 33) behandelt werden können, z.B. die Verwendung von Versionsvektoren oder die Berechnung von überlebenden Änderungen durch Regeln. Bei RegRess werden Schreib-/Schreibkonflikte$_R$ durch das Protokoll für synchrone Schreibzugriffe vermieden (siehe Listing 4.1 auf Seite 71), speziell durch die chronologische Verarbeitung der Zugriffe. Allerdings muss sich hierfür der KARMA in dem Fall, dass der KARMA selbst verteilt ist, synchronisieren. So ist z.B. eine Synchronisation der verteilten Replica Queue nötig. Wenn auf diesen Aufwand bei RegRess verzichtet werden soll, dann müsste eines der in Abschnitt 3.3.2 genannten Verfahren implementiert werden.

Des Weiteren wurden bidirektionale Replikationsstrategien wie CODA oder ROAM entwickelt, die primär bei der Verwendung von mobilen Systemen eingesetzt werden oder bei eingeschränkter Verbindung, z.B. auf Grund geringer Netzbandbreite. Bei diesen Replikationsstrategien werden die Komponenten mit Replikat häufig strukturiert bzw. klassifiziert, z.B. als Server und ressourcenschwacher Client. Weil derartige Unterscheidungen bei RegRess nicht vorgenommen werden, wird auf ein Vergleich verzichtet.

Vergleich zu adaptiven Replikationsstrategien

Die adaptiven synchronen Replikationsstrategien sind häufig Mischverfahren, wobei meistens Votierungsverfahren verwendet werden. Missing Updates wechselt z.B. von der Replikationsstrategie ROWA zum Votierungsverfahren Majority Consensus, falls bei einem Schreibzugriff ein Replikat nicht mehr erreicht wird. Weil der Fokus von RegRess auf die asynchrone Aktualisierung liegt und in der Version 2.0 keine Votierungsverfahren oder andere synchrone Replikationsstrategien mit Ausnahme von ROWA nachgestellt werden können, erübrigt sich ein Vergleich.

Die adaptive asynchrone Replikationsstrategie FAS zeichnet sich dadurch aus, dass Änderungen gemäß der Replikationsstrategie Primary Copy ohne Verwendung von Sperren propagiert werden. Für Lesezugriffe kann ein so genannter Frischeindex als Konsistenzbedingung angegeben werden, auf Basis dessen ein geeignetes Replikat für den Lesezugriff ausgewählt werden

kann. Diese Idee wurde bei RegRess durch die Leseregeln aufgenommen und um die Möglichkeit erweitert, auch technische Bedingungen beim Lesen zu berücksichtigen:

(7a) ON *read*(E_1) IF *true* THEN *getConsistency(V)*
(7b) ON *read*(E_2) IF *true* THEN *getPerformance(R)*

Die Regel (7a) erlaubt analog zur Replikationsstrategie FAS die Auswahl von Replikaten, die bestimmten Konsistenzbedingungen genügen, z.B. in diesem Fall ist durch den Parameter „V“ der Versionsabstand für die Replikationseinheit E_1 spezifiziert. Ein Client gibt bei einem Lesezugriff die maximale Grenze für den Versionsabstand als Parameter mit. Demgegenüber bezieht sich die Regel (7b) auf die Replikationseinheit E_2, für die durch den Parameter „R“ die Antwortzeit der Komponenten mit Replikat berücksichtigt wird. In RegRess können dieses Regeln auch kombiniert werden, d.h. es werden Replikate für den Lesezugriff bestimmt, die bestimmten Konsistenz- und Performancekriterien genügen.

Eine Pufferung von Änderungen wird auch bei der Replikationsstrategie Fracs vorgenommen. Wenn eine Grenze erreicht wird, dann werden keine weiteren Schreibzugriffe zugelassen, bis der Puffer geleert ist. Als Abstandsfunktion dient der Versionsabstand. RegRess puffert Änderungen in der Replica Queue, erlaubt aber gegenüber Fracs unterschiedlichste Funktionen, die die Bearbeitung der Aufträge der Replica Queue anstoßen. Weiterhin kann mittels Regeln festgelegt werden, ob alle Aufträge der Replica Queue bearbeitet werden sollen oder nur ein einzelner Auftrag je Replikat mit anschließender erneuter Prüfung der Bearbeitungsbedingung:

(8a) ON *update*(E_3) IF *true* THEN *async_update*(E_3)
(8b) ON *write*(E_3) IF $\mathit{diff_time} < 300$ THEN *later*(E_3)
(8b) ON *write*(E_3) IF $\mathit{diff_time} \geq 300$ THEN *one*(E_3)

Für Replikate der Replikationseinheit E_3 wird im Beispiel durch die Regel (8a) die asynchrone Aktualisierung festgelegt. Bei der asynchronen Aktualisierung wird durch die Regel (8b) bestimmt, dass Aufträge, die noch keine fünf Minuten alt sind, nicht bearbeitet werden. Die Regel (8c), die höher priorisiert werden muss als Regel (8b), sorgt dafür, dass genau ein Auftrag des Replikats bearbeitet wird. Folgt ein weiterer Auftrag für dieses Replikat, dann wird erneut der Zeitverzug geprüft.

Die Replikationsstrategie Tact nutzt einen dreidimensionalen Vektor für die Kohärenzbedingungen, d.h. für den Versionsabstand, für den Zeitverzug und für die Wertdifferenz. Bei RegRess können die unterschiedlichen Konsistenzeigenschaften in der Bedingung einer Regel durch logische Verknüpfung entsprechender arithmetischer Ausdrücke kombiniert werden. Wenn z.B. die Regel (8b) durch die Regel (9) ersetzt wird, dann wird ein Auftrag der Replica Queue nicht bearbeitet, solange der Zeitverzug kleiner als 5 Minuten oder der Versionsabstand kleiner als drei ist:

(9) ON *write*(E_3) IF $\mathit{diff_time} < 300 \text{ or } \mathit{diff_version} < 3$ THEN *later*(E_3)

Auch bei der Replikationsstrategie Aspect werden im Wesentlichen Kohärenzbedingungen für die Ermittlung von Abständen benutzt. RegRess erweitert diesen Ansatz insbesondere um die technischen Konsistenzbedingungen, die z.B. durch die Funktionen `conflicts`, `datasize` oder `responsetime` in den Bedingungen von Regeln einfließen können:

(10) ON *update*(E_3) IF $\mathit{conflicts} < 100$ THEN *async_update*(E_3)
(11) ON *write*(E_3) IF $\mathit{responsetime} > 1000$ THEN *later*(E_3)

Durch die Regel (10) wird z.B. festgelegt, dass eine asynchrone Aktualisierung der Replikate der Replikationseinheit E_3 nur dann zulässig ist, solange die Anzahl an Konflikten kleiner als

100 ist. Eine Aktualisierung der asynchron zu aktualisierenden Replikate erfolgt aber nur dann, wenn, wie durch Regel (11) bestimmt, die Antwortzeit der Zielkomponente kleiner oder gleich 1000 Millisekunden ist.

Durch die Sprachmächtigkeit der RRML, Version 2.0, können also mit der in dieser Dissertation entwickelten Replikationsstrategie RegRess die oben genannten Replikationsstrategien nachgebildet werden. Weiterhin können Mischverfahren durch geeignete Regeln definiert werden bzw. Erweiterungen bestehender Replikationsstrategien vorgenommen werden. Die feingranulare Definition von Regeln erlaubt unterschiedliches Verhalten für Replikationseinheiten, logische Objekte oder Komponenten mit Replikat. Weil in den Regeln neben Datum- und Zeitfunktionen auch Systemkennzahlen berücksichtigt werden können, kann eine Adaption zur Laufzeit erreicht werden. Votierungsverfahren oder andere strukturierte Replikationsstrategien werden derzeit von RegRess nicht unterstützt.

Teil IV.

Zusammenfassung, Fazit und Ausblick

8. Zusammenfassung und Fazit

In heterogenen, autonomen Informationssystemen stellt die Replikation von Daten hohe Ansprüche an eine geeignete Replikationsstrategie. So ist eine Kopplung der Informationssysteme nötig und für die Schreib- und Lesezugriffe auf die Replikate wird ein Transaktionskonzept benötigt, das zumindest verteilte Transaktionen verarbeiten kann. Vor allem aber muss hinsichtlich der Replikationsziele Verfügbarkeit, Performance und Konsistenz durch die Replikationsziele ein geeigneter Trade-Off erreicht werden. Dieses Abwägen hinsichtlich der Replikationsziele wird dadurch erschwert, dass die beteiligten Informationssysteme ihre Systemzustände ändern und dass auf diese Veränderungen reagiert werden muss. Daher wurden in der jüngeren Vergangenheit im Allgemeinen adaptive Replikationsstrategien entwickelt, die sich zur Laufzeit den veränderten Systemzuständen anpassen.

Der Hauptbeitrag dieser Dissertation ist die regelbasierte Replikationsstrategie RegRess sowie die Regelsprache RRML, die die Formulierung von Replikationsregeln für RegRess ermöglicht. Bei RegRess erfolgt die Koordination für Schreib- und Lesezugriffe auf Basis dieser Regeln, d.h. bei jedem Zugriff wird eine Inferenz der Regeln durchgeführt, womit die von dem Zugriff betroffenen Replikate ermittelt werden. Durch diese Vorgehensweise wird unterschiedlichstes Konsistenzverhalten von RegRess realisiert, insbesondere werden temporäre Inkonsistenzen toleriert, sodass RegRess zu den asynchronen Replikationsstrategien zu zählen ist. Eine Regelmenge mit für den Anwendungsfall spezifizierten Regeln bildet die Konfiguration von RegRess. Weil in den Regeln Systemzustände berücksichtigt werden können, kann zur Laufzeit das Verhalten angepasst werden, d.h. bei RegRess handelt es sich um eine konfigurierbare, adaptive, asynchrone Replikationsstrategie.

Für die Präsentation der hier entwickelten Konzepte wurden in den Grundlagen zunächst allgemeine Begriffe verteilter Systeme eingeführt. Speziell für die Datenreplikation wurden Begriffe definiert, um eine einheitliche Terminologie für diese Dissertation festzulegen. Einen Schwerpunkt dabei bildeten die Definitionen für die abgeschwächte Konsistenz in replizierten Datenbanken, genauer gesagt, wurden verschiedene Konsistenzmaße zusammengetragen, die auf Forschungsarbeiten in replizierten Datenbanken beruhen. Diese Konsistenzmaße konnten damit bei der Spezifikation des Konsistenzverhaltens von RegRess berücksichtigt werden.

Bei RegRess werden Schreibzugriffe auf ein logisches Objekt derart koordiniert, dass die Replikate des logischen Objekts in synchron und asynchron zu aktualisierende Replikate partitioniert werden. Die Partitionierung wird durch Inferenz der spezifizierten Regeln ermittelt. Bei der synchronen Aktualisierung werden innerhalb einer synchronisierten (Teil-)Transaktion die synchron zu aktualisierenden Replikate geschrieben sowie Aufträge für die asynchron zu aktualisierenden Replikate in die Replica Queue geschrieben. Die Replica Queue, die eine persistente Pufferung realisiert, muss demnach eine transaktionale Verarbeitung von verteilten Transaktionen unterstützen. Die asynchron zu aktualisierenden Replikate werden zeitversetzt in der so genannten asynchronen Aktualisierung geschrieben. Die Veralterung der Replikate wird dabei wiederum durch Inferenz der Regeln gesteuert, d.h. mittels Auswertung der Regeln wird bestimmt, ob derzeit ein asynchron zu aktualisierendes Replikat geschrieben werden darf oder nicht. Auch die Menge der Replikate, aus der ein Replikat für einen Lesezugriff ausgewählt wird, wird per Inferenz der Regeln berechnet, wobei spezielle Eigenschaften der zu lesenden Replikate festgelegt werden können.

Die Protokolle von RegRess für die synchrone und die asynchrone Aktualisierung sowie für Lesezugriffe wurden spezifiziert, wodurch die regelbasierte Koordination festgeschrieben wurde. Ein wichtiger Aspekt bei den Protokollen für die Schreibzugriffe ist die Einhaltung der chro-

nologischen Verarbeitung, wodurch dass Korrektheitskriterium „letztendliche Konsistenz“ von RegRess gewährleistet wird. Daneben wurde in den Protokollen das transaktionale Verhalten der Zugriffe spezifiziert. So bedeutet z.B. ein Transaktionsfehler beim synchronen Schreibzugriff nicht zwingend ein Zurückrollen der gesamten Transaktion, sondern ein Zurückrollen der aktuellen Teiltransaktion mit folgendem Neuaufsetzen, wobei das fehlerverursachende Replikat dann asynchron aktualisiert wird, wenn es gemäß Regeln wechselfähig ist.

Um eine möglichst große Flexibilität bei der Koordination von Zugriffen zu erreichen, wurde eine Anforderungsanalyse hinsichtlich der Replikationsziele Verfügbarkeit, Performance und Konsistenz durchgeführt. Dabei wurden zunächst Anforderungen an die Inferenz als solches spezifiziert, die z.B. den Start- und Endzustand einer Inferenz bzw. die Reaktion im Fehlerfall festlegen, wobei dann der Schwerpunkt auf Konsistenz liegt. Die Anforderungen für die fachlichen Konsistenzbedingungen ergeben sich im Wesentlichen durch die in den Grundlagen zusammengetragenen abgeschwächten Konsistenzbegriffe. Als Beitrag von RegRess sind speziell die Anforderungen für die technischen Konsistenzbedingungen zu sehen, bei denen z.B. die Performance in Form von Antwortzeiten berücksichtigt sind.

Ein wichtiges Beurteilungsmerkmal einer Replikationsstrategie ist das zugrunde liegende Korrektheitskriterium, nach dem die Koordination der Zugriffe erfolgt. RegRess erreicht durch die chronologische Verarbeitung das aus der Literatur bekannte Korrektheitskriterium „letztendliche Konsistenz“. Um aber genauere Angaben zur Konsistenz der replizierten Datenbank vorzunehmen, wurde das Konsistenzmaß „Konsistenzgrad“ definiert bzw. gezeigt, wie ein derartiger Konsistenzgrad durch Messung am realen System, durch Simulation oder analytisch bestimmt werden kann. Des Weiteren wurde eine Vorgehensweise aufgezeigt, wie ein Trade-Off hinsichtlich der Replikationsziele Verfügbarkeit, Performance und Konsistenz ermittelt wird, wobei dann der Konsistenzgrad verwendet werden kann.

Der Schwerpunkt bei der Entwicklung der Regelsprache RRML, mit der Replikationsregeln für RegRess formuliert werden, bildet der konzeptionelle Entwurf der Sprache, bei dem die Syntax und die Semantik festgelegt wurden. Die Regeln werden dabei in der ON-IF-THEN-Darstellung präsentiert, die auch bei der Festlegung der Syntax in der Backus-Naur-Form verwendet wird. Weil Replikationsregeln in Widerspruch stehen können, d.h. die Aktionen von auszuführenden Regeln lösen unterschiedliche Zustandsänderungen aus, werden widersprüchliche Regeln durch Widerspruchsregeln behandelt. Dabei wird im Wesentlichen eine überlebende Regel bestimmt.

Die semantische Bedeutung der Ereignisse und Aktionen von Regeln für die synchrone und asynchrone Aktualisierung, von Regeln für Lesezugriffe und von Widerspruchsregeln wurde detailliert spezifiziert, wobei die Ereignisse und die Aktionen parametrisiert sind. Mögliche Parameter sind z.B. Replikationseinheiten, logische Objekte oder die initiierenden Komponenten. Durch die Aktionen wird im Fall der synchronen Aktualisierung die Partitionierung bestimmt, im Fall der asynchronen Aktualisierung die Erlaubnis für einen Schreibzugriff, im Fall eines Lesezugriffs die Menge der passenden Replikate und im Fall von Widerspruchsregeln die Widerspruchsbehandlung. Der Ablauf der jeweiligen Aktionsart, der bei der Inferenz auszuführen ist, wurde als Protokoll spezifiziert. Es sei nochmals angemerkt, dass die Aktionen keine Zugriffe auf die Replikate durchführen, sondern für einen Replikationsmanager die Art der Zugriffe ermitteln.

Der Bedingungsteil ist für alle Regeln gleich. Eine Bedingung ist eine vereinfachte, prädikatenlogische Formel. Dabei wird auf Quantoren verzichtet und Prädikate können durch logische Operatoren zu Formeln verknüpft werden. Prädikate selbst sind zweistellig, wobei eine Funktion mit einer Konstanten verglichen wird. Es wurden Funktionen für Datum und Uhrzeit, für die Identifikation der beteiligten Komponenten und Objekte sowie für fachliche und technische Konsistenzbedingungen definiert. Mittels dieser Funktionen in den Bedingungen wird das flexible Verhalten von RegRess erreicht, weil sowohl Konsistenzmaße als auch technische Kennzahlen zeitabhängig berücksichtigt werden können.

Damit RegRess eingesetzt werden kann, wurde der Replikationsmanager KARMA entworfen. Der KARMA realisiert RegRess inklusive eines Regelinterpreters für die Regeln der RRML.

Daher wurde eine geeignete, komponentenbasierte Softwarearchitektur erstellt und eine Spezifikation der Komponenten vorgenommen. Weil es sich bei den Zugriffen auf die Replikate um verteilte, geschachtelte Transaktionen handelt, wurde das benötigte Transaktionsverhalten spezifiziert. Idealerweise nutzt der KARMA den Transaktionsmanager RD-TM, der in einer studentischen Arbeit entwickelt wurde und so genannte High-Level-Services für eine transaktionale Verarbeitung bietet.

Neben dem RD-TM wurden weitere experimentelle Prototypen durch studentische Arbeiten entwickelt, die spezielle Funktionalität wie z.B. die technische Anbindung von Informationssystemen an den KARMA auf die technische Machbarkeit untersuchten. Diese Prototypen werden in dieser Dissertation kurz vorgestellt, wobei detaillierter auf das Simulationsframework F4SR eingegangen wird, der die Basis für die Simulation und Evaluation bildet. Der in dieser Dissertation vorgestellte Prototyp KARMA, der selbst implementiert wurde, dient in erster Linie zum Test der spezifizierten Protokolle für Schreib- und Lesezugriffe sowie zur Prüfung des Regelinterpreters, indem entsprechende Log-Informationen ausgewertet werden.

Um RegRess simulieren zu können, wurde der Prototyp KARMA als KARMA-Plugin für den F4SR angepasst. Dazu mussten im Wesentlichen die entsprechenden Schnittstellen, die der F4SR für das Einbinden von Replikationsstrategien bereitstellt, implementiert werden. Der F4SR mit dem KARMA-Plugin erlaubt den Vergleich zu anderen Replikationsstrategien und insbesondere den Vergleich verschiedener Regelmengen in einem Simulationsmodell, das ein verteiltes System mit Lastspitzen und Ausfällen simuliert. Der Analysator des F4SR zeigt die Ergebnisse eines Simulationslaufs in verschiedenen Diagrammen, z.B. Kreisdiagramme für das Verhältnis erfolgreicher zu abgewiesener Zugriffe oder ein Verlaufsdiagramm, das u.a. Antwortzeiten der Zugriffe darstellt. Zusätzlich wird für RegRess der gemittelte Versionsabstand und Konsistenzgrad der replizierten Datenbank aufgelistet.

Bei der Evaluation wurden exemplarisch zwei Regelmengen untersucht, für die der gemittelte Versionsabstand und Konsistenzgrad zuvor analytisch berechnet wurde. Die Simulationsläufe haben gezeigt, dass die vorhergesagten Werte im Wesentlichen erreicht werden. Dabei wurden auch die unterschiedlichen Interpretationsmöglichkeiten der beiden Kennzahlen diskutiert. Des Weiteren wurden in diesem Zusammenhang Regelmengen gezeigt, die das Nachstellen und Erweitern von Replikationsstrategien erlauben. Auf die Replikationsstrategien, die zu den verwandten Arbeiten zählen, wurde hierbei besonders eingegangen.

Durch diese Dissertation wurde die Machbarkeit der regelbasierten Replikationsstrategie RegRess demonstriert. Die Sprachmächtigkeit der RRML erlaubt unterschiedlichste Konfigurationen durch entsprechende Regelmengen und bedingt damit eine hohe Flexibilität von RegRess. Dadurch kann ein geeigneter Trade-Off hinsichtlich der Replikationsziele Verfügbarkeit, Performance und Konsistenz erreicht werden. Die Verwendung von Regeln bietet den Vorteil, dass die Anwendungslogik von der Replikationslogik, d.h. von der Vorschrift der Koordination für Schreib- und Lesezugriffe, getrennt ist. Bei den Simulationsläufen wurden im Allgemeinen nur einige Replikationsregeln benötigt, um das gewünschte Verhalten zu erreichen. Allerdings muss angemerkt werden, dass bei großen Datenbanken und/oder vielen Informationssystemen der Aufwand für die Erstellung der Regeln durchaus groß sein kann, bietet dann aber den Vorteil einer feingranularen Steuerung der Koordination.

Ein weiterer Vorteil liegt bei RegRess in der hohen Flexibilität, der durch die Inferenz von Regeln bei jedem Zugriff bedingt ist und damit bei jedem Zugriff eine unterschiedliche Koordination bewirken kann. Insbesondere kann sich RegRess damit dem Systemzustand anpassen. Dieser Vorteil bedingt aber den Nachteil, dass vor jedem Zugriff Rechenaufwand für die Ermittlung der betroffenen Replikate nötig ist. Je nach Größe der verwendeten Regelmenge sollte der Rechner, auf dem der KARMA zum Einsatz kommt, entsprechend performant sein.

Ein weiterer Nachteil besteht darin, dass bei einem verteilten KARMA die benötigten Metainformationen wie z.B. Regeln und Replica Queue repliziert werden müssen, wobei hier Konsistenz gefordert ist. Andernfalls könnte die chronologische Verarbeitung von RegRess gefährdet sein

bzw. es könnten Schreib-/Schreibkonflikte auftreten. Dieser Nachteil entsteht nicht bei einem zentralen KARMA. Dann allerdings ist eine Abhängigkeit von dieser zentralen Stelle gegeben. Daher ist die Entscheidung, ob der KARMA zentral oder verteilt ist und wo er lokalisiert wird, von großer Bedeutung. Wenn hierfür eine geeignete Lösung gefunden ist, dann bietet der KARMA bzw. RegRess eine große Flexibilität und genügt hohen Ansprüchen.

9. Ausblick

In diesem Kapitel werden mögliche Erweiterungen und Folgearbeiten dieser Dissertation diskutiert. Dabei werden nicht die Punkte angesprochen, die als idealistische Voraussetzung durch die Annahmen getroffen wurden. Die Erweiterungen, die bei Verzicht auf die Annahmen erforderlich sind, wurden im Abschnitt 4.1.1 erläutert. Hier soll vielmehr auf funktionale und technische Erweiterungen eingegangen werden, wobei folgende Aspekte betrachtet werden:

1. Erweiterung des Simulationsframeworks F4SR.
2. Einsatz von JBoss Rules anstelle des eigenentwickelten Regelinterpreters.
3. Keine chronologische Verarbeitung, sondern Zulassen von Schreib-/Schreibkonflikten$_R$.
4. Verbesserung der Einhaltung von Kohärenzbedingungen.
5. Behandlung der Widerspruchsregeln als eigenständige Regeln.
6. Optimierung der Bearbeitung der Replica Queue.
7. Sortierung der Menge der passenden Replikate bei Lesezugriffen.
8. Berücksichtigung von Sessiongarantien.
9. Abbildung von Votierungsverfahren durch Regeln.

zu 1.) Eine mögliche Erweiterung des Simulationsframeworks F4SR (siehe Abschnitt 7.3.1) betrifft sowohl das Simulationsmodell als auch den Analysator. Im Simulationsmodell könnte vor allem die Simulation des verteilten Systems, genauer gesagt die möglichen Eigenschaften der Knoten (in dieser Arbeit als Komponenten mit Replikat bezeichnet) verbessert werden. Zwar können mittlere Antwortzeiten, Lastspitzen und Nicht-Verfügbarkeit vorgegeben werden, wobei die beiden letztgenannten Eigenschaften auf Basis einer Zufallszahl variieren, aber die Konfiguration ist eher umständlich. Außerdem wären hier mehr Variationsmöglichkeiten wünschenswert, wie z.B. eine unterschiedliche Datengröße der Replikate, variablere Leistungsprofile der Knoten etc., um eine bessere Affinität zu realen Systemen erzielen zu können.

Weiterhin könnten die Auswertungen des F4SR-Analysators verbessert werden. Dabei könnten einerseits die bestehenden Diagramme erweitert werden und andererseits kumulative Auswertungen über mehrere Simulationsergebnisse ermöglicht werden. So könnte das Kreisdiagramm für das Verhältnis von erfolgreichen zu abgewiesenen Schreibzugriffen dadurch verfeinert werden, dass nach den Gründen für den Fehler aufgeschlüsselt wird, wie z.B. Fehler wegen Schreibfehler, Fehler wegen Nicht-Verfügbarkeit des Knotens, etc. Zusätzlich könnten weitere Kennzahlen visualisiert werden, wie z.B. Konsistenzmaße für abgeschwächte Konsistenz (siehe Abschnitt 3.2.3)

Auch beim Antwortzeit-Diagramm (siehe Abbildung 7.14 auf Seite 205) wären detaillierte Angaben wünschenswert. Hier sind zusätzliche Auswertungen für mittlere Antwortzeiten je Knoten bzw. für Verfügbarkeiten der Knoten für die Analyse hilfreich. Dabei könnten z.B. die in Abschnitt 4.4 vorgestellten Metriken und Berechnungsvorschriften verwendet werden, um detailliertere Kennzahlen für einen Trade-Off der Replikationsziele Verfügbarkeit, Performance und Konsistenz zu erhalten.

Neben den Auswertungen je Simulationslauf würde eine aggregierte Darstellung mehrerer Simulationsergebnisse die Analyse vereinfachen. Dabei könnten statistische Verfahren verwendet werden, um Abweichungen kenntlich zu machen oder Ausreißer zu eliminieren. Auch ein Vergleich von Simulationsergebnissen, die z.B. bei RegRess auf Grund unterschiedlicher Regelmengen entstanden sind, würde den Analysten unterstützen.

Speziell für das KARMA-Plugin sollte der Transaktionsmanager des F4SR durch einen Transaktionsmanager wie den RD-TM (siehe Abschnitt 6.3.2) ersetzt werden. Damit würde die Simulation verteilter, verschachtelter Transaktionen ermöglicht werden. Dabei sollten dann auch

entsprechende Bearbeitungszeiten für die Commit-Protokolle der jeweiligen Transaktionsdienste in der Simulation berücksichtigt werden. Außerdem sollte ebenfalls die transaktionale Anbindung benötigter Ressourcen wie die Replica Queue ermöglicht werden.

zu 2.) JBoss Rules (vormals Drools [Rul08]) ist eine Schlüsselkomponente der JBoss Enterprise SOA Platform. Im Prototypen RD-TM (siehe Abschnitt 6.3.2), einem Transaktionsmanager für so genannte High-Level-Services, unterstützt JBoss Rules bei der Auswahl eines geeigneten Transaktionsdienstes. JBoss Rules bietet eine eigene Regelsprache mit zugehörigem Regelinterpreter. Es könnte untersucht werden, ob die RRML auf diese Regelsprache abgebildet werden kann, um dann den JBoss Regelinterpreter zu verwenden. Hierdurch würde die eigene Implementierung des Prototypen RIM (siehe Abschnitt 7.2) durch eine professionelle Komponente ersetzt.

zu 3.) Bei RegRess werden Schreib-/Schreibkonflikte$_R$ (siehe Definition 16 auf Seite 33) dadurch vermieden, dass eine chronologische Reihenfolge der Schreibzugriffe auf ein Replikat gewährleistet wird. Die Chronologie wird durch das Verwalten von Sperren bei Schreibzugriffen auf logische Objekte und durch Einhaltung der Bearbeitungsreihenfolge von Replikaten in der Replica Queue erreicht (siehe Abschnitt 4.2.2). Wenn der Replikationsmanager KARMA verteilt ist, dann müssen die Sperren global verwaltet werden und die Replica Queue muss konsistent repliziert werden, damit auch im verteilten Fall die Chronologie eingehalten wird. Durch diesen erhöhten Aufwand wird keine Behandlung von Schreib-/Schreibkonflikten$_R$ benötigt und RegRess erfüllt trivialerweise das Korrektheitskriterium letztendliche Konsistenz.

Wenn Schreib-/Schreibkonflikte$_R$ zugelassen werden, dann kann auf die Sperren für logische Objekte und auf das Replizieren der Replica Queue verzichtet werden, d.h. insbesondere, dass jeder lokale `ReplicaQueueManager` (siehe Abschnitt 6.2) eine eigene, unabhängige Replica Queue verwalten würde. Angemerkt sei, dass damit die Berechnung von Versionsabständen nicht mehr allein über eine Replica Queue erfolgen kann. Der Vorteil dieser Vorgehensweise im verteilten Fall ist es, dass die lokalen `ReplicaQueueManager` autonom sind und die Replikation damit nicht mehr von einer zentralen Stelle bzw. einer zentralen Synchronisation abhängt. Der Nachteil liegt darin, dass die Schreib-/Schreibkonflikte$_R$ behandelt werden müssen. Hierfür könnten z.B. Versionsvektoren für jedes Replikat verwendet werden (siehe Replikationsstrategie Independent Updates in Abschnitt 3.3.2).

zu 4.) Wenn Kohärenzbedingungen (siehe Definition 22 auf Seite 37) als Regeln für die asynchrone Aktualisierung definiert werden, dann wird bei RegRess bis zur spezifizierten Grenze keine Aktualisierung nachgeholt. Nachdem die Überschreitung der Grenze festgestellt ist, werden die entsprechenden Schreibzugriffe, die in der Replica Queue anstehen, durchgeführt. Zwischenzeitlich können jedoch weitere Schreibzugriffe auf das logische Objekt initiiert worden sein, sodass die vorgegebene Grenze theoretisch bei Weitem nicht eingehalten wird. Im Abschnitt 5.3.6 wurde bei der Vorstellung der Funktionen für fachliche Konsistenzbedingungen ein Beispiel gezeigt, wie durch eine entsprechende Regel für die synchrone Aktualisierung eine Grenze exakt eingehalten werden kann. Allerdings werden dann Schreibzugriffe auf das logische Objekt abgewiesen.

Bei der Replikationsstrategie Quasi Copies (siehe Abschnitt 3.3.2) werden Replikate, die eine Kohärenzbedingung verletzen, gesperrt. Der Vorteil liegt darin, dass keine Schreibzugriffe auf logische Objekte abgewiesen werden. Ein Nachteil ist bei dieser Vorgehensweise, dass die lokale Komponente mit Replikat mit einer zentralen Stelle bzw. mit einem aktuellen Replikat kommunizieren muss, um den Abstand gemäß der Kohärenzbedingung zu bestimmen, ggf. werden so genannte Alive-Nachrichten benötigt. Diese Funktionsweise könnte auch bei RegRess verwendet werden. Zusätzlich wäre denkbar, dass bei Eintreffen eines Schreibzugriffs, der die Kohärenzbedingung verletzt, zuvor entsprechende Aufträge aus der Replica Queue abzuarbeiten.

zu 5.) Widerspruchsregeln der RRML, Version 2.0 (siehe Abschnitt 5.3.5), werden bei der Behandlung von widersprüchlichen Regeln verwendet, die entweder für die synchrone oder für die asynchrone Aktualisierung oder für Lesezugriffe spezifiziert wurden. Bei der Ausführung einer Aktion der aktuellen Regel wird zu Beginn geprüft, ob ein Widerspruch vorliegt und ob die

aktuelle Aktion ausgeführt werden darf, wobei im letzten Fall eine Inferenz der Widerspruchsregeln durchgeführt wird. Die Inferenz liefert ein Verfahren, mit dem die Rangfolge von Regeln festgelegt wird, wodurch letztendlich entschieden wird, ob die aktuelle Aktion ausgeführt werden darf oder abgebrochen wird.

Es wäre auch denkbar, Widerspruchsregeln „gleichberechtigt" zu den anderen Regeln zu behandeln. Beim Erkennen widersprüchlicher Regeln muss dann einfach ein entsprechendes Ereignis ausgelöst werden, im vorliegenden Fall z.B. `conflict(<Parameter>)`. Grundsätzlich würde diese Vorgehensweise den Vorteil bieten, dass eine umfangreichere Widerspruchsbehandlung möglich wäre. In diesem Fall könnte nicht nur eine Rangfolge der Regeln festgelegt werden, mittels derer die „überlebende" Aktion bestimmt wird, sondern es könnte in den Aktionen von Widerspruchsregeln unterschiedlichstes Verhalten spezifiziert werden. Allerdings müsste dann sichergestellt werden, dass ein Konflikt tatsächlich behandelt wird.

zu 6.) Die Bearbeitung der Aufträge aus der Replica Queue könnte dahingehend optimiert werden, dass bei mehreren Aufträgen, die das gleiche Replikat betreffen, nur ein Schreibzugriff durchgeführt wird, der eine resultierende Änderung vornimmt. Wenn das Replikat jeweils absolut geändert wird, dann ist die resultierende Änderung der zeitlich letzte Auftrag des Replikats in der Replica Queue, d.h. der letzte Auftrag ist die „überlebende" Änderung. Wenn relative Änderungen möglich sind, z.B. wenn das Replikat inkrementiert wird, dann muss eine entsprechende relative Änderung berechnet werden, z.B. bei Vorliegen von n Aufträgen, die ein Replikat inkrementieren, ist eine Addition um den Wert n erforderlich.

Voraussetzung für eine derartige Optimierung ist es, dass keine Seiteneffekte gegenüber einer Einzelverarbeitung unberücksichtigt bleiben und dass die Inferenz für die asynchrone Aktualisierung als Ergebnis `AlleAuftraege` liefert, d.h. ohne erneute Inferenz dürfen alle Aufträge des Replikats bearbeitet werden. Wenn dann eine resultierende Änderung möglich ist, könnte in einer Transaktion ein entsprechender Schreibzugriff auf das Replikat und ein Löschen aller Aufträge des Replikats aus der Replica Queue erfolgen.

zu 7.) Im Prototyp KARMA (siehe Abschnitt 7.3.2) wurden Lesezugriffe auf ein logisches Objekt derart implementiert, dass gemäß Protokoll eine Liste passender Replikate bestimmt wurde. Die Liste wird dabei nicht nach den vorgegebenen Eigenschaften sortiert, sondern es wird vom Regelinterpreter eine unsortierte Liste geliefert. Der KARMA liest das erste Replikat der Liste, ggf. wird bei einem Fehler das nächste Replikat der Liste gelesen, und liefert dem Client den Wert des Replikats. Weil die aktuellen Replikate z.B. Konsistenzeigenschaften immer erfüllen, ist es damit möglich, dass ein Lesezugriff immer ein aktuelles Replikat liefert, obwohl laut Eigenschaften z.B. ein Versionsabstand von n zulässig ist.

Eine Implementierungsvariante wäre diejenige, bei der die Liste der passenden Replikate nach den vom Client übergebenen Eigenschaften sortiert wird und dann dasjenige Replikat gelesen wird, dass gerade die gegebenen Eigenschaften erfüllt. Eine weitere Variante wäre es, dass dem Client nicht der Wert geliefert wird, sondern die Liste mit den Referenzen der passenden Replikate sowie deren ermittelten Eigenschaften. Dann kann der Client selbst das zu lesende Replikat bestimmen. In diesem Fall wäre eine Anpassung des Protokolls für Lesezugriffe erforderlich (siehe Listing 4.3 auf Seite 77) und der Client müsste selbst das Einbinden des Zugriffs in die Transaktionskontrolle übernehmen.

zu 8.) In Definition 21 auf Seite 36 wurden Sessiongarantien vorgestellt, die einer Clientsession z.B. garantieren, dass in der Session durchgeführte Schreibzugriffe bei einem Lesezugriff enthalten sind. Bei RegRess sind Sessiongarantien nur dann erfüllt, wenn grundsätzlich aktuelle Replikate gelesen werden, weil diese alle Schreibzugriffe enthalten. Es wäre denkbar, die in der Definition 21 auf Seite 36 aufgestellten Sessiongarantien in Regeln abzubilden. Dann wäre es zumindest erforderlich, Session-Identifikatoren und die Änderungen einer Session zu protokollieren.

zu 9.) Die RRML, Version 2.0, bietet keine Möglichkeit, mittels Regeln Votierungsverfahren (siehe Abschnitt 3.3.1) nachzustellen. Wenn unstrukturierte Votierungsverfahren berücksich-

tigt werden sollen, dann muss die Syntax der RRML erweitert werden und dann müssen die Protokolle für die synchrone Aktualisierung (siehe Listing 4.1 auf Seite 71) und für einen Lesezugriff (siehe Listing 4.3 auf Seite 77) aufeinander abgestimmt werden. Das Protokoll eines Schreibzugriffs muss dahingehend angepasst werden, dass die Menge der synchron zu aktualisierenden Replikate einem Schreibquorum entspricht, und das Protokoll eines Lesezugriffs muss so geändert werden, dass die passenden Replikate einem Lesequorum entsprechen. Zusätzlich sind in diesem Fall Zeitstempel zwingend erforderlich. Wenn strukturierte Votierungsverfahren abgebildet werden, muss zusätzlich die Möglichkeit geschaffen werden, entsprechende Strukturen bei den Quoren zu berücksichtigen.

Anhang

A. Regelsprachen und regelbasierte Systeme

Beim Prototyp KARMA (siehe Abschnitt 7.3.2) handelt es sich um ein regelbasiertes System, weil er die regelbasierte Replikationsstrategie RegRess (siehe Kapitel 4) implementiert. Das Ziel der Auswertung der Replikationsregeln ist die Festlegung der Mengen von Replikaten, die bei einem Schreib- oder Lesezugriff betroffen sind, d.h. die Koordination der Zugriffe basiert auf der Inferenz der Replikationsregeln. Zwar bilden regelbasierte Systeme nicht die theoretische Grundlage der in dieser Dissertation vorgestellten Replikationsstrategie RegRess, weil die entsprechenden Konzepte der regelbasierten Systeme nur angewendet werden und daher auf die Replikation als solches fokussiert wird, dennoch wird in diesem Anhang ein kurzer Überblick gegeben. Dazu wird zunächst in Abschnitt A.1 auf Regeln eingegangen. In Abschnitt A.2 werden dann Regelsprachen vorgestellt, die die konzeptionelle Basis der in dieser Dissertation entwickelten RRML bilden (siehe Kapitel 5). Abschließend werden in Abschnitt A.3 regelbasierte Systeme diskutiert.

A.1. Regeln

Eine Regel in einem regelbasierten System hat die Bedeutung einer Richtlinie bzw. einer Vorschrift, mit der ein bestimmtes Verhalten festgelegt wird. Derartige Regeln werden häufig als Geschäftsregeln bezeichnet, die nach [TW01] wie folgt definiert sind:

„Geschäftsregeln sind Anweisungen, die bestimmte Teile der Geschäftsrichtlinien ausdrücken, wie z.B. die Festlegung der Geschäftsbedingungen, die Festlegung von Rechten und Pflichten und die Festlegung oder Bedingungen der Arbeitsabläufe.“

Regeln werden z.B. in Datenbankmanagementsystemen verwendet, um die Integrität zu sichern, oder in Email-Systemen, um Nachrichten zu filtern, zu verschieben oder zu sortieren. Im allgemeinen Sinn sind Regeln formalisierte Konditionalsätze in folgender Form:

WENN *Prämisse* DANN *Konklusion*

Die Prämisse wird auch als Antezedenz oder Bedingung und die Konklusion als Konsequenz bezeichnet. Eine Bedingung ist eine logische Aussage, insbesondere können logische Aussagen durch logische Operatoren verknüpft werden. Die Konklusion kann wie im folgenden Beispiel (a) eine Aussage sein oder wie im Beispiel (b) eine Aktion darstellen:

(a) WENN *Person hat Führerschein* DANN *Person darf Auto fahren*
(b) WENN *Aufzug ist defekt* DANN *benutze Treppe*

In [TW01] werden drei grundlegende Regeltypen unterschieden: Integritätsregeln, Ableitungsregeln und Reaktionsregeln. Eine Integritätsregel oder Integritätsbedingung besteht nur aus der Konklusion, die in Form einer Aussage immer erfüllt sein muss, z.B. die Integritätsbedingungen in einer Datenbank. Die Konklusion einer Ableitungsregel in Form einer Aussage lässt sich aus den Aussagen der Prämisse ableiten, wie im Beispiel (c):

(c) WENN *Tank ist leer* ODER *Motor defekt* DANN *Auto fährt nicht*

Die Konklusion einer Reaktionsregeln besteht aus einer Aktion, die ausgeführt wird, wenn die Prämisse (Bedingung) eingetreten ist. Reaktionsregeln werden häufig in Anwendungssystemen

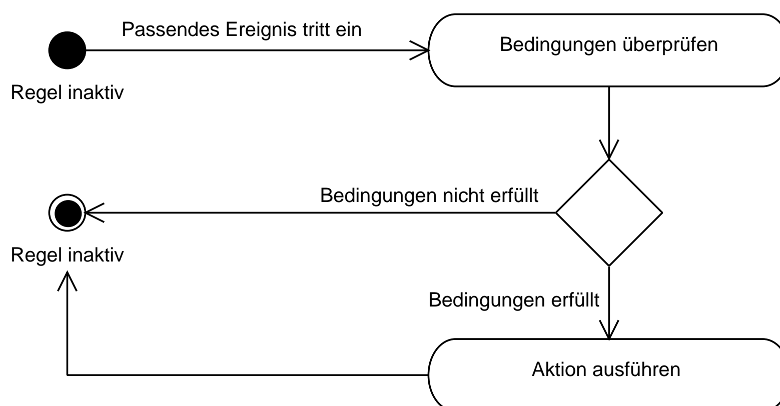

Abbildung A.1.: Aktivitätsdiagramm einer Reaktionsregel [Add05]

verwendet, um beim Eintreten einer Bedingung eine Aktion (Methode) aufzurufen, die in irgendeiner Form auf die Bedingung reagiert. Das folgende Beispiel (d) zeigt die Reaktion, nämlich das Löschen der Nachricht, eines Email-Programms, wenn eine Spam-Nachricht empfangen wird:

(d) WENN *Nachricht empfangen* UND *Nachricht ist Spam* DANN *lösche Nachricht*

Im Allgemeinen wird die Bedingung einer Reaktionsregel erst dann geprüft, wenn ein bestimmtes Ereignis eingetreten ist. Für eine Reaktionsregel muss also ein so genanntes beobachtbares Ereignis definiert sein, auf das reagiert wird. Der Ablauf einer derartigen Reaktionsregel ist in Abbildung A.1 dargestellt: Wenn das zur Reaktionsregel passende Ereignis eingetreten ist, dann wird die Bedingung geprüft. Wenn nun auch die Bedingung erfüllt ist, dann wird die Aktion ausgeführt. Allgemein kann eine Reaktionsregel somit wir folgt notiert werden:

(e) ON *Ereignis* IF *Bedingung* THEN *Aktion*

Im Beispiel (e) wird der Ereignisteil mit „ON“ eingeleitet, der Bedingungsteil mit „IF“ und der Aktionsteil mit „THEN“. Anstatt „THEN“ wird häufig auch „DO“ verwendet. Diese Form wird als ON-IF-THEN-Darstellung bezeichnet. Damit kann die Reaktionsregel im Beispiel (d) z.B. in der ON-IF-THEN-Darstellung wie im folgenden Beispiel (f) notiert werden:

(f) ON *Nachricht empfangen* IF *Nachricht ist Spam* THEN *lösche Nachricht*

Reaktionsregeln werden z.B. in aktiven Datenbanken [WC95] verwendet, wo die Bezeichnung ECA-Regel gebräuchlich ist (EVENT-CONDITION-ACTION).

A.2. Regelsprachen

Unter einer Regelsprache wird in dieser Dissertation eine Sprache verstanden, die es erlaubt, Regeln für ein regelbasiertes System zu formulieren. Häufig haben konkrete regelbasierte Systeme eine eigene Regelsprache, zumindest werden die Prämisse und die Konklusion einer Regel (siehe Abschnitt A.1) in einer speziellen Syntax notiert. Das gilt auch für die in dieser Dissertation entwickelte RRML (**R**eplication **R**ule **M**arkup **L**anguage, siehe Kapitel 5), die z.B. in den Bedingungen spezifische Funktionen für Replikationsanforderungen zulässt. In diesem Abschnitt wird auf zwei XML-basierte Regelsprachen fokussiert, die die Vorlage für die RRML bilden: Die RuleML (Rule Markup Language, [Bol01]) und die XRML (eXtensible Rule Markup Language, [LS03]).

Die RuleML basiert auf XML und RDF [MBK00] und soll zur Beschreibung allgemeiner Regeln dienen. Die Regeln sollen für unterschiedliche Anwendungssysteme nutzbar sein und zwischen den Anwendungssystemen austauschbar sein. In der RuleML werden vier Klassen von Regeln unterschieden: Reaktionsregeln, Integritätsregeln, Ableitungsregeln und Fakten. Die Klassen sind hierarchisch aufgebaut, d.h. Reaktionsregeln schließen Integritäts- und Ableitungsregeln ein und Ableitungsregeln schließen Fakten ein.

Gegenüber der in Abschnitt A.1 vorgenommenen Beschreibung von Integritätsregeln werden die Aussagen der Integritätsregeln der RuleML nicht in der Konklusion, sondern in der Prämisse notiert. Wenn die Prämisse einer Integritätsregel gilt, dann wird ein „Signal“ ausgelöst. Das Signal signalisiert die Erfüllung oder Verletzung einer Integritätsbedingung. Fakten in der RuleML sind Aussagen, die immer gelten. Für die vier Regelklassen der RuleML wurde die Syntax mittels XML-Kennungen (Tags) definiert.

Die XRML (nicht zu verwechseln mit der XrML (eXtensible rights Markup Language)) bietet die Möglichkeit, in Webseiten eingebettete Regeln zu identifizieren und zu markieren. Diese Regeln können in ein strukturiertes Austauschformat transformiert werden, um so von verschiedenen Anwendungen genutzt zu werden. Dafür bietet die XRML drei Teilsprachen:

1. Rule Identification Markup Language (RIML): Die RIML beinhaltet XML-Tags, die die Identifikation von Regeln erlaubt. Die ursprüngliche Form der Regel in natürlicher Sprache wird nicht verändert, aber innerhalb eines Regeltextes können Tags z.B. zur Identifikation von Funktionen etc. eingefügt werden. Ein Web-Browser ignoriert die RIML-Tags bei der Darstellung der Seite.
2. Rule Structure Markup Language (RSML): Mit der RSML lassen sich Regeln auf formale Weise beschreiben, d.h. ohne natürlichsprachliche Füllwörter oder dergleichen, sodass die Regeln der RSML von einer Anwendung interpretiert werden können. Eine Regel der RSML besteht aus einer Prämisse und einer Konklusion.
3. Rule Triggering Markup Language (RTML): Die RTML erlaubt die Spezifikation von Daten bzw. Datenquellen, die von einer Regel benötigt werden. Des Weiteren kann beschrieben werden, unter welchen Bedingungen die Regel angewendet wird und was das Resultat ist.

Mit den drei Teilsprachen bietet die XRML die Möglichkeit, natürlichsprachliche Regeln mittels der RIML in Webseiten einzubetten, die natürlichsprachlichen Regeln in der RIML in formale Regeln der RSML zu transformieren und diese mittels der RTML für regelbasierte Systeme einsatzfähig zu machen.

A.3. Regelbasierte Systeme

Ein regelbasiertes System [HR85] führt eine Schlußfolgerung (Inferenz) auf einer Menge von Regeln durch und liefert abschließend ein Ergebnis, wozu eine Wissensbasis (Knowledge Base) und ein Regelinterpreter (Inferenzmaschine, Inference Engine) benötigt werden. Die Wissensbasis enthält alle Regeln und Fakten, die das regelbasierte System zur Inferenz benötigt und die im Allgemeinen in einer systemspezifischen Struktur vorliegen. Der Regelinterpreter führt die Inferenz durch und liefert ein Ergebnis. Allgemein sollten regelbasierte Systeme nach [HR85] fünf Eigenschaften haben:

1. Sie betten geeignetes menschliches Wissen in bedingte *„wenn-dann“*-Regeln ein.
2. Ihre Fähigkeiten erhöhen sich proportional zur Erweiterung ihrer Wissensbasis.
3. Sie können eine große Auswahl von unter Umständen sehr komplexen Problemen lösen, indem sie die relevanten Regeln auswählen und deren Ergebnisse in adäquater Art und Weise kombinieren.
4. Sie bestimmen adaptiv die beste Sequenz von Regeln, die ausgeführt werden sollen.
5. Sie erläutern ihre Entscheidungen, indem sie ihre Schlussfolgerung zurückverfolgen und die verwendeten Regeln in eine natürliche Sprache übersetzen.

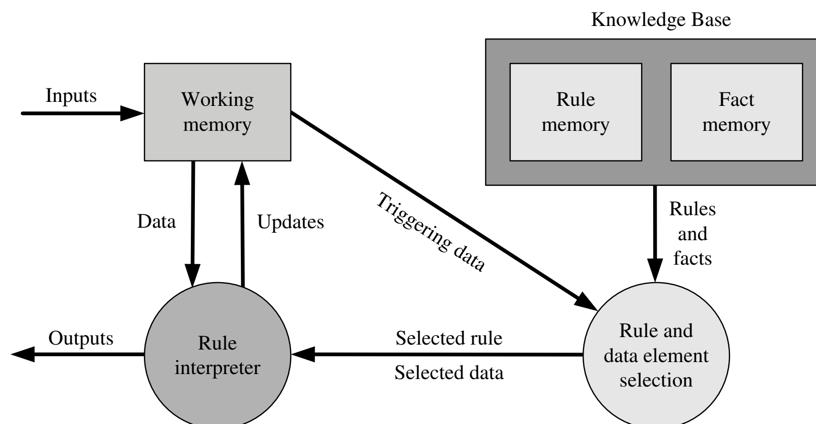

Abbildung A.2.: Ein einfaches regelbasiertes System nach [HR85]

Die Abbildung A.2 zeigt ein einfaches regelbasiertes System nach [HR85]. Im Anwendungsspeicher (Working Memory) werden Eingaben eines Clients, der das regelbasierte System nutzt, gespeichert. Auf Basis der Eingaben werden Parameter bestimmt, mittels derer die Regeln für die Inferenz ausgewählt werden. Die Regeln und die Fakten sind in der Wissensbasis (Knowledge Base) gespeichert, die direkt mit der Selektionskomponente (Rule and data element selection) verbunden ist. Die benötigten Regeln und Daten werden von der Selektionskomponente an Hand der Parameter im Arbeitsspeicher ausgewählt und an den Regelinterpreter (Rule interpreter) übergeben. Der Regelinterpreter berechnet unter Verwendung der Eingaben aus dem Anwendungsspeicher die Ergebnisse der ausgeführten Regeln. Dabei werden nicht nur Ausgaben erzeugt, sondern auch Daten im Anwendungsspeicher geändert oder eingefügt.

Wenn es zu Änderungen im Anwendungsspeicher kommt, dann muss die Selektionskomponente darauf reagieren und ggf. neue Regeln und Daten an den Regelinterpreter liefern. Dieser Zyklus wird wiederholt, insbesondere dann, wenn viele voneinander abhängige Regeln existieren. Daher sind Terminierung und Konfluenz wichtige Fragestellungen in regelbasierten Systemen. Konfluenz bedeutet, dass unabhängig von der Ausführungsreihenfolge der Regeln der gleiche Endzustand bzw. das gleiche Ergebnis erreicht wird.

B. Tabellarische Aufstellung der Anforderungen an das regelbasierte System

Die in dieser Dissertation aufgestellten Anforderungen an das regelbasierte System, das die Inferenz der Replikationsregeln der Replikationsstrategie RegRess durchführt, werden nachfolgend aufgelistet. Eine Anforderung kann einerseits den Regelinterpreter des regelbasierten Systems betreffen oder die Regel selbst bzw. die RRML, mittels derer die Syntax und die Semantik der Regeln festgelegt wird.

- **Anforderungen 1 - 15**: Durch diese Anforderungen, die den Regelinterpreter betreffen, wird ein gültiges Ergebnis der Inferenz gefordert. Es werden gültige Start- und Endzustände, auch im Fehlerfall, sowie Terminierung der verschiedenen Inferenzarten spezifiziert:
  - Anforderung 1, Seite: 79: InferenzSynchron: Startzustand
  - Anforderung 2, Seite: 79: InferenzSynchron: Endzustand
  - Anforderung 3, Seite: 79: InferenzSynchron: Fehler
  - Anforderung 4, Seite: 79: InferenzSynchron: Terminierung
  - Anforderung 5, Seite: 80: InferenzSynchron: Widersprüchliche Regeln
  - Anforderung 6, Seite: 80: InferenzAsynchron: Startzustand
  - Anforderung 7, Seite: 80: InferenzAsynchron: Endzustand
  - Anforderung 8, Seite: 80: InferenzAsynchron: Fehler
  - Anforderung 9, Seite: 80: InferenzAsynchron: Terminierung
  - Anforderung 10, Seite: 81: InferenzAsynchron: Widersprüchliche Regeln
  - Anforderung 11, Seite: 81: InferenzLesen: Startzustand
  - Anforderung 12, Seite: 81: InferenzLesen: Endzustand
  - Anforderung 13, Seite: 81: InferenzLesen: Fehler
  - Anforderung 14, Seite: 81: InferenzLesen: Terminierung
  - Anforderung 15, Seite: 81: InferenzLesen: Widersprüchliche Regeln
- **Anforderungen 16 - 18**: Die nachfolgenden Anforderungen betreffen alle Regeln und bestimmen die Struktur sowie Gültigkeitszeiträume:
  - Anforderung 16, Seite: 85: Struktur von Regeln
  - Anforderung 17, Seite: 85: Aktive Regeln bei einer Inferenz
  - Anforderung 18, Seite: 86: Gültigkeitszeitraum von Regeln
- **Anforderungen 19 - 23**: Mit den folgenden Anforderungen werden die fachlichen Konsistenzbedingungen der RRML festgelegt, wobei entweder einzelne Replikate oder die gesamte replizierte Datenbank angesprochen werden:
  - Anforderung 19, Seite: 86: Zustand eines Replikats
  - Anforderung 20, Seite: 87: Kohärenzbedingungen
  - Anforderung 21, Seite: 87: Periodische Aktualisierung
  - Anforderung 22, Seite: 88: Konsistenzgradbedingung
  - Anforderung 23, Seite: 88: Lesen veralteter Daten
- **Anforderungen 24 - 30**: Diese Anforderungen zielen auf die technischen Konsistenzbedingungen, mit denen auf Zustandsänderungen der beteiligten Komponenten mit Replikat reagiert werden kann:
  - Anforderung 24, Seite: 89: Verfügbarkeit
  - Anforderung 25, Seite: 89: Performance bei Schreibzugriffen

  - Anforderung 26, Seite: 89: Performance bei Lesezugriffen
  - Anforderung 27, Seite: 89: Datengröße
  - Anforderung 28, Seite: 90: Konfliktbedingung
  - Anforderung 29, Seite: 90: Regelung von Widersprüchen
  - Anforderung 30, Seite: 90: Vergabe von Prioritäten für Regeln

- **Anforderungen 31 - 37**: Die verbleibenden Anforderungen dienen der Erhöhung der Sprachmächtigkeit der RRML:
  - Anforderung 31, Seite: 114: Weglassen von expliziten Replikationsregeln
  - Anforderung 32, Seite: 114: Default-Regeln
  - Anforderung 33, Seite: 114: Regeln für alle Replikate einer Komponente
  - Anforderung 34, Seite: 114: Beschreibungsmöglichkeit von Regeln
  - Anforderung 35, Seite: 114: Gruppierung von Regeln
  - Anforderung 36, Seite: 115: Auslagerung von Regeln
  - Anforderung 37, Seite: 115: Allgemeine Aktualisierungsart

C. UML-Modell des KARMA-Prototypen

In Abschnitt 7.3.2 wurde die Implementierung des Prototyps KARMA vorgestellt. Das dort gezeigte UML-Implementierungsmodell des KARMA (siehe Abbildung 7.6 auf Seite 188) zeigt in einer Abbildung die implementierten Komponenten mit ihren angebotenen und benötigten Schnittstellen. Aus Platzgründen wurden dort nur die wichtigsten Methoden ohne Parameter gezeigt. Hier wird die komplette UML-Modellierung dargestellt, wobei die einzelnen Komponenten bzw. deren UML-Modelle in den folgenden Abbildungen gezeigt werden (angegeben sind die Komponenten (Klassen) sowie die angebotenen (implementierten) Schnittstellen):

- Abbildung C.1: `AsynchronousWriter` mit `IAsynchronousWriter`
- Abbildung C.2: `DataTransformer` mit `IDataTransformer` und `IPhysicalAccess`
- Abbildung C.3: `Reader` mit `IReader`
- Abbildung C.4: `ReplicaQueueManager` mit `IQueueAccess`
- Abbildung C.5: `RequestHandler` mit `ILogicalAccess` und `IRequestHandler`
- Abbildung C.6: `RuleInterpreter` mit `IRuleInterpreter`
- Abbildung C.7: `SynchronousWriter` mit `ISynchronousWriter`
- Abbildung C.8: `TxController` mit `INestedTx`, `ISubTx` und `IExtendedTx`
- Abbildung C.9: `KarmaErrors`, `Rules`, etc.

Auf eine detaillierte Beschreibung der einzelnen Klassen und Schnittstellen wird in dieser Dissertation verzichtet, weil die gegenüber dem Entwurf (siehe Kapitel 6) hinzukommenden Methoden im Wesentlichen Implementierungshilfen darstellen. In der Klasse `KarmaErrors` (siehe Abbildung C.9 auf Seite 234) sind Konstanten deklariert, die auch in den Log-Informationen angeben werden. Hierbei handelt es sich hauptsächlich um die Ereignis- und Aktionsarten sowie um Rückgabewerte mit folgender Bedeutung:

100	`update()`
101	`sync_update()`
102	`async_update()`
103	`changeableTrue()`
104	`changeableFalse()`
200	`write()`
201	`later()`
202	`one()`
203	`all()`
300	`read()`
301	`getConsistency()`
302	`getPerformance()`
900	Ergebnis der Widerspruchsbehandlung: `Action`
901	Ergebnis der Widerspruchsbehandlung: `NoAction`
902	Ergebnis der Widerspruchsbehandlung: `NoSolution`
991	Verfahren der Widerspruchsbehandlung: `CGP`
992	Verfahren der Widerspruchsbehandlung: `FGP`
993	Verfahren der Widerspruchsbehandlung: `PCG`
994	Verfahren der Widerspruchsbehandlung: `PFG`

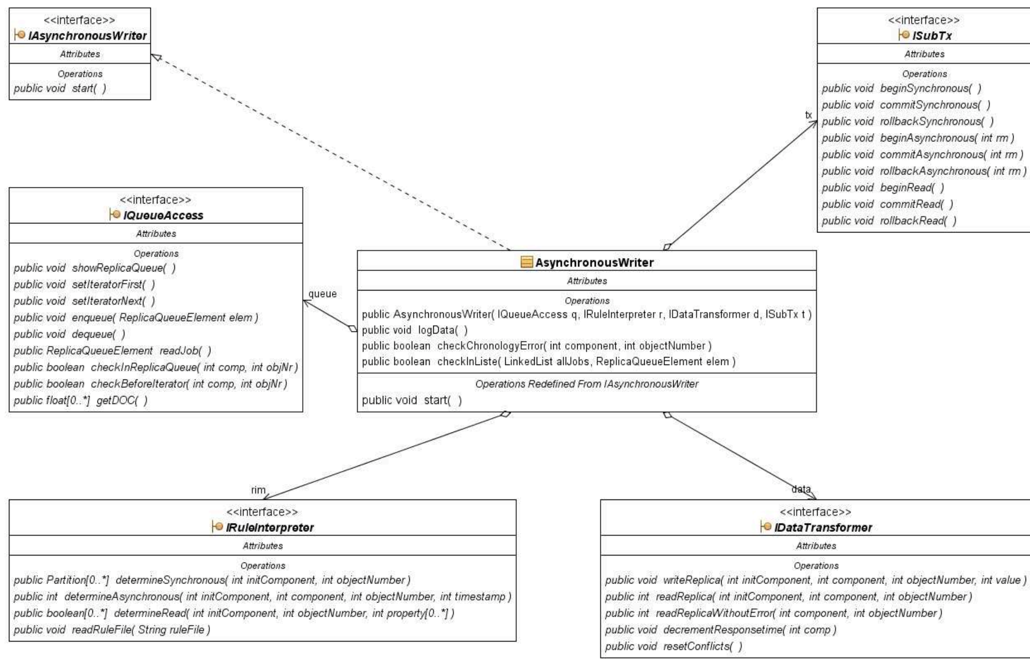

Abbildung C.1.: Implementierte Komponente AsynchronousWriter

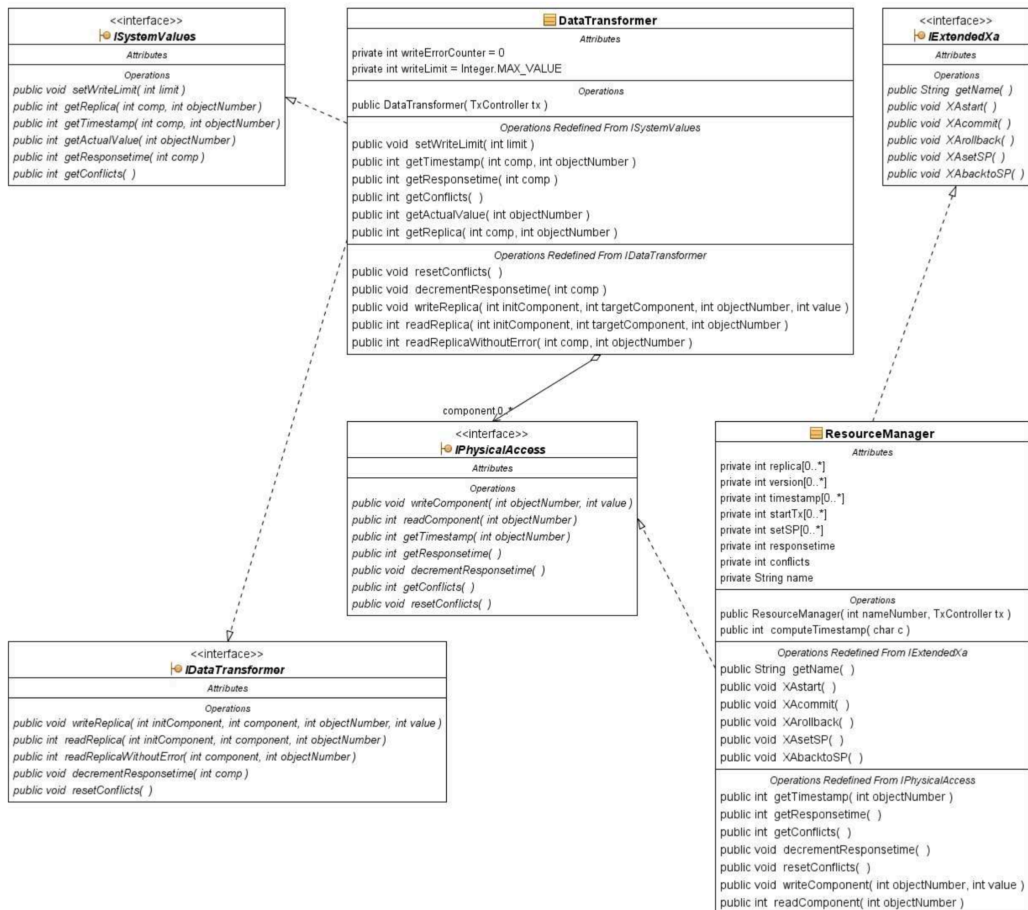

Abbildung C.2.: Implementierte Komponente DataTransformer

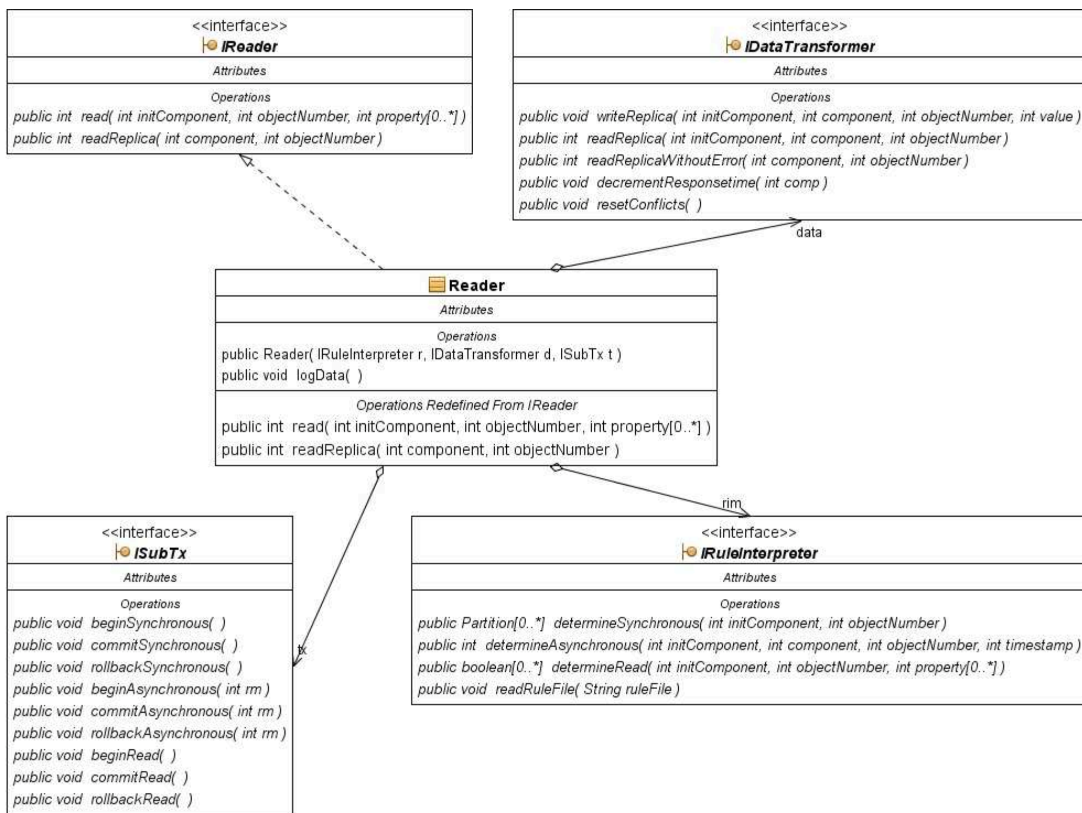

Abbildung C.3.: Implementierte Komponente Reader

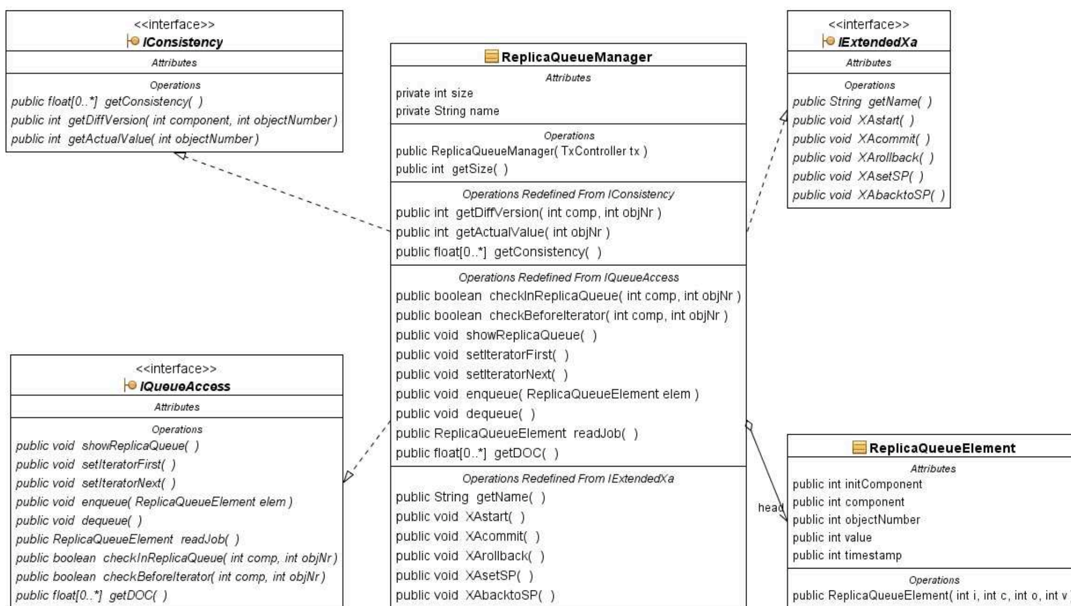

Abbildung C.4.: Implementierte Komponente ReplicaQueueManager

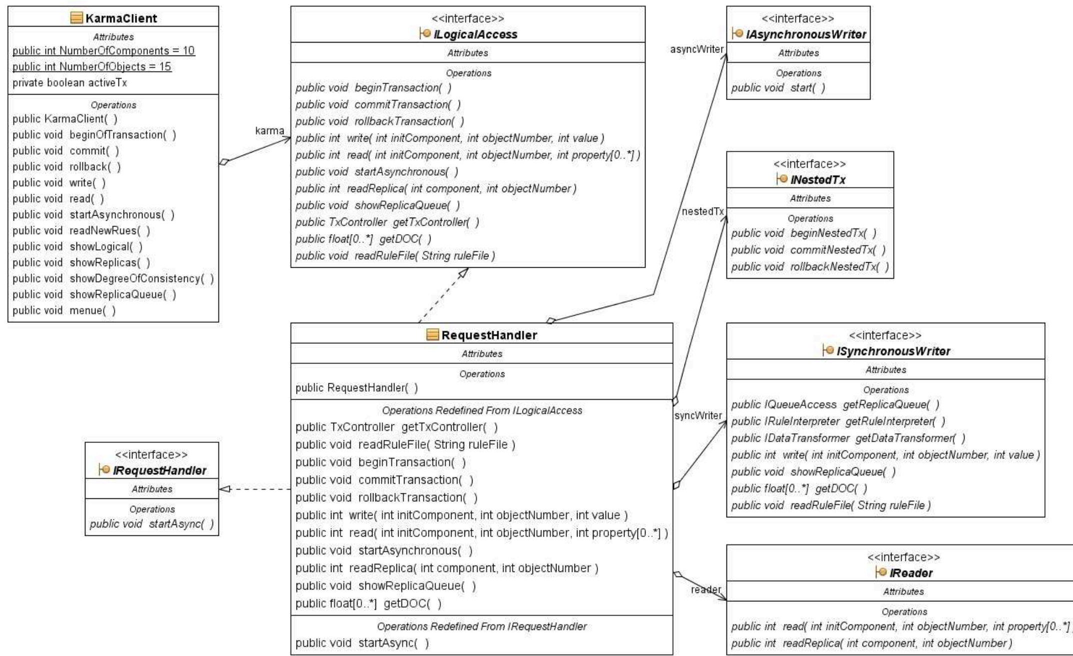

Abbildung C.5.: Implementierte Komponente RequestHandler

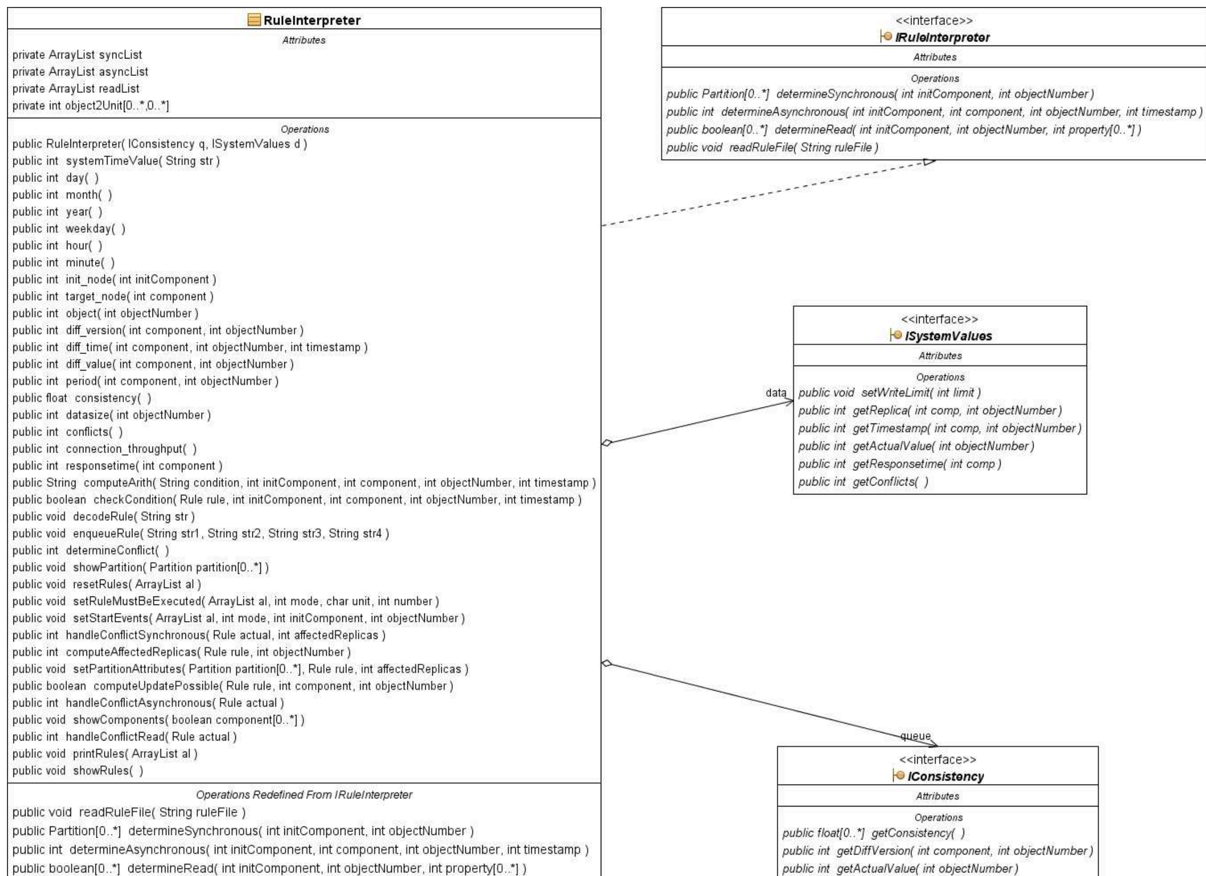

Abbildung C.6.: Implementierte Komponente RuleInterpreter

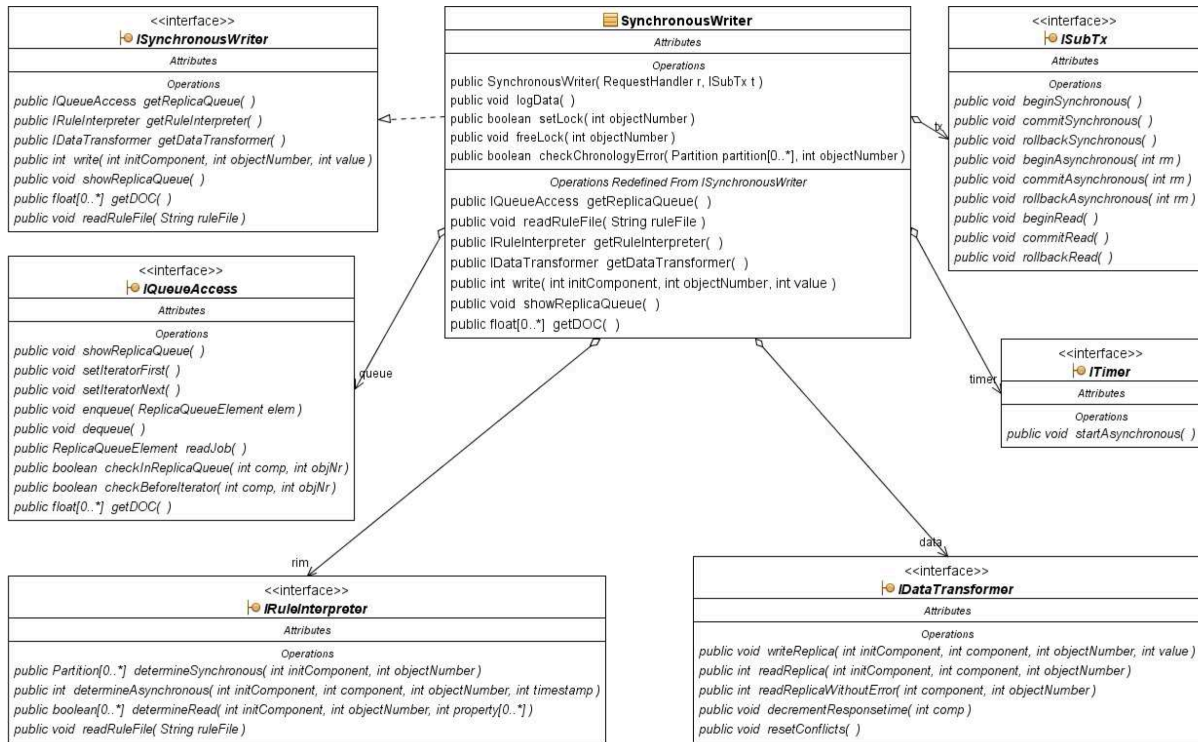

Abbildung C.7.: Implementierte Komponente SynchronousWriter

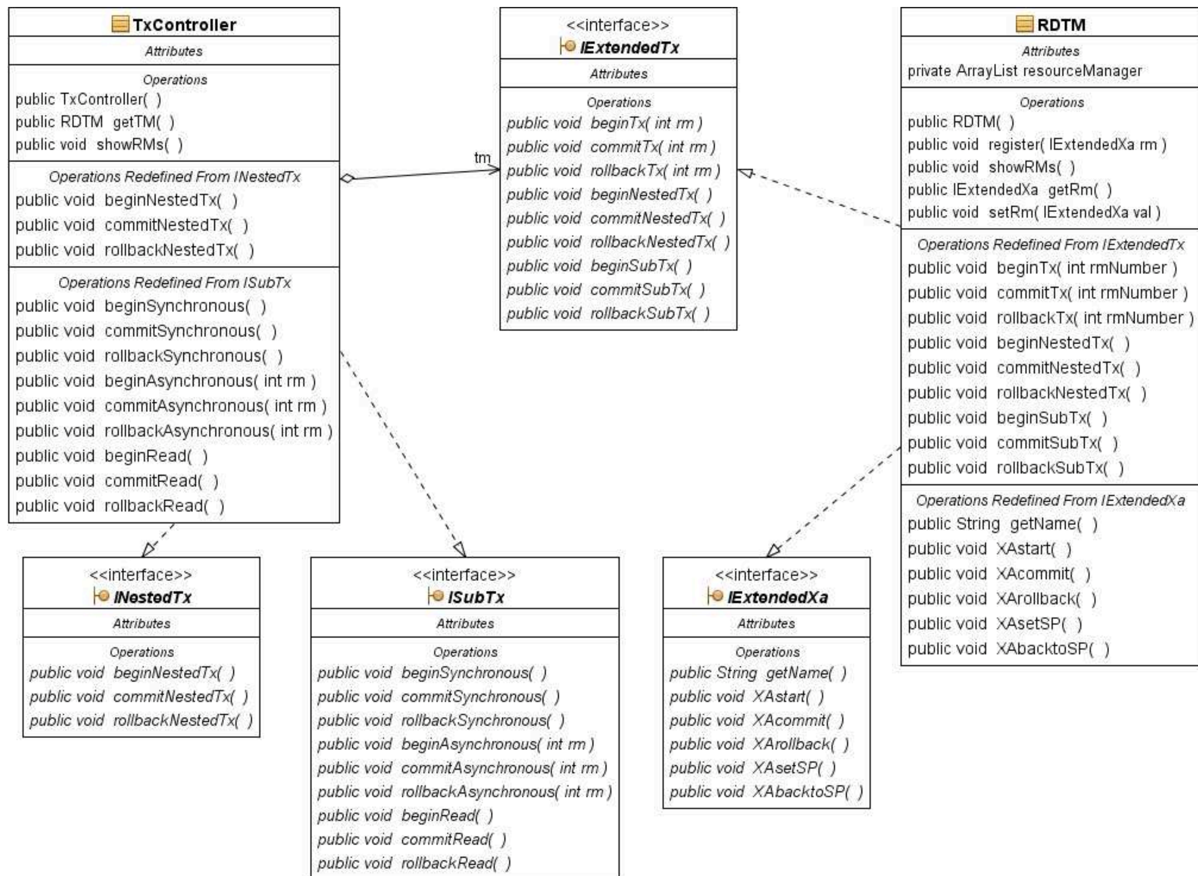

Abbildung C.8.: Implementierte Komponente TxController

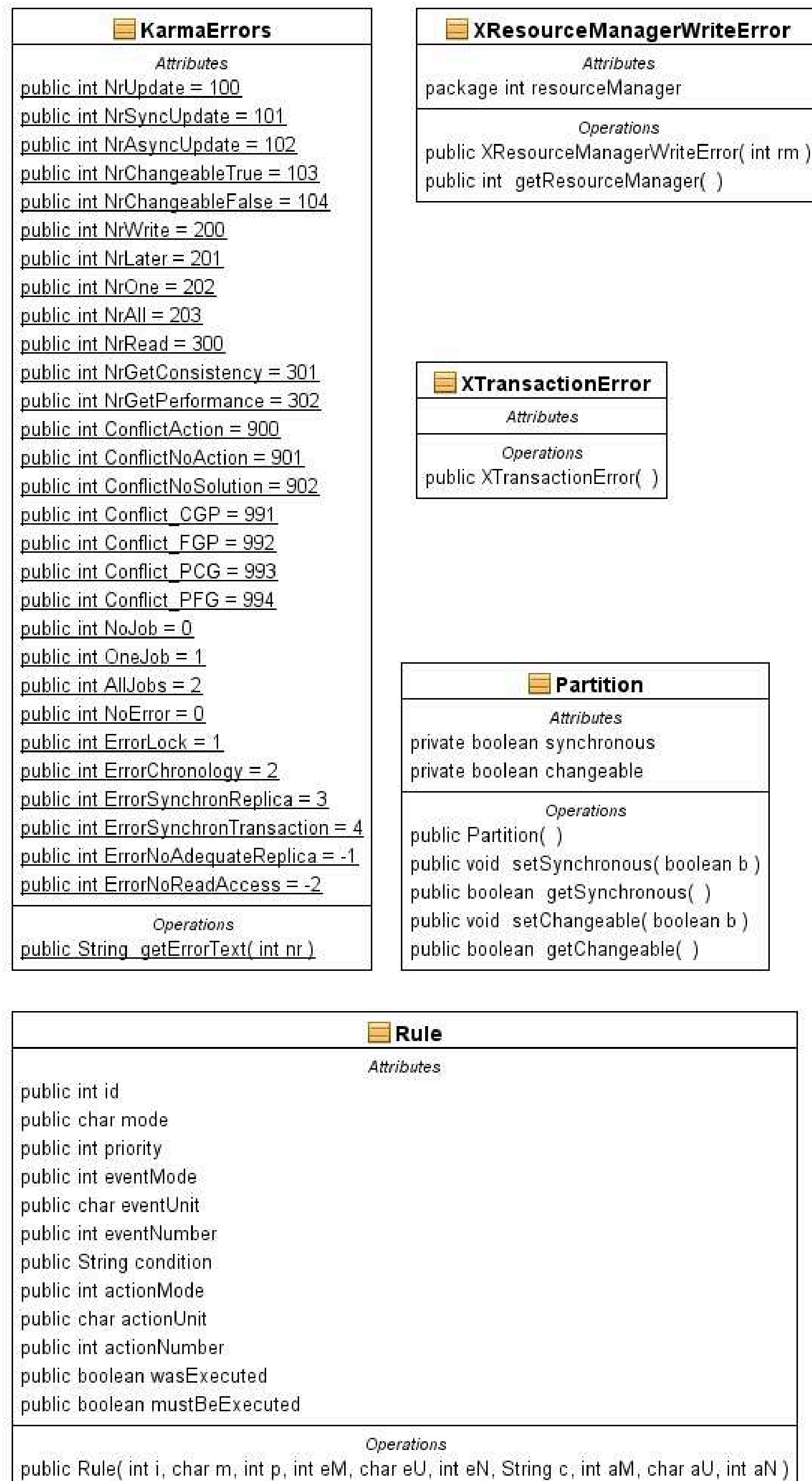

Abbildung C.9.: Implementierte Komponenten KarmaErrors, Partition, Rule, XTransactionError, XResourceManagerWriteError

Definitionen

Nummerische Sortierung

Definition 1. Logisches Datenobjekt . . . 29
Definition 2. Physisches Datenobjekt . . . 29
Definition 3. Replikation . . . 29
Definition 4. Replikat . . . 30
Definition 5. Aktuelles Replikat . . . 30
Definition 6. Replikationsgrad . . . 30
Definition 7. Replizierte und nicht-replizierte Datenbank . . . 30
Definition 8. Rechner mit Replikat . . . 30
Definition 9. System mit Replikat . . . 31
Definition 10. Replikationsstrategie . . . 31
Definition 11. Lese-Replikat . . . 32
Definition 12. Unikat . . . 32
Definition 13. Duplikat . . . 32
Definition 14. Konflikt . . . 32
Definition 15. Schreib-/Lesekonflikt$_R$ . . . 33
Definition 16. Schreib-/Schreibkonflikt$_R$ . . . 33
Definition 17. Replikationskonsistenz . . . 34
Definition 18. 1-Kopien-Serialisierbarkeit . . . 34
Definition 19. Letztendliche Konsistenz . . . 34
Definition 20. Abgeschwächte Konsistenz in replizierten Datenbanben . . . 35
Definition 21. Session-Garantie . . . 36
Definition 22. Kohärenzbedingung . . . 37
Definition 23. Snapshot-Aktualisierungszeitpunkt . . . 37
Definition 24. Datenfrische . . . 38
Definition 25. Frische eines Replikats . . . 38
Definition 26. Frische einer replizierten Datenbank . . . 38
Definition 27. Bemerkte Frische . . . 39
Definition 28. Probabilistische Konsistenz . . . 39
Definition 29. Probabilistische Frische . . . 39
Definition 30. Synchrone Replikationsstrategie . . . 42
Definition 31. Asynchrone Replikationsstrategie . . . 42
Definition 32. Komponente . . . 60
Definition 33. Replikationsmanager . . . 60
Definition 34. Komponente mit Replikat . . . 61
Definition 35. Replikationseinheit . . . 65
Definition 36. Synchrone Aktualisierung von Replikaten . . . 67
Definition 37. Asynchrone Aktualisierung von Replikaten . . . 68
Definition 38. Versionsabstand . . . 92
Definition 39. Konsistenzgrad eines Replikats . . . 92
Definition 40. Konsistenzgrad einer voll-replizierten Datenbank . . . 93
Definition 41. Versionsabstand einer voll-replizierten Datenbank . . . 93

Definition 42. Konsistenzgrad von Lesezugriffen 95
Definition 43. Konsistenzgrad durch Simulation 96
Definition 44. Versionsabstandswahrscheinlichkeit 97
Definition 45. Zufallsvariable für den Versionsabstand 97
Definition 46. Zufallsvariable für den normierten Versionsabstand 97
Definition 47. Erwarteter Versionsabstand eines Replikats 97
Definition 48. Erwarteter Konsistenzgrad eines Replikats 98
Definition 49. Erwarteter Versionsabstand einer Replikationseinheit 98
Definition 50. Erwarteter Konsistenzgrad einer Replikationseinheit 98
Definition 52. Probabilistischer Versionsabstand 99
Definition 52. Probabilistischer Konsistenzgrad 99
Definition 53. Verfuegbarkeit 106
Definition 54. Wahrscheinlichkeit für die synchrone Aktualisierung eines Replikats 107
Definition 55. Verfügbarkeit eines Replikats 107
Definition 56. Verfügbarkeit eines logischen Objekts 107
Definition 57. Verfügbarkeit einer replizierten Datenbank 108
Definition 58. Antwortzeit 108
Definition 59. Mittlere Antwortzeit eines Replikats 108
Definition 60. Mittlere Antwortzeit eines logischen Objekts 109
Definition 61. Mittlere Antwortzeit einer replizierten Datenbank 109

Alphabetische Sortierung

Definition 18. 1-Kopien-Serialisierbarkeit 34
Definition 20. Abgeschwächte Konsistenz in replizierten Datenbanben 35
Definition 5. Aktuelles Replikat 30
Definition 58. Antwortzeit 108
Definition 37. Asynchrone Aktualisierung von Replikaten 68
Definition 31. Asynchrone Replikationsstrategie 42
Definition 27. Bemerkte Frische 39
Definition 24. Datenfrische 38
Definition 13. Duplikat 32
Definition 50. Erwarteter Konsistenzgrad einer Replikationseinheit 98
Definition 48. Erwarteter Konsistenzgrad eines Replikats 98
Definition 49. Erwarteter Versionsabstand einer Replikationseinheit 98
Definition 47. Erwarteter Versionsabstand eines Replikats 97
Definition 26. Frische einer replizierten Datenbank 38
Definition 25. Frische eines Replikats 38
Definition 22. Kohärenzbedingung 37
Definition 32. Komponente 60
Definition 34. Komponente mit Replikat 61
Definition 14. Konflikt 32
Definition 43. Konsistenzgrad durch Simulation 96
Definition 39. Konsistenzgrad eines Replikats 92
Definition 40. Konsistenzgrad einer voll-replizierten Datenbank 93
Definition 42. Konsistenzgrad von Lesezugriffen 95
Definition 11. Lese-Replikat 32
Definition 19. Letztendliche Konsistenz 34
Definition 1. Logisches Datenobjekt 29
Definition 61. Mittlere Antwortzeit einer replizierten Datenbank 109
Definition 60. Mittlere Antwortzeit eines logischen Objekts 109
Definition 59. Mittlere Antwortzeit eines Replikats 108
Definition 2. Physisches Datenobjekt 29
Definition 29. Probabilistische Frische 39
Definition 28. Probabilistische Konsistenz 39
Definition 52. Probabilistischer Konsistenzgrad 99
Definition 52. Probabilistischer Versionsabstand 99
Definition 8. Rechner mit Replikat 30
Definition 4. Replikat 30
Definition 3. Replikation 29
Definition 35. Replikationseinheit 65
Definition 6. Replikationsgrad 30
Definition 17. Replikationskonsistenz 34
Definition 33. Replikationsmanager 60
Definition 7. Replizierte und nicht-replizierte Datenbank 30
Definition 10. Replikationsstrategie 31
Definition 15. Schreib-/Lesekonflikt$_R$ 33
Definition 16. Schreib-/Schreibkonflikt$_R$ 33
Definition 21. Session-Garantie 36
Definition 23. Snapshot-Aktualisierungszeitpunkt 37
Definition 36. Synchrone Aktualisierung von Replikaten 67
Definition 30. Synchrone Replikationsstrategie 42
Definition 9. System mit Replikat 31

Definition 12. Unikat 32
Definition 53. Verfuegbarkeit 106
Definition 57. Verfügbarkeit einer replizierten Datenbank 108
Definition 56. Verfügbarkeit eines logischen Objekts 107
Definition 55. Verfügbarkeit eines Replikats 107
Definition 38. Versionsabstand 92
Definition 41. Versionsabstand einer voll-replizierten Datenbank 93
Definition 44. Versionsabstandswahrscheinlichkeit 97
Definition 54. Wahrscheinlichkeit für die synchrone Aktualisierung eines Replikats 107
Definition 46. Zufallsvariable für den normierten Versionsabstand 97
Definition 45. Zufallsvariable für den Versionsabstand 97

Abbildungsverzeichnis

1.1. Ausschnitt aus der elektronischen ADT- und Leistungsdatenkommunikation am Universitätsklinikum Leipzig [NHW+02] . . . 2
1.2. Horizontale Integration zur Unterstützung der Geschäftsprozesse [Has00] . . . 4

2.1. X/Open-Modell eines zentralisierten Transaktionssystems (nach [Lam94]) . . . 16
2.2. Queued Transaction Processing Modell (nach [BN97]) . . . 21

3.1. Zielkonflikte der Datenreplikation (nach [Has97]) . . . 28
3.2. Klassifizierung von Replikationsstrategien mit jeweils zwei Beispielen . . . 42
3.3. Aufgaben des Konfliktmanagements als UML-Klassendiagramm . . . 46
3.4. Enge Kopplung klinischer Informationssysteme [NHW+02] . . . 50

4.1. Beispiel einer Systemumgebung als UML-Verteilungsdiagramm . . . 61
4.2. Allgemeiner Ablauf eines Zugriffs auf ein logisches Datenobjekt . . . 64
4.3. Struktur der Replikationsregeln als UML-Klassendiagramm . . . 66
4.4. Partitionierung in synchron und asynchron zu aktualisierende Replikat . . . 69
4.5. Versionsabstandswahrscheinlichkeiten mittels zeitstetiger Markov-Kette . . . 101
4.6. Zeitstetige Markov-Kette für Versionsabstand von 4 . . . 103

6.1. Die Architektur des adaptiven Replikationsmanagers (ARM, [NHHT03]) . . . 154
6.2. Grob-Architektur des KARMA . . . 155
6.3. Feingranulare Softwarearchitektur des KARMA . . . 156
6.4. Komponente AsynchronousWriter . . . 158
6.5. Komponente DataTransformer . . . 159
6.6. Komponente Reader . . . 160
6.7. Komponente RegRessReplicator . . . 161
6.8. Komponente ReplicaQueueManager . . . 161
6.9. Komponente RequestHandler . . . 162
6.10. Komponente RuleInterpreter . . . 163
6.11. Komponente SynchronousWriter . . . 164
6.12. Komponente Timer . . . 165
6.13. Komponente TxController . . . 165
6.14. Die Architektur des RD-TM [Aus06] . . . 175

7.1. Integrationsmodelle [RMB00] . . . 178
7.2. Realisierung des ARM mittels der Java EE Technologie [NHHT03] . . . 179
7.3. Ein-/Ausgabemodell des F4SR [Fro06] . . . 184
7.4. Simulationsereignisse des F4SR [Fro06] . . . 185
7.5. Grobe Architektur des F4SR [Fro06] . . . 186
7.6. Vereinfachtes UML-Implementierungsmodell des KARMA . . . 188
7.7. Screenshot des Menüs im KARMA-Prototypen . . . 194
7.8. Screenshot der Log-Daten des KARMA-Prototypen: Regelinterpreter . . . 195
7.9. Screenshot der Log-Daten des KARMA-Prototypen: Protokoll Schreibzugriff . . . 196
7.10. F4SR-Menüs „Settings“ und „System“ . . . 201

7.11. Konsistenzgrad bzgl. der Regeln im Listing 7.3 202
7.12. Konsistenzgrad bzgl. der Regeln im Listing 7.4 203
7.13. Kreisdiagramme des F4SR 204
7.14. Antwortzeit-Diagramm des F4SR 205

A.1. Aktivitätsdiagramm einer Reaktionsregel [Add05] 224
A.2. Ein einfaches regelbasiertes System nach [HR85] 226

C.1. Implementierte Komponente AsynchronousWriter 230
C.2. Implementierte Komponente DataTransformer 230
C.3. Implementierte Komponente Reader 231
C.4. Implementierte Komponente ReplicaQueueManager 231
C.5. Implementierte Komponente RequestHandler 232
C.6. Implementierte Komponente RuleInterpreter 232
C.7. Implementierte Komponente SynchronousWriter 233
C.8. Implementierte Komponente TxController 233
C.9. Implementierte Komponenten KarmaErrors, Partition, Rule, XTransactionError, XResourceManagerWriteError 234

Listings

4.1. Synchroner Schreibzugriff . . . 71
4.2. Asynchroner Schreibzugriff . . . 74
4.3. Lesezugriff . . . 77
5.1. Backus-Naur-Form der Regeln in RRML . . . 117
5.2. Ablauf einer Aktion bei der synchronen Aktualisierung . . . 121
5.3. Ablauf einer Aktion bei der asynchronen Aktualisierung . . . 126
5.4. Ermittlung der Ergebnisreferenzen . . . 129
5.5. Ablauf einer Aktion bei einem Lesezugriff . . . 131
5.6. Behandlung von widersprüchlichen Regeln . . . 133
5.7. XML-basierte RRML-Syntax . . . 144
5.8. Mapping mittels XML . . . 146
7.1. Datei rules.txt . . . 190
7.2. Ablauf der Inferenz beim Regelinterpreter, Version 2.0 . . . 191
7.3. Regeln, die einen Versionsabstand von höchstens 4 zulassen . . . 202
7.4. Regeln, die einen Versionsabstand von höchstens 10 zulassen . . . 202

Literaturverzeichnis

[AA90] Divyakant Agrawal und Amr El Abbadi. The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. In Dennis McLeod, Ron Sacks-Davis, und Hans-Jörg Schek, Hrsg., *VLDB*, Seiten 243–254. Morgan Kaufmann, 1990.

[AA92a] Divyakant Agrawal und Amr El Abbadi. The Generalized Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. *ACM Trans. Database Syst.*, 17(4):689–717, 1992.

[AA92b] Divyakant Agrawal und Amr El Abbadi. Resilient Logical Structures for Efficient Management of Replicated Data. In Li-Yan Yuan, Hrsg., *VLDB*, Seiten 151–162. Morgan Kaufmann, 1992.

[AA96] Divyakant Agrawal und Amr El Abbadi. Using Reconfiguration for Efficient Management of Replicated Data. *IEEE Trans. Knowl. Data Eng.*, 8(5):786–801, 1996.

[AAC91] Mustaque Ahamad, Mostafa H. Ammar, und Shun Yan Cheung. Multidimensional voting. *ACM Trans. Comput. Syst.*, 9(4):399–431, 1991.

[AAS97] Divyakant Agrawal, Amr El Abbadi, und Robert C. Steinke. Epidemic Algorithms in Replicated Databases (Extended Abstract). In *Principles of Database Systems PODS*, Seiten 161–172. ACM Press, 1997.

[ABGMA88] Rafael Alonso, Daniel Barbará, Hector Garcia-Molina, und Soraya Abad. Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems. In *Conference on Extending Database Technology EDBT*, Band 303 der Reihe *Lecture Notes in Computer Science*, Seiten 443–468. Springer, 1988.

[ACPT99] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, und Riccardo Torlone. *Database Systems: Concepts, Languages & Architectures.* McGraw-Hill Higher Education, 1999.

[AD76] Peter Alsberg und J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *ICSE*, Seiten 562–570, 1976.

[Add05] Jan Stefan Addicks. *Entwicklung eines XML-basierten Regelsprachensystems für Replikationsstrategien.* Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik, Abteilung Software Engineering, 2005.

[AG96] Sarita V. Adve und Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AHW95] Alexander Aiken, Joseph M. Hellerstein, und Jennifer Widom. Static Analysis Techniques for Predicting the Behavior of Active Database Rules. *ACM Trans. Database Syst.*, 20(1):3–41, 1995.

[AL80] Michel E. Adiba und Bruce G. Lindsay. Database Snapshots. In *Sixth International Conference on Very Large Data Bases, VLDB*, Seiten 86–91, Montreal, Quebec, Canada. IEEE Computer Society, 1980.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, und Carl E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[And01] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons Inc, 2001.

[ASC85] Amr El Abbadi, Dale Skeen, und Flaviu Cristian. An Efficient, Fault-Tolerant Protocol for Replicated Data Management. In *Symposium on Principles of Database Systems PODS*, Seiten 215–229, Portland, Oregon. ACM, 1985.

[AT89] Amr El Abbadi und Sam Toueg. Maintaining Availability in Partitioned Replicated Databases. *ACM Trans. Database Syst.*, 14(2):264–290, 1989.

[Aus06] Andreas Austing. *Erweiterte Transaktionskonzepte in der Java Enterprise Edition*. Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik, Abteilung Software Engineering, 2006.

[AWH92] Alexander Aiken, Jennifer Widom, und Joseph M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *SIGMOD Conference*, Seiten 59–68, 1992.

[BBG+95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, und Patrick E. O'Neil. A Critique of ANSI SQL Isolation Levels. In Michael J. Carey und Donovan A. Schneider, Hrsg., *ACM SIGMOD International Conference on Management of Data, San Jose, California*, Seiten 1–10. ACM Press, 1995.

[BCN99] Jerry Banks, John S. Carson, und Barry L. Nelson. *Discrete-Event System Simulation*. Prentice Hall, 2. Auflage, 1999.

[BCSS99] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, und Daniel C. Sturman. A Case for Message Oriented Middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, Seiten 1–18. Springer-Verlag, London, UK, 1999.

[BD96] T. Beuter und P. Dadam. Prinzipien der Replikationskontrolle in verteilten Systemen. *Informatik Forschung und Entwicklung*, 11(4):203–212, 1996.

[BE05] Jon Barwise und John Etchemendy. *Sprache, Beweis und Logik. Band I. Aussagen- und Prädikatenlogik*. Mentis-Verlag, 2005.

[BG81] Philip A. Bernstein und Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[BG83a] Philip A. Bernstein und Nathan Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[BG83b] Philipp A. Bernstein und Nathan Goodman. The Failure and Recovery Problem for Replicated Databases. In *Symposium on Princiles of Distributed Computing*, Seiten 114–122. ACM, Montreal, Quebec, Canada, 1983.

[BG84] Philip A. Bernstein und Nathan Goodman. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. *ACM Trans. Database Syst.*, 9(4):596–615, 1984.

[BGM94] Daniel Barbará und Hector Garcia-Molina. Replicated Data Management in Mobile Environments: Anything New Under the Sun? In Claude Girault, Hrsg., *Applications in Parallel and Distributed Computing*, Band A-44 der Reihe *IFIP Transactions*, Seiten 237–246. North-Holland, 1994.

[BGMS86] Daniel Barbará, Hector Garcia-Molina, und Annemarie Spauster. Protocols for Dynamic Vote Reassignment. In *PODC*, Seiten 195–205, 1986.

[BGMS92] Yuri Breitbart, Hector Garcia-Molina, und Abraham Silberschatz. Overview of Multidatabase Transaction Management. *VLDB J.*, 1(2):181–239, 1992.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, und Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[BHM90] Philip A. Bernstein, Meichun Hsu, und Bruce Mann. Implementing Recoverable Requests Using Queues. In *[GMJ90]*, Seiten 112–122, 1990.

[Bis02] Ludger Bischofs. *Anbindung von SAP R/3 über die J2EE Connector-Architektur*. Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik, Abteilung Software Engineering, 2002.

[BKK+03] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, und Ion Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46(2):43–48, Februar 2003.

[BN84] Andrew Birrell und Bruce Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[BN96] P. Bernstein und E. Newcomer. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, 1996.

[BN97] Philip Bernstein und Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, 1997.

[Bol01] Harold Boley. The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations. In *International Conference on Applications of Prolog (INAP 2001)*, Seiten 124–139, 2001.

[Bör03] Josef Börcsök. *Elektronische Sicherheitssysteme*. Hüthig, 2003.

[BP04] Mokrane Bouzeghoub und Verónika Peralta. A Framework for Analysis of Data Freshness. In *International Workshop on Information Quality in Information Systems IQIS*, Seiten 59–67, Paris, France. ACM, 2004.

[BR92] B. R. Badrinath und Krithi Ramamritham. Semantics-based concurrency control: beyond commutativity. *ACM Trans. Database Syst.*, 17(1):163–199, 1992.

[Bre83] Pearl Brereton. Detection and Resolution of Inconsistencies among Distributed Replicates of Files. *Operating Systems Review*, 17(1):10–15, 1983.

[Bur97] Marie Buretta. *Data Replication: Tools and Techniques for Managing Distributed Information*. John Wiley & Sons, 1997.

[CAA90] Shun Yan Cheung, Mostafa H. Ammar, und Mustaque Ahamad. The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data. In *ICDE*, Seiten 438–445. IEEE Computer Society, 1990.

[CAA92] Shun Yan Cheung, Mostafa H. Ammar, und Mustaque Ahamad. The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data. *IEEE Trans. Knowl. Data Eng.*, 4(6):582–592, 1992.

[CDK00] George Coulouris, Jean Dollimore, und Tim Kindberg. *Distributed Systems: Concepts and Design (3rd Edition)*. Addison Wesley, 2000.

[CES71] E. G. Coffman, M. Elphick, und A. Shoshani. System Deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[CGM00] Junghoo Cho und Hector Garcia-Molina. Synchronizing a database to Improve Freshness. In Weidong Chen, Jeffrey F. Naughton, und Philip A. Bernstein, Hrsg., *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Seiten 117–128, 2000.

[CHK05] Stefan Conrad, Wilhelm Hasselbring, und Arne Koschel. *Enterprise Application Integration. Grundlagen - Konzepte - Entwurfsmuster - Praxisbeispiele*. Spektrum Akademischer Verlag, 2005.

[CHKS91] Stefano Ceri, Maurice A. W. Houtsma, Arthur M. Keller, und Pierangela Samarati. A classification of update methods for replicated databases. Technischer Bericht STAN-CS-91-1392, CS Dept., Stanford University, Stanford, CA, USA, 1991.

[CHKS92] Stefano Ceri, Maurice A. W. Houtsma, Arthur M. Keller, und Pierangela Samarati. The Case for Independent Updates. In *Workshop on the Management of Replicated Data*, Seiten 17–19, 1992.

[CHKS95] Stefano Ceri, Maurice A. W. Houtsma, Arthur M. Keller, und Pierangela Samarati. Independent Updates and Incremental Agreement in Replicated Databases. *Distributed and Parallel Databases*, 3(3):225–246, 1995.

[CKBF03] Ugur Cetintemel, Peter J. Keleher, Bobby Bhattacharjee, und Michael J. Franklin. Deno: A Decentralized, Peer-to-Peer Object-Replication System for Weakly Connected Environments. *IEEE Transactions on Computers*, 52(7):943–959, 2003.

[CL99] Miguel Castro und Barbara Liskov. Practical Byzantine Fault Tolerance. In *Operating Systems Design and Implementation OSDI*, Seiten 173–186, 1999.

[CN01] Landon P. Cox und Brian D. Noble. Fast Reconciliations in Fluid Replication. In *ICDCS*, Seiten 449–458, 2001.

[Cod90] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison Wesley Publishing Company, 1990.

[Com08] JBoss Community. JBoss Application Server, 2008. http://www.jboss.org/. Besucht am 28.12.2008.

[Con97] S. Conrad. *Föderierte Datenbanksysteme*. Springer-Verlag, 1997.

[Con02] S. Conrad. Schemaintegration - Integrationskonflikte, Lösungsansätze, aktuelle Herausforderungen. *Informatik Forschung und Entwicklung*, 17(3):101–111, 2002.

[COR02] CORBA. The OMG's CORBA Website, 2002. http://www.corba.org. Besucht am 28.12.2008.

[Cri89] Flaviu Cristian. Probabilistic Clock Synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[Dad96] Peter Dadam. *Verteilte Datenbanken und Client/Server-Systeme*. Springer-Verlag, 1996.

[Dat03] C.J. Date. *An Introduction to Database Systems, Eighth Edition*. Addison Wesley, 2003.

[Dav84] Susan Davidson. Optimism and Consistency in Partitioned Distributed Database Systems. *ACM Transactions on Computer Systems*, 9(3):456–481, 1984.

[Dav89] Danco Davcev. A Dynamic Voting Scheme in Distributed Systems. *IEEE Trans. Software Eng.*, 15(1):93–97, 1989.

[DD97] C. J. Date und Hugh Darwen. *A Guide to SQL Standard (4th Edition)*. Addison-Wesley Professional, 1997.

[DE89] W. Du und A. K. Elmagarmid. Quasi-serializability: A correctness criterion on global concurrency control in Interbase. In *Very Large Data Bases Conference (VLDB)*, Seiten 347–355, 1989.

[DGH03] Shahram Dustdar, Harald Gall, und Manfred Hauswirth. *Software-Architekturen für Verteilte Systeme*. Springer, Berlin, 2003.

[DGMD85] Susan Davidson, Hector Garcia-Molina, und Skeen Dale. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, 1985.

[DHA03] Anwitaman Datta, Manfred Hauswirth, und Karl Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *ICDCS*, Seiten 76–. IEEE Computer Society, 2003.

[dLPC04] Cristian Pérez de Laborda, Christopher Popfinger, und Stefan Conrad. DIGAME: A Vision of an Active Multidatabase with Push-based Schema and Data Propagation. In Wilhelm Hasselbring und Manfred Reichert, Hrsg., *EAI*, Band 93 der Reihe *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[DLPR02] Wilhelm Dangelmaier, Hagen Lessing, Ulrich Pape, und Michael Rüther. Klassifikation von EAI-Systemen. *HMD - Praxis Wirtschaftsinform.*, 225, 2002.

[Dör03] Peter Dörsam. *Grundlagen der Entscheidungstheorie, anschaulich dargestellt*. Pd-Verlag, 2003.

[DSB88] Michel Dubois, Christoph Scheurich, und Faye A. Briggs. Synchronization, Coherence, and Event Ordering in Multiprocessors. *IEEE Computer*, 21(2):9–21, 1988.

[Ech90] Klaus Echtle. *Fehlertoleranzverfahren*. Springer, 1990.

[EGLT76] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, und Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, 1976.

[Elm86] Ahmed K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Rec.*, 15(3):37–45, 1986.

[EN02] Ramez A. Elmasri und Shamkant B. Navathe. *Grundlagen von Datenbanksystemen*. Pearson Studium, 2002.

[ES83] Derek L. Eager und Kenneth C. Sevcik. Achieving Robustness in Distributed Database Systems. *ACM Trans. Database Syst.*, 8(3):354–381, 1983.

[EW02] Franz Eisenfuehr und Martin Weber. *Rationales Entscheiden.* Springer, Berlin, 4. auflage Auflage, 2002.

[Fer05] George Fernandez. *A Federated Approach To Enterprise Integration.* Phd thesis, Faculty of Information and Communication Technologies, Swinburne University, 2005.

[FG01] Svend Frølund und Rachid Guerraoui. Implementing E-Transactions with Asynchronous Replication. *IEEE Trans. Parallel Distrib. Syst.*, 12(2):133–146, 2001.

[Fis88] Marek Fisz. *Wahrscheinlichkeitsrechnung und mathematische Stochastik.* VEB Deutscher Verlag der Wissenschaften, 1988.

[FL04] Rui Fan und Nancy Lynch. Gradient clock synchronization. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, Seiten 320–327, St. John's, Newfoundland, Canada. ACM Press, New York, NY, USA, 2004.

[Flo84] Christiane Floyd. A Systematic Look at Prototyping. In *Approaches to Prototyping*, Seiten 1–18. Springer-Verlag, 1984.

[FÖ89] Abdel Aziz Farrag und M. Tamer Özsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Trans. Database Syst.*, 14(4):503–525, 1989.

[For99] Paul Fortier. *SQL 3: Implmenting the SQL Foundation Standard.* McGraw-Hill Companies, 1999.

[Fro06] Markus Fromme. *Framework zur Simulation und Evaluation von Replikationsstrategien.* Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik, Abteilung Software Engineering, 2006.

[GA02] Sanny Gustavsson und Sten F. Andler. Self-stabilization and eventual consistency in replicated real-time databases. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, Seiten 105–107, Charleston, South Carolina. ACM Press, 2002.

[Gal99] Doreen L. Galli. *Distributed Operating Systems: Concepts and Practice.* Prentice Hall, 1999.

[GGH00] J. Grimson, W. Grimson, und W. Hasselbring. The System Integration Challenge in Healthcare. *Communications of the ACM*, 43(6):48–55, 2000.

[GGL03] Sanjay Ghemawat, Howard Gobioff, und Shun-Tak Leung. The Google file system. In Michael L. Scott und Larry L. Peterson, Hrsg., *SOSP*, Seiten 29–43. ACM, 2003.

[GHJ04] Erich Gamma, Richard Helm, und Ralph E. Johnson. *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software (Programmer's Choice).* Addison-Wesley, München, 2004.

[GHM+90] Richard G. Guy, John S. Heidemann, Wai-Kei Mak, Thomas W. Page Jr., Gerald J. Popek, und Dieter Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Summer*, Seiten 63–72, 1990.

[GHOS96] J. Gray, P. Helland, P. O'Neil, und D. Sasha. The Dangers of Replication and a Solution. In *ACM Sigmod Conference*, Seiten 173–182, 1996.

[Gif79] David K. Gifford. Weighted Voting for Replicated Data. In *ACM Symposium on Operating Systems Principles (SIGOPS)*, Seiten 150–159, 1979.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, und John L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*, Seiten 15–26, 1990.

[GLPT76] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, und Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, Seiten 365–394, 1976.

[GM83] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, 1983.

[GMAB+83] Hector Garcia-Molina, Tim Allen, Barbara T. Blaustein, R. Mark Chilenskas, und Daniel R. Ries. Data-Patch: Integrating Inconsistent Copies of a Database After a Partition. In *Symposium on Reliability in Distributed Software and Database Systems*, Seiten 38–44, 1983.

[GMB85] Hector Garcia-Molina und Daniel Barbara. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985.

[GMJ90] Hector Garcia-Molina und H. V. Jagadish, Hrsg. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.* ACM Press, 1990.

[GMS87] Hector Garcia-Molina und Kenneth Salem. Sagas. In Umeshwar Dayal und Irving L. Traiger, Hrsg., *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, Seiten 249–259. ACM Press, 1987.

[GMW82] Hector Garcia-Molina und Gio Wiederhold. Read-Only Transactions in a Distributed Database. *ACM Trans. Database Syst.*, 7(2):209–234, 1982.

[Gor00] Ian Gorton. *Enterprise Transaction Processing Systems: Putting the CORBA OTS, Encina++ and Orbix OTM to Work.* Addison-Wesley Pub (Sd), 2000.

[GR93] Jim Gray und Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[Gra78] J. Gray. Notes on Database Operating Systems. In R. Bayer, R. M. Graham, und G. Seegmüller, Hrsg., *Operating Systems: an Advanced Course*, Seiten 393–481. Springer Lecture Notes in Computer Science, Heidelberg, 1978.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *VLDB*, Seiten 144–154, 1981.

[Gra92] Timm Grams. *Simulation - Strukturiert und objektorientiert programmiert.* BI Wissenschaftsverlag, 1992.

[GRG+98] Richard Guy, Peter Reiher, Michial Gunter, Wilkie Ma, und Gerald Popek. Rumor: Mobile Data Access Through Optimistic Peer-To-Peer Replication. In *International Conference on Conceptual Modeling (ER)*, Seiten 254–265, 1998.

[Gri98] Frank Griffel. *Componentware. Konzepte und Techniken eines Softwareparadigmas.* Dpunkt Verlag, 1998.

[Gro01] William Grosso. *Java RMI.* O'Reilly, 2001.

[GZ89] Riccardo Gusella und Stefano Zatti. The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Trans. Software Eng.*, 15(7):847–853, 1989.

[HA90] Phillip W. Hutto und Mustaque Ahamad. Slow Memory: Weakening Consistency to Enchance Concurrency in Distributed Shared Memories. In *ICDCS*, Seiten 302–309. IEEE Computer Society, 1990.

[HAA99] JoAnne Holliday, Divyakant Agrawal, und Amr El Abbadi. Database Replication: If You Must be Lazy, be Consistent. In *Symposium on Reliable Distributed Systems*, Seiten 304–305, 1999.

[Has97] Wilhelm Hasselbring. Federated integration of replicated information within hospitals. *International Journal on Digital Libraries*, 1(3):192–208, 1997.

[Has00] Wilhelm Hasselbring. Information System Integration. *Communications of the ACM*, 43(6):32–36, 2000.

[HCH04] Jianchao Han, Nick Cercone, und Xiaohua Hu. A Weighted Freshness Metric for Maintaining Search Engine Local Repository. In *International Conference on Web Intelligence*, Seiten 677–680, Beijing, China. IEEE Computer Society, 2004.

[Hül02] Michael Hülsmann. *Entwicklung eines adaptiven Replikationsmanagers auf Basis von EJB-JTS/-JMS*. Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik, Abteilung Software Engineering, 2002.

[HLUA02] Uwe Herrmann, Dierk Lenz, Günter Unbescheid, und Johannes Ahrends. *Oracle9i für den DBA . Effizient konfigurieren, optimieren und verwalten*. Addison-Wesley, 2002.

[HR83] Theo Härder und Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[HR85] Frederick Hayes-Roth. Rule-Based Systems. *Commun. ACM*, 28(9):921–932, 1985.

[HR01] Theo Härder und Erhard Rahm. *Datenbanksysteme*. Springer, Berlin, 2001.

[HS00] Andreas Heuer und Gunter Saake. *Datenbanken Konzepte und Sprachen*. mitp, 2000.

[HT94] Vassos Hadzilacos und Sam Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technischer Bericht TR94-1425, Cornell University, Ithaca, NY, USA, 1994.

[JBC+06] Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, und Kim Haase. *Java(TM) EE 5 Tutorial, The (3rd Edition) (The Java Series)*. Prentice Hall PTR, 2006.

[JM87] Sushil Jajodia und David Mutchler. Dynamic Voting. In Umeshwar Dayal und Irving L. Traiger, Hrsg., *SIGMOD Conference*, Seiten 227–238. ACM Press, 1987.

[JM90] Sushil Jajodia und David Mutchler. Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database. *ACM Trans. Database Syst.*, 15(2):230–280, 1990.

[JRW90] Stefan Jablonski, Thomas Ruf, und Hartmut Wedekind. Implementation of a distributed data management system for technical applications-a feasibility study. *Inf. Syst.*, 15(2):247–256, 1990.

[Kal07] Nicolai Kalisch. *Simulation von Konfliktmanagementstrategien für die asynchrone Replikation*. Individuelles Projekt, Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik, Abteilung Software Engineering, 2007.

[KB91] Narayanan Krishnakumar und Arthur J. Bernstein. Bounded Ignorance in Replicated Systems. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 29-31, 1991, Denver, Colorado*, Seiten 63–74. ACM Press, 1991.

[KK06] Martin Kreutzer und Stefan Kühling. *Logik für Informatiker*. Pearson Studium, 2006.

[KKB88] Henry F. Korth, Won Kim, und François Bancilhon. On Long-Duration CAD Transactions. *Inf. Sci.*, 46(1-2):73–107, 1988.

[KL03] Juyoung Kang und Jae Kyu Lee. Extraction of Structured Rules from Web Pages and Maintenance of Mutual Consistency: XRML Approach. In Michael Schroeder und Gerd Wagner, Hrsg., *Rules and Rule Markup Languages for the Semantic Web, Second International Workshop*, Band 2876 der Reihe *Lecture Notes in Computer Science*, Seiten 150–163. Springer, 2003.

[KLS90] Henry F. Korth, Eliezer Levy, und Abraham Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In Dennis McLeod, Ron Sacks-Davis, und Hans-Jörg Schek, Hrsg., *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, Seiten 95–106. Morgan Kaufmann, 1990.

[Klu01] Niels Klussmann. *Lexikon der Kommunikations- und Informationstechnik*. Hüthig Telekommunikation, 2001.

[Kor83] Henry F. Korth. Locking Primitives in a Database System. *J. ACM*, 30(1):55–79, 1983.

[KR81] H. T. Kung und John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[Krü00] Guido Krüger. *Go To Java 2*. Addison Wesley, 2000.

[KS88] Henry F. Korth und Gregory D. Speegle. Formal Model of Correctness Without Serializability. In Haran Boral und Per-Åke Larson, Hrsg., *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, Seiten 379–386. ACM Press, 1988.

[KSSA02] Kyoung-Don Kang, Sang Hyuk Son, John A. Stankovic, und Tarek F. Abdelzaher. A QoS-Sensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases. In *14th Euromicro Conference on Real-Time Systems ECRTS*, Seiten 203–212, Vienna, Austria. IEEE Computer Society, 2002.

[Kum91] Akhil Kumar. Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data. *IEEE Trans. Computers*, 40(9):996–1004, 1991.

[LA90] Peter A. Lee und Thomas Anderson. *Fault Tolerance: Principles and Practice*. Springer, Secaucus, NJ, USA, 2. Auflage, 1990.

[LA00] Thanasis Loukopoulos und Ishfaq Ahmad. Static and Adaptive Data Replication Algorithms for Fast Information Access in Large Distributed Systems. In *ICDCS*, Seiten 385–392, 2000.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[Lam94] Winfried Lamersdorf. *Datenbanken in verteilten Systemen.* Vieweg, 1994.

[Len96] Richard Lenz. Adaptive distributed data management with weak consistent replicated data. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, Seiten 178–185, Philadelphia, Pennsylvania, United States. ACM Press, New York, NY, USA, 1996.

[Len97] Richard Lenz. *Adaptive Datenreplikation in verteilten Systemen.* Teubner, Leipzig, 1997.

[Lev03] Gertrude Neuman Levine. Defining deadlock. *SIGOPS Oper. Syst. Rev.*, 37(1):54–64, 2003.

[Ley03] Frank Leymann. Web Services: Distributed Applications Without Limits. In Gerhard Weikum, Harald Schöning, und Erhard Rahm, Hrsg., *BTW*, Band 26 der Reihe *LNI*, Seiten 2–23. GI, 2003.

[LH00] A. Lubinski und A. Heuer. Configured Replication for Mobile Applications. In *Proc. of Baltic DB & IS*, Seiten 1–5, 2000.

[Lin99] David S. Linthicum. *Enterprise Application Integration.* Addison-Wesley, 1999.

[Lis91] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, Seiten 1–9, Montreal, Quebec, Canada. ACM Press, New York, NY, USA, 1991.

[LK91] Averill M. Law und W. David Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, 2. Auflage, 1991.

[LLS90] Rivka Ladin, Barbara Liskov, und Liuba Shrira. Lazy Replication: Exploiting the Semantics of Distributed Services. In *Workshop on the Management of Replicated Data*, Seiten 31–34, 1990.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira, und Sanjay Ghemawat. Providing High Availability Using Lazy Replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, 1992.

[LMP04] Mark Little, Jon Maron, und Greg Pavlik. *Java Transaction Processing: Design and Implementation.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[LPH99] Matthias Lange, Hans-Ulrich Prokosch, und Wilhelm Hasselbring. Eine Taxonomie für Kommunikationsserver im Krankenhaus. *Informatik, Biometrie und Epidemiologie in Medizin und Biologie*, 30(1):21–43, 1999.

[LPST95] Christoph Liebig, Hans-Henning Pagnia, Frank Schwappacher, und Oliver Theel. Applying Scalable Read Consistency to Data Replication. In *Proc. of the 13th International Conference on Applied Informatics*, IASTED, February 1995.

[LPST96] Christoph Liebig, Hans-Henning Pagnia, Frank Schwappacher, und Oliver Theel. A Quality-of-Service Approach for Mobile Users of Replicated Data in Distributed Systems. In *Proc. of the 10th European Simulation Multiconference, ESM96, Budapest, Hungary*, Seiten 842–851, SCS, June 1996.

[LR03] Alexandros Labrinidis und Nick Roussopoulos. Balancing Performance and Data Freshness in Web Database Servers. In *VLDB*, Seiten 393–404, 2003.

[LS88] Richard Lipton und Jonathan Sandberg. PRAM: A Scalable Shared Memory. Technischer Bericht CS-TR-180-88, Princeton University, 1988.

[LS03] Jae Kyu Lee und Mye M. Sohn. The extensible rule markup language. *Communications of the ACM*, 46(5):59–64, 2003.

[LSP82] Leslie Lamport, Robert E. Shostak, und Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[MA02] Daniel Menasce und Virgilio Almeida. *Capacity Planning for Web Services*. Prentice Hall, Upper Saddle River, 2002.

[Mat89] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In Cosnard M. et al., Hrsg., *Proc. Workshop on Parallel and Distributed Algorithms*, Seiten 215–226, 1989.

[MBK00] Didier Martin, Mark Birbeck, und Michael Kay. *XML professionell*. mitp, 2000.

[Men05] Daniel A. Menasce. MOM vs. RPC: Communication Models for Distributed Applications. *IEEE Internet Computing*, 9(2):90–93, 2005.

[MH00] Jeremy M. R. Martin und Yvonne Huddart. Parallel Algorithms for Deadlock and Livelock Analysis of Concurrent Systems. In Prof. Peter H. Welch und Andrè W. P. Bakkers, Hrsg., *Communicating Process Architectures 2000*, Seiten 1–14. IOS Press, 2000.

[MHC01] Richard Monson-Haefel und David A. Chappel. *Java Message Service*. O'Reilly, 2001.

[Mil03] David L. Mills. A brief history of NTP time: memoirs of an Internet timekeeper. *SIGCOMM Comput. Commun. Rev.*, 33(2):9–21, 2003.

[MNSO04] M.D. Mustafa, B. Nathrah, M.H. Suzuri, und M.T. Abu Osman. Improving Data Availability Using Hybrid Replication Technique in Peer-to-Peer Environments. In *Proc. 18th International Conference on Advanced Information Networking and Applications (AINA'04)*, Seiten 593–598, Fukuoka, Japan. IEEE CS Press, 2004.

[Mos85] J. Eliot B. Moss. *Nested Transactions : An Approach to Reliable Distributed Computing (Digital Communication)*. The MIT Press, 1985.

[Mos93] David Mosberger. Memory Consistency Models. *Operating Systems Review*, 27(1):18–26, 1993.

[MR98] Dahlia Malkhi und Michael K. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, 1998.

[MRKS98] Sharad Mehrotra, Rajeev Rastogi, Henry F. Korth, und Abraham Silberschatz. Ensuring consistency in multidatabases by preserving two-level serializability. *ACM Trans. Database Syst.*, 23(2):199–230, 1998.

[MTJ+08] John MacCormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, und Ryan Peterson. Niobe: A practical replication protocol. *TOS*, 3(4), 2008.

[Mul93] Sape Mullender. *Distributed Systems (2nd Edition)*. Addison Wesley Publishing Company, 1993.

[MW82] Toshimi Minoura und Gio Wiederhold. Resilient Extended True-Copy Token Scheme for a Distributed Database System. *IEEE Trans. Software Eng.*, 8(3):173–189, 1982.

[Neu94] B. Clifford Neuman. Scale in distributed systems. In Thomas L. Casavant und Mukesh Singhal, Hrsg., *Readings in Distributed Computing Systems*, Seiten 463–489. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[NH02] Heiko Niemann und Wilhelm Hasselbring. Datenreplikationstechniken für stationäre und mobile Informationssyteme im Krankenhaus. In *GI Jahrestagung*, Band 19 der Reihe *LNI*, Seiten 604–608. GI, 2002.

[NHHT03] Heiko Niemann, Wilhelm Hasselbring, Michael Hülsmann, und Oliver Theel. Realisierung eines adaptiven Replikationsmanagers auf Basis der J2EE-Technologie. In *Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, Band 26 der Reihe *LNI*, Seiten 443–452. GI, Leipzig, 2003.

[NHW+02] Heiko Niemann, Willi Hasselbring, Thomas Wendt, Alfred Winter, und Matthias Meierhofer. Kopplungsstrategien für Anwendungssysteme im Krankenhaus. *Wirtschaftsinformatik*, 44(5):425–434, 2002.

[OFG07] Johannes Osrael, Lorenz Froihofer, und Karl M. Göschka. Availability/Consistency Balancing Replication Model. In *IPDPS*, Seiten 1–8. IEEE, 2007.

[OFGG06] Johannes Osrael, Lorenz Froihofer, Matthias Gladt, und Karl M. Göschka. Adaptive Voting for Balancing Data Integrity with Availability. In Robert Meersman, Zahir Tari, und Pilar Herrero, Hrsg., *OTM Workshops (2)*, Band 4278 der Reihe *Lecture Notes in Computer Science*, Seiten 1510–1519. Springer, 2006.

[OMG06] Object Management Group OMG. UML 2.0 Superstructure Specification (formal/05-07-04), 2006. http://www.omg.org/technology/documents/formal/uml.htm. Besucht am 16.8.2006.

[Ope03] OpenGroup. Transaction Processing, 2003. http://www.opengroup.org/products/publications/catalog/tp.htm. Besucht am 3.10.2003.

[Ozs99] Ozsu. *The Principles of Distributed Databases (International Edition) (Prentice Hall International Editions)*. Pearson US Imports & PHIPEs, 1999.

[Pap86] Christos Papadimitriou. *Theory of Database Concurrency Control (Principles of computer science series)*. Computer Science Pr, 1986.

[Par89] Jehan-François Paris. Voting with Bystanders. In *ICDCS*, Seiten 394–405, 1989.

[Par90] Jehan-François Paris. Efficient voting protocols with witnesses. In *ICDT '90: Proceedings of the third international conference on database theory on Database theory*, Seiten 305–317, Paris, France. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[PB99] Evaggelia Pitoura und Bharat K. Bhargava. Data Consistency in Intermittently Connected Distributed Systems. *IEEE Trans. Knowl. Data Eng.*, 11(6):896–915, 1999.

[Pei87] Peter Peinl. *Synchronisation in zentralisierten Datenbanksystemen.* Springer, Heidelberg, 1987.

[PG06] Cécile Le Pape und Stéphane Gançarski. Replica Refresh Strategies in a Database Cluster. In Michel J. Daydé, José M. L. M. Palma, Alvaro L. G. A. Coutinho, Esther Pacitti, und João Correia Lopes, Hrsg., *VECPAR*, Band 4395 der Reihe *Lecture Notes in Computer Science*, Seiten 679–691. Springer, 2006.

[PGV04] Cécile Le Pape, Stéphane Gançarski, und Patrick Valduriez. Refresco: Improving Query Performance Through Freshness Control in a Database Cluster. In *CoopIS/DOA/ODBASE (1)*, Band 3290 der Reihe *Lecture Notes in Computer Science*, Seiten 174–193. Springer, 2004.

[PL90] Calton Pu und Avraham Leff. Epsilon-serialisability. Technischer Bericht CUCS-054-90, Columbia University, 1990.

[PL91] Calton Pu und Avraham Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In James Clifford und Roger King, Hrsg., *International Conference on Management of Data*, Seiten 377–386. ACM SIGMOD, Denver, Colorado, USA, 1991.

[PL92] Calton Pu und Avraham Leff. Autonomous Transaction Execution with Epsilon Serializability. In *RIDE-TQP*, Seiten 2–11, 1992.

[PPR83] D. S. Parker, Gerald Popek, und G. Rudisin. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, 1983.

[PS97] Evaggelia Pitoura und George Samaras. *Data Management for Mobile Computing (Kluwer International Series on Advances in Database Systems)*. Kluwer Academic Publishers, 1997.

[PS00] Esther Pacitti und Eric Simon. Update Propagation Strategies to Improve Freshness in Lazy Master Replicated Databases. *VLDB J.*, 8(3-4):305–318, 2000.

[PTM97] Jelica Protic, Milo Tomasevic, und Veljko Milutinovic. *Distributed Shared Memory: Concepts and Systems.* Wiley-IEEE Computer Society Pr, 1997.

[Rah93] E. Rahm. *Hochleistungs-Transaktionssysteme.* vieweg, 1993.

[Rah94] E. Rahm. *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung.* R. Oldenbourg Verlag, 1994.

[Ram93] Krithi Ramamritham. Real-Time Databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[RBSS02] U. Röhm, K. Böhm, H.-J. Schek, und H. Schuldt. FAS - A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. In *Very Large Data Bases Conference (VLDB)*, Seiten 754–765, 2002.

[RC96] Krithi Ramamritham und Panos K. Chrysanthis. A Taxonomy of Correctness Criteria in Database Applications. *VLDB J.*, 5(1):85–97, 1996.

[Reu87] Andreas Reuter. Maßnahmen zur Wahrung von Sicherheits- und Integritäsbedingungen. In Peter C. Lockemann und Joachim W. Schmidt, Hrsg., *Datenbankhandbuch.*, Seiten 337–479. Springer, 1987.

[RF02] Matei Ripeanu und Ian T. Foster. A Decentralized, Adaptive Replica Location Mechanism. In *HPDC*, Seiten 24–32. IEEE Computer Society, 2002.

[RHQ+05] Chris Rupp, Jürgen Hahn, Stefan Queins, Mario Jeckle, und Barbara Zengler. *UML 2 glasklar. Praxiswissen für die UML -Modellierung und Zertifizierung*. Hanser Fachbuchverlag, 2. Auflage, 2005.

[RJB04] James Rumbaugh, Ivar Jacobson, und Grady Booch. *Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley Professional, 2004.

[RL92] Michael Rabinovich und Edward D. Lazowska. Improving Fault Tolerance and Supporting Partial Writes in Structured Coterie Protocols for Replicated Objects. In Michael Stonebraker, Hrsg., *SIGMOD Conference*, Seiten 226–235. ACM Press, 1992.

[RL93] Michael Rabinovich und Edward D. Lazowska. Asynchronous Epoch Management in Replicated Databases. In *Distributed Algorithms, 7th International Workshop, WDAG*, Band 725 der Reihe *Lecture Notes in Computer Science*, Seiten 115–128, Lausanne, Switzerland. Springer, 1993.

[RL03] Maya Rodrig und Anthony LaMarca. Decentralized weighted voting for P2P data management. In *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access (MobiDe'03)*, Seiten 85–92, San Diego, CA, USA. ACM Press, 2003.

[RLPT97] Peter Reichl, Claudia Linnhoff-Popien, und Dirk Thissen. Einbeziehung von Nutzerinteressen bei der QoS-basierten Dienstvermittlung unter CORBA. In Martina Zitterbart, Hrsg., *Kommunikation in Verteilten Systemen (KiVS)*, Seiten 236–251. Springer, Braunschweig, 1997.

[RMB+93] Rajeev Rastogi, Sharad Mehrotra, Yuri Breitbart, Henry F. Korth, und Abraham Silberschatz. On Correctness of Non-serializable Executions. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC*, Seiten 97–108. ACM Press, 1993.

[RMB00] William Ruh, Francis Maginnis, und William Brown. *Enterprise Application Integration: A Wiley Tech Brief*. John Wiley and Sons, 2000.

[Rob03] Ian Robinson. J2EE Activity Service Specification, JSR095, Proposed Final Draft, 2003. http://jcp.org/en/jsr/detail?id=95. Besucht am 28.12.2008.

[RRP99] David Ratner, Peter L. Reiher, und Gerald J. Popek. Roam: A Scalable Replication System for Mobile Computing. In *DEXA Workshop*, Seiten 96–104, 1999.

[RSK91] Marek Rusinkiewicz, Amit P. Sheth, und George Karabatis. Specifying Interdatabase Dependencies in a Multidatabase Environment. *IEEE Computer*, 24(12):46–53, 1991.

[Rul08] JBoss Rules. Rules Framework, 2008. http://jboss.com/products/rules. Besucht am 28.12.2008.

[Sal79] Arto Salomaa. *Formale Sprachen*. Springer-Verlag, 1979.

[Sat02] Mahadev Satyanarayanan. The Evolution of CODA. *ACM Transactions on Computer Systems*, 20(2):85–124, 2002.

[Sch90] H. Schöning. Preserving Consistency in Nested Transactions. In *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences*, Band II, Seiten 472–480. IEEE Computer Society Press, 1990.

[Sch98] August-Wilhelm Scheer. *Wirtschaftsinformatik. Studienausgabe. Referenzmodelle für industrielle Geschäftsprozesse*. Springer-Verlag, 1998.

[Sch01a] Rüdiger Schollmeier. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In Ross Lee Graham und Nahid Shahmehri, Hrsg., *Peer-to-Peer Computing*, Seiten 101–102. IEEE Computer Society, 2001.

[Sch01b] Uwe Schöning. *Theoretische Informatik - kurzgefasst. (Spektrum Hochschultaschenbücher)*. Spektrum Akademischer Verlag, 2001.

[SGM02] Clemens Szyperski, Dominik Gruntz, und Stephen Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2. Auflage, 2002.

[SH99] Gunter Saake und Andreas Heuer. *Datenbanken: Implementierungstechniken*. mitp, 1999.

[Sno99] Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, 1999.

[Spe82] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Commun. ACM*, 25(4):246–260, 1982.

[SR90] Amit P. Sheth und Marek Rusinkiewicz. Management of Interdependent Data: Specifying Dependency and Consistency Requirements. In *Workshop on the Management of Replicated Data*, Seiten 133–136, 1990.

[SS83a] Gunter Schlageter und Wolffried Stucky. *Datenbanksysteme, Konzepte und Modelle*. Teubner, 1983.

[SS83b] Richard D. Schlichting und Fred B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983.

[SS00] G. Saake und Kai-Uwe Sattler. *Datenbanken und Java. JDBC, SQLJ und ODMG*. dpunkt-Verlag, 2000.

[SS03] Alexander Schwinn und Joachim Schelp. Data Integration Patterns. In *6th International Conference on Business Information Systems BIS*, Seiten 232–238. Department of Management Information Systems at The Poznan University of Economics (Poznan, Poland), 2003.

[SS05] Yasushi Saito und Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[SSN02] Rahul Sharma, Beth Stearns, und Tony Ng. *J2EE Connector Architecture and Enterprise Application Integration*. Addison Wesley, 2002.

[ST06] Christian Storm und Oliver Theel. Highly Adaptable Dynamic Quorum Schemes for Managing Replicated Data. In *Conference on Availability, Reliability and Security, ARES*, Seiten 245–253, 2006.

[Sto79] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.

[SW85] Dieter Steinbauer und Hartmut Wedekind. Integritätsaspekte in Datenbanksystemen. *Informatik Spektrum*, 8(2):60–68, 1985.

[SW04a] R. Steinmetz und K. Wehrle. Peer-to-Peer-Networking und -Computing. *Informatik-Spektrum*, 27(1):51–54, Februar 2004.

[SW04b] Ulrike Stocker und Karl-Heinz Waldmann. *Stochastische Modelle: Eine anwendungsorientierte Einführung*. Springer, Berlin, 2004.

[SWL90] B. Simons, J. L. Welch, und N. Lynch. An overview of clock synchronization. *LNCS, Fault-tolerant distributed computing*, Seiten 84–96, 1990.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[TDP+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, und Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Parallel and Distributed Information Systems PDIS*, Seiten 140–149. IEEE Computer Society, Austin, Texas, 1994.

[TGGL82] I. L. Traiger, J. Gray, C. A. Galthier, und B. G. Lindsay. Transactions and consistency in distributed database systems. *ACM Transactions on Database Systems*, 7(3):323–342, 1982.

[The93a] Oliver Theel. *Ein vereinheitlichendes Konzept zur Konstruktion hochverfügbarer Dienste (in German)*. Ph.d. dissertation, Dept. of Computer Science, University of Darmstadt, Germany, Juli 1993.

[The93b] Oliver E. Theel. General Structured Voting: A Flexible Framework for Modelling Cooperations. In *ICDCS*, Seiten 227–236, 1993.

[Tho79] Robert H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[TPST98] Douglas B. Terry, Karin Petersen, Mike Spreitzer, und Marvin Theimer. The Case for Non-transparent Replication: Examples from Bayou. *IEEE Data Eng. Bull.*, 21(4):12–20, 1998.

[TR85] Andrew S. Tanenbaum und Robbert Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17(4):419–470, 1985.

[Tri01] Kishor Shridharbhai Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. John Wiley & Sons, New York, 2nd Auflage, 2001.

[TS99] Oliver E. Theel und Thomas Strauß. An Excursion to the Zoo of Dynamic Coterie-Based Replication Schemes. In *ICPP*, Seiten 344–, 1999.

[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, und Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Symposium on Operating Systems Principles*, Seiten 172–183, 1995.

[TvS02] Andrew S. Tanenbaum und Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[TW01] Kuldar Taveter und Gerd Wagner. Agent-Oriented Enterprise Modeling Based on Business Rules. In Hideko S. Kunii, Sushil Jajodia, und Arne Sølvberg, Hrsg., *ER*, Band 2224 der Reihe *Lecture Notes in Computer Science*, Seiten 527–540. Springer, 2001.

[vdV02] Eric van der Vlist. *XML Schema*. O'Reilly & Associates, Inc., Cambridge, 1st Auflage, June 2002.

[VR01] Paulo Verissimo und Luis Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[vRS04] Robbert van Renesse und Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, Seiten 91–104, 2004.

[vRT88] Robbert van Renesse und Andrew S. Tanenbaum. Voting with Ghosts. In *ICDCS*, Seiten 456–462, 1988.

[War05] Timo Warns. *Replication for Peer-to-Peer Systems to Improve Dependability*. Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik, Abteilung Software Engineering, 2005.

[WC95] Jennifer Widom und Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc,US, 1995.

[Web98] M. Weber. *Verteilte Systeme*. Spektrum Akademischer Verlag, 1998.

[Wee03] Michael Weers. *UML-basierte Abbildungssprache zwischen Datenschemata*. Diplomarbeit (in arbeit), Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik, Abteilung Software Engineering, 2003.

[Wei88] William E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Trans. Computers*, 37(12):1488–1505, 1988.

[Wei91] Gerhard Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Trans. Database Syst.*, 16(1):132–180, 1991.

[WJH97] Ouri Wolfson, Sushil Jajodia, und Yixiu Huang. An Adaptive Data Replication Algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, 1997.

[Wol02] Thomas Wolf. *Benchmark für EJB-Transaction und Message-Service*. Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik, Abteilung Software Engineering, 2002.

[WPE+83] Bruce J. Walker, Gerald J. Popek, Robert English, Charles S. Kline, und Greg Thiel. The LOCUS Distributed Operating System. In *SOSP*, Seiten 49–70, 1983.

[WQ87] Gio Wiederhold und Xiaolei Qian. Modeling Asynchrony in Distributed Databases. In *Proceedings of the Third International Conference on Data Engineering*, Seiten 246–250. IEEE Computer Society, Washington, DC, USA, 1987.

[WQ90] Gio Wiederhold und Xiaolei Qian. Consistency Control of Replicated Data in Federated Databases. In *Workshop on the Management of Replicated Data*, Seiten 130–132, 1990.

[WR92] Helmut Wächter und Andreas Reuter. The ConTract model. In Ahmed K. Elmagarmid, Hrsg., *Database Transaction Models for Advanced Applications*, Seiten 219–263. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[WSP+00] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, und Gustavo Alonso. Database Replication Techniques: A Three Parameter Classification. In *SRDS*, Seiten 206–215, 2000.

[WV01] Gerhard Weikum und Gottfried Vossen. *Transactional Information Systems : Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[YKC99] Sang B. Yoo, K. C. Kim, und Sang Kyun Cha. A Middleware Implementation of Active Rules for ODBMS. In Arbee L. P. Chen und Frederick H. Lochovsky, Hrsg., *Database Systems for Advanced Applications, Proceedings of the Sixth International Conference on Database Systems for Advanced Applications (DASFAA), April 19-21, Hsinchu, Taiwan*, Seiten 347–354. IEEE Computer Society, 1999.

[YV02] Haifeng Yu und Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.

[ZJ98] Hengming Zou und Farnam Jahanian. Real-Time Primary-Backup (RTPB) Replication with Temporal Consistency Guarantees. In *ICDCS*, Seiten 48–56, 1998.

[ZSJ00] Hengming Zou, Nandit Soparkar, und Farnam Jahanian. Probabilistic Data Consistency for Wide-Area Applications. In *ICDE*, Seite 85, 2000.

[ZZ03] Chi Zhang und Zheng Zhang. Trading Replication Consistency for Performance and Availability: an Adaptive Approach. In *23rd International Conference on Distributed Computing Systems (ICDCS)*, Seiten 687–695. IEEE Computer Society, 2003.