

Prüfer: Prof. Dr. Rothermel

Betreuerin: Dr. Cora Burger

Begonnen am: 01.08.2001

Beendet am: 31.01.2002

CR-Nummer: H.4.1, I.2.1, I.2.11, I.7.1

Diplomarbeit Nr. 1956

**Konsistenzerhaltung mit
mobilen Agenten bei
gemeinsamer Dokumentenbearbeitung**

Georgios Tzizlis

Inhaltsverzeichnis

Inhaltsverzeichnis.....	ii
Abkürzungsverzeichnis	iv
Abbildungs- und Tabellenverzeichnis	v
1. Einführung	1
1.1 Motivation	1
1.2 Aufgabenstellung	2
1.3 Überblick.....	2
2. Grundlagen.....	4
2.1 Gemeinsame Dokumentenerstellung.....	4
2.1.1 Beschreibung von Dokumenten	4
2.1.2 Das Team	5
2.2 Konsistenz als Ergebnis von Regeleinhaltung.....	6
2.2.1 Konsistenz-Arten von Dokumenten	6
2.2.2 Besonderheiten einer DDE (Distributed Document Environment).....	7
3. Agententechnologie.....	10
3.1 Definitionen.....	10
3.2 Multiagenten-System	11
3.3 Mobile Agentensysteme	12
3.4 Eigenschaften.....	12
3.5 Bewertung	13
3.6 Das Mole-System	14
3.6.1 Die Struktur von Mole.....	14
3.6.2 Mole Lokationen und Agenten	15
3.6.3 Agentenkommunikation	15
4. Das Mole-DDE-System.....	16
4.1 Überblick über die Struktur des DDE-Systems	16
4.2 Verwendete Dokumentenstruktur	17
4.3 Informationsaustausch.....	18
4.4 Aktionen.....	18
4.5 Editor	19
4.6 Wrapper.....	19
4.7 Schnittstelle zwischen Editor und Mole-DDE	20
4.8 Grafische Benutzungsoberfläche.....	22
5. Aspekte eines Konsistenzerhaltungssystems (KES).....	23
5.1 Transparenzaspekte	23
5.1.1 Anforderungen mobiler Benutzer und Replikate	23
5.1.2 Gewollte und ungewollte Trennung vom System.....	25
5.1.3 'Single System Image' und Transparenz.....	25
5.1.4 Anwendungs-transparent oder 'Application-aware'	26
5.2 Konsistenzerhaltungsverfahren	26
5.2.1 Replikationsstrategien	27
5.2.2 Konflikterkennung.....	28
5.3 Reintegration	31
5.3.1 An Reintegration beteiligte Replikate	31

5.3.2 Beispielszenario	32
5.3.3 Zeitpunkt der Reintegration	34
5.3.4 Benutzerbeteiligung bei Reintegration	34
5.4 Zusammenfassung	35
6. Anforderungsanalyse und Spezifikation	36
6.1 Ist-Zustand.....	36
6.2 Soll-Zustand	37
6.2.1 Anforderungen aus der Sicht der Software-Qualität	37
6.2.2 Anforderungen aus funktioneller Sicht.....	38
6.2.3 Spezifikation	39
7. Entwurf	41
7.1 Dokumentenstruktur und die auf ihr erlaubten Operationen	41
7.2 Von Änderungsoperationen verursachte Konfliktfälle	42
7.2.1 Komponentenebene Buch-Kapitel	43
7.2.2 Komponentenebene Absatz	44
7.3 Eine Diskussion über Konsistenzregeln.....	45
7.4 Das REVISE-System.....	46
7.4.1 Erweitertes logisches Programm.....	46
7.4.2 Das REVISE-System als Teil eines Konfliktlösungsmechanismus.....	48
7.4.3 Möglichkeiten, das REVISE-System in die Mole-DDE zu integrieren	49
7.5 Kooperierende und argumentierende Agenten.....	50
7.5.1 Argumentation in der Philosophie.....	50
7.5.2 Argumentation eines Agenten.....	51
7.5.3 Multiagenten-Argumentation.....	52
7.6 Unterstützung für entkoppelte Arbeit	53
8. Feinentwurf.....	55
8.1 Randbedingungen	55
8.2 Struktur des MOLE-KES.....	57
8.3 Reintegration	60
8.3.1 Konsistenzagenten als MAS	61
8.3.2 Interaktion der Konsistenzagenten mit dem REVISE-System	63
9. Implementierung	66
9.1 Konsistenzagent.....	66
9.2 Das Konsens-Modul	68
9.3 Schnittstelle zwischen Java und SICStus-Prolog.....	69
9.3.1 Jasper zum Laufen bringen	69
9.3.2 Prolog von Java aus aufrufen.....	70
10. Testszenarien.....	73
11. Zusammenfassung und Bewertung.....	75
12. Ausblick	76
Literaturverzeichnis	77

Abkürzungsverzeichnis

AFS	A ndrew F ile S ystem
API	A pplication P rograming I nterface
BDI	B elief D esire I ntention
CORBA	C ommon O bject R equest B roker A rchitecture
CSCW	C omputer- S upported C ollaborative W ork
DDE	D istributed D ocument E nvironment
IE	I nformation S Einheit
IPVR	I nstitut für P arallele und V erteilte H öchstleistungs R echner
JDK	J ava D evelopment K it
JVM	J ava V irtual M achine
KES	K onsistenz E rhaltung S ystem
MAS	M ulti A genten S ystem
NFS	N etwork F ile S ystem
NMR	N on M onotonic R easoning
RMI	R emote M ethod I nvocation
RPC	R emote P rocedure C all
SICS	S wedish I nstitut for C omputer S cience
WFSX	W ell F ounded S emantics with eX plicit negation

Abbildungs- und Tabellenverzeichnis

Abbildung 2.1: Granularität der Struktur eines Dokumentes	5
Abbildung 2.2: Gekoppelte/Entkoppelte Arbeit.....	7
Abbildung 2.3: Caching	8
Abbildung 2.4: Entstehung und Behebung von Inkonsistenzen	9
Abbildung 3.1: Umgebung von Agenten.....	11
Abbildung 3.2: Darstellung von Objekten und Agenten.....	13
Abbildung 3.3: Das Mole System	14
Abbildung 4.1: Komponenten der Mole-DDE	16
Abbildung 4.2: hierarchische Dokumentenstruktur der Mole-DDE	17
Abbildung 4.3: Definition einer IE in BNF	21
Abbildung 5.1: Replikationsstrategien	27
Abbildung 5.2: Konflikterkennung durch Version-Vectors	29
Abbildung 5.3: Historie-Aufzeichnung	30
Abbildung 5.4: Client-Server Reintegration	32
Abbildung 5.5: Peer-To-Peer Reintegration	33
Tabelle 7.1: Konfliktfälle bei Änderungsoperationen auf Dokumentkomponenten	42
Abbildung 7.1: Ein kleines erweitertes logisches Programm	47
Abbildung 7.2: Komponenten des Konfliktlösungsmechanismus	48
Abbildung 8.1: Mole-DDE Agentenstruktur.....	56
Abbildung 8.2: Agentenstruktur der Mole-DDE, nach der Einführung des KA.....	58
Abbildung 8.3: Interaktion zwischen Konsistenzagenten und REVISE-Systems.....	64
Abbildung 9.1: UML-Diagramm für den Konsistenzagenten.....	66

1. Einführung

1.1 Motivation

Eine breit verfügbare Infrastruktur von vernetzten Rechnern mit der Fähigkeit der Verarbeitung von beliebigen Daten eröffnet Anwendern die Möglichkeit kooperativ zusammenzuarbeiten und dabei sowohl räumliche als auch zeitliche Entfernungen zu überbrücken. Diese Form der Kooperation wird allgemein unter dem Begriff Computer-Supported Collaborative Work (CSCW) beschrieben und zusammengefasst [StRa99].

Ein Teilbereich des CSCW sind die Systeme für Verteilte Dokumenten-Bearbeitung auf englisch **Distributed Document Environment (DDE)**. DDE bietet Menschen weltweit die Möglichkeit, gemeinsam und auch zu unterschiedlichen Zeiten an demselben Dokument zu arbeiten. DDE kommt beispielsweise zum Einsatz, wenn:

- Experten weit voneinander entfernt sind.
- Experten in unterschiedlichen Zeitzonen leben.
- Kulturell bedingte Unterschiede zwischen verschiedenen Ländern zu Unterschieden bezüglich Arbeitszeiten und Arbeitsweisen führen.
- Mitarbeiter desselben Standorts sehr oft durch weite und lange Reisen voneinander getrennt sind.

Beispiele solcher Zusammenarbeit sind das gemeinsame Schreiben von Code, sowie dessen Dokumentation und das Verfassen von Benutzer-Handbüchern, was oft in der gemeinsamen Entwicklung von Software vorkommt.

Bei einer gemeinsamen Dokumentenbearbeitung können gleichzeitig mehrere Replikate des ursprünglichen Dokuments existieren. Jeder Bearbeiter kann mit einem Replikat arbeiten, welches nach Abschluss der Bearbeitung das ursprüngliche Dokument ersetzt. Benutzer, die z.B. mit mobilen Endgeräten beteiligt sind, haben keine permanente Verbindung zu dem ursprünglichen Dokument. Sie editieren lokal und bringen neue Inhalte in das gemeinsame Dokument ein.

Aus diesem Grund treten **Abweichungen** in zwei verschiedenen Formen auf:

1. Abweichungen zwischen den verschiedenen Replikaten der entsprechenden Bearbeiter.
2. Abweichungen zwischen den verschiedenen Replikaten und dem ursprünglichen Dokument.

Abweichungen der Replikate untereinander und Abweichungen vom Original werden in der relevanten Literatur als **Inkonsistenzen** bezeichnet. Ersetzt ein Replikat das Original so werden die **Inkonsistenzen** behoben und es tritt ein neuer **konsistenter Zustand** ein, in dem alle Änderungen des Replikats übernommen werden. Es kann sein, dass sich nicht alle Änderungen eines Replikats auf das Original übertragen lassen, da z.B. in einem anderen Replikat an identischen Stellen ebenfalls Änderungen vorgenommen wurden.

Ziel in einer DDE ist die **Konsistenzerhaltung** durch Übernahme aller Änderungen oder die Vermeidung aller Änderungen. Treten Inkonsistenzen auf, so übernimmt man so viele Änderungen wie möglich, indem man geeignete Regeln definiert, die festlegen, wie in Konfliktfällen entschieden wird. Kann der Konflikt nicht automatisch durch eine Regel behoben werden, so werden die entsprechenden Autoren über den Zustand benachrichtigt. Daraufhin müssen sie gemeinsam eine Entscheidung treffen, um den Konflikt zu beheben.

Als ein Beispiel für einen **inkonsistenten Zustand** können wir folgendes nennen: Wird ein solches Replikat, welches das Original ersetzt hat, von einem anderen Replikat überschrieben, so gehen die Änderungen des ersten Replikats verloren.

In obiger Schilderung ist bereits eine Konsistenzkondition für DDE festgelegt. Alle Replikate müssen den gleichen Inhalt haben. Neben dieser Regel können weitere Konditionen festgelegt werden. Man kann z.B. für jede Dokumentart eine Menge von Regeln definieren, die einzuhalten sind. Jeder Zustand, der eine von diesen Regeln nicht erfüllt, wird folglich als inkonsistent bezeichnet. Die entsprechenden Regeln werden dementsprechend Konsistenz-Regeln genannt.

1.2 Aufgabenstellung

In der Abteilung "Verteilte Systeme" der Universität-Stuttgart wurden zwei DDE-Systeme basierend auf verschiedenen Ansätzen entwickelt. Ein DDE-System auf Basis von CORBA, im folgenden CORBA-DDE genannt, und ein DDE-System auf Basis des mobilen Agentensystems Mole, im folgenden Mole-DDE genannt.

Die CORBA-DDE wurde in [ScPe99] um den Bereich Konsistenzbehandlung in mobiler Umgebung erweitert.

Für die Mole-DDE existiert noch keine Konsistenzbehandlung.

Ziel dieser Diplomarbeit ist es, für die existierende Mole-DDE geeignete Konzepte zur Konsistenzverhandlung zwischen mobilen Agenten zu entwerfen, diese prototypisch zu realisieren und in das existierende System zu integrieren. Hintergrund dieser Diskussion ist die Analyse des Potentials der mobilen Agenten gegenüber dem CORBA-Ansatz in Bezug auf Konsistenzerhaltung. Es gibt Gründe zu glauben, dass mit mobilen Agenten mehr Potential zur Konsistenzerhaltung existiert, basierend auf der Tatsache, dass mobile Agenten Argumente austauschen und miteinander kooperieren können.

1.3 Überblick

Die folgende Arbeit teilt sich in zwölf Kapitel. Nach der Einführung in diesem Kapitel folgen im zweiten Kapitel Erläuterungen von Begriffen, die uns im Laufe der gesamten Arbeit begegnen werden.

Im dritten Kapitel folgt eine Einführung in die Agententechnologie. Hier wird auch das in der Abteilung "Verteilte Systeme" am IPVR der Universität Stuttgart entwickelte mobile Agentensystem Mole vorgestellt.

Im vierten Kapitel wird die Mole-DDE und ihre für diese Diplomarbeit wichtigsten Komponenten vorgestellt.

Das fünfte Kapitel ist eine allgemeine Beschreibung von Ansätzen, die man bei der Implementierung eines Systems zur Konsistenzerhaltung verfolgen kann. Diese Ansätze decken verschiedene Aspekte eines solchen Systems ab und sind bereits in der Praxis eingesetzt worden. Parallel folgt eine Überprüfung, inwiefern diese Ansätze für die Konsistenzbehandlung in der Mole-DDE geeignet sind.

Im Kapitel sechs werden die Anforderungen an unser Konsistenzerhaltungssystem vorgestellt und spezifiziert.

Kapitel sieben beschreibt den Entwurf dieses Systems. Es wird auch ein geeignetes Konzept zur Konsistenzbehandlung mit mobilen Agenten vorgestellt.

Kapitel acht beschreibt, welche Änderungen der existierenden Mole-DDE nötig sind, um unser Konsistenzerhaltungssystem zu implementieren. Die Interaktion zwischen den existierenden und neu hinzugekommenen Komponenten wird möglichst genau beschrieben.

Kapitel neun zeigt wichtige Details für die Implementierung eines solchen Systems auf.

Im Kapitel zehn werden Testszenarien zur Kontrolle der Funktionalität und der Effektivität des KES vorgestellt.

Kapitel elf und zwölf schließlich sind eine Zusammenfassung und Bewertung des KES bzw. ein Vorschlag zur Erweiterung und Verbesserung des in dieser Diplomarbeit entwickelten Konzepts.

2. Grundlagen

Im Folgenden werden Begriffe erläutert, die uns in der ganzen Arbeit begleiten werden. Angefangen wird mit Basisbegriffen, wie z.B. Dokument oder Team und dann wird zu spezielleren Begriffen übergegangen, wie Konsistenz und was sie im konkreten Fall der gemeinsamen verteilten Dokumentenerzeugung bedeutet.

2.1 Gemeinsame Dokumentenerstellung

Diese Arbeit verwendet häufig den allgemeinen Begriff eines Dokuments. Um zu verstehen welche Bedeutung dieser Begriff hat, soll nun definiert werden, was unter einem Dokument im folgenden zu verstehen ist. Darüber hinaus wird noch der Begriff des Teams erläutert, auch ein grundlegender Begriff im Fall der gemeinsamen verteilten Dokumentenbearbeitung.

2.1.1 Beschreibung von Dokumenten

Beim Umgang mit Dokumenten ist es wichtig, eine Trennung in Daten, Layout und Struktur vorzunehmen. Die Komponenten haben dabei folgende Bedeutung:

- **Daten:** Diese bilden den aus Text, Abbildungen und Videosequenzen bestehenden Informationsgehalt des Dokuments.
- **Layout:** Um den Informationsgehalt darstellen zu können, muss dieser in einem bestimmten Format präsentiert werden. Dem Leser eines Dokuments muss die Informationsentnahme durch unterschiedliche Schriftgrößen oder Unterstreichungen erleichtert werden.
- **Struktur:** Die Gliederung des Dokuments in Kapitel, Abschnitte, Abbildungen oder Aufzählungen bezeichnet man als die Struktur eines Dokuments.

Speziell für diese Arbeit sind jedoch von großer Bedeutung die Daten und die Struktur eines Dokuments. Wichtig sowohl für die Daten als auch für die Struktur ist, dass beides nicht nur gelesen, sondern auch geändert werden kann und dass unter konkreten Voraussetzungen, die wir im weiteren Teil dieses Kapitels erläutern werden, diese Änderungen auch permanent werden können.

Die Struktur gliedert das Dokument in Teilbereiche auf, die miteinander verknüpft sind. Dies kann in unterschiedlicher Granularität geschehen. Eine sehr feine Granularität wäre gegeben, würde man jedes einzelne Wort eines Absatzes als einzelnes Strukturelement betrachten. Das Ziel einer solchen Gliederung ist es, Komponenten zu erhalten, die unabhängig voneinander behandelt werden können. Je feiner die Granularität ist, desto mehr Komponenten erhält man. Dadurch erhöht sich jedoch auch der Verwaltungsaufwand bzw. der Überhang an Informationen zur Verknüpfung dieser Komponenten. Deshalb ist eine sehr feine Gliederung häufig nur in der Theorie, jedoch nicht in der Praxis sinnvoll. Die Unabhängigkeit einzelner Komponenten und deren Granularität ist für ein Verteiltes Dokumentensystem von entscheidender Bedeutung, da Mechanismen wie Zugriffskontrollen oder Datenverwaltung diese direkt verwenden.

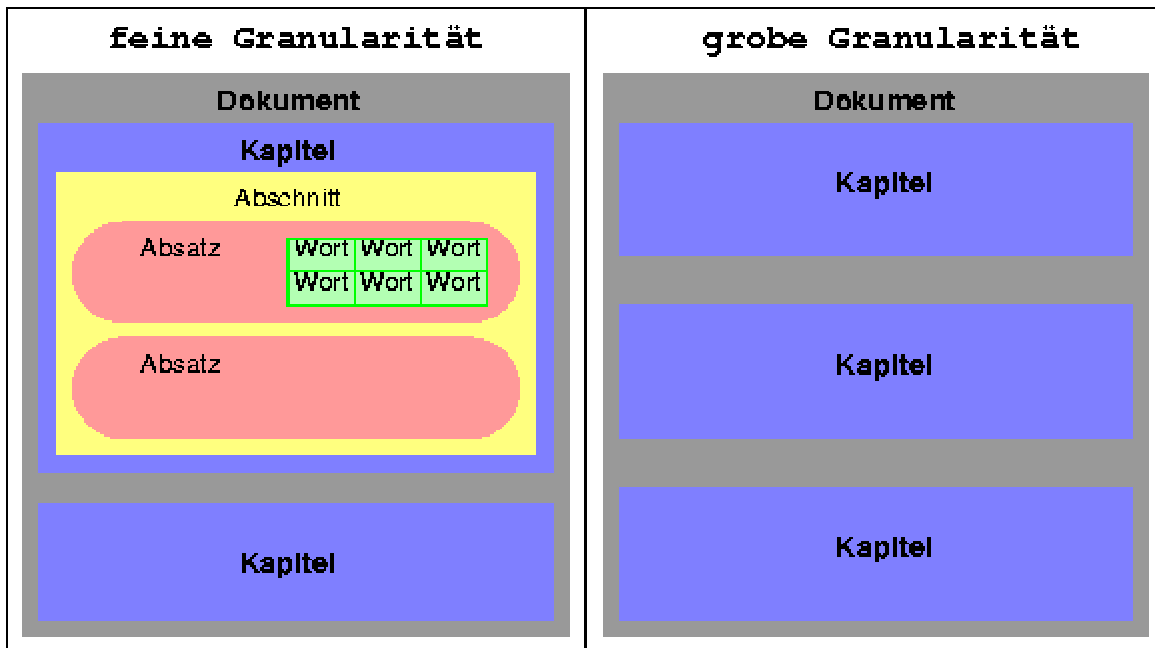


Abbildung 2.1: Granularität der Struktur eines Dokumentes

2.1.2 Das Team

Bei vielen Projekten sind Komplexität und Umfang der Arbeit für einen einzelnen Entwickler zu groß. Ein Team¹, bestehend aus mehreren, möglicherweise geographisch verteilten Teamteilnehmern muss diese Rolle übernehmen. Durch ihren kooperativen, sich in den Fähigkeiten der einzelnen Teilnehmer ergänzenden Charakter, sind Teams eher dazu in der Lage, solche Projekte zu einem erfolgreichen Ende zu führen.

Teamteilnehmer lassen sich folgendermaßen charakterisieren:

- Die Teilnehmer eines Teams verfolgen ein gemeinsames Ziel.
- Jeder verfügt über besondere Kenntnisse in einem Teilbereich des Projekts, die er in das Team einbringt.
- Ein Teilnehmer verfügt über einen bestimmten aktuellen Arbeitsplatz und bestimmte Arbeitszeiten, die sich zum Teil durch die geographische Verteilung ergeben und sich ebenfalls dynamisch ändern können.
- Innerhalb eines Teams übernimmt jeder eine bestimmte Rolle und damit ein bestimmtes Aufgabengebiet. Diese Rollenvergabe kann auch dynamisch sein und führt zu einer (eventuell hierarchischen) Struktur innerhalb des Teams.

Um Rechte und Aufgaben des Teams auch elektronisch verwalten zu können, bedarf es eines administrativen Benutzers, der zum einen die Definitionen der Rollen vornimmt und zum anderen den Teilnehmern die Rollen zuweisen kann.

¹ Definition des Begriffes Team kann in [BuCo97] nachgelesen werden.

Unterstützung bekommen sie hierbei durch geeignete Groupware-Systeme. Im Bereich dieser Groupware-Systeme ist für die vorliegende Arbeit speziell die gemeinsame Dokumentenerstellung von großer Bedeutung. Die Teilnehmer eines Teams wollen gemeinsam und zum Teil gleichzeitig an bestimmten Teilen der Dokumente arbeiten und dabei unter anderem Informationen über den momentanen Bearbeitungszustand des Dokumentes erhalten. Je nachdem welche Rolle ein bestimmter Benutzer im Team einnimmt, hat er spezielle Rechte und Aufgaben beim Zugriff auf die Dokumente. So kann beispielsweise ein Kommentator das vorhandene Dokument nicht löschen, sondern nur in einem speziell dafür vorgesehenen Bereich Anmerkungen einbringen. In dieser Arbeit werden uns die verschiedenen Rollen in einem Team nicht weiter beschäftigen. Wir werden uns auf Probleme konzentrieren, die zwischen Teamteilnehmern mit denselben Rollen entstehen.

2.2 Konsistenz als Ergebnis von Regeleinhaltung

Der Begriff Konsistenz und viele seiner Abkömmlinge wurden bereits mehrmals verwendet. Mit Hilfe von einigen Konsistenzregeln wird hier gezeigt, was im Fall von Dokumenten Konsistenz bedeuten könnte. Darüber hinaus wird auf Besonderheiten von DDE-Systemen hinsichtlich Konsistenzerhaltung eingegangen.

2.2.1 Konsistenz-Arten von Dokumenten

Im Fall eines Dokuments kann man mit Hilfe von Regeln, die immer einzuhalten sind, die verschiedenen Arten von Konsistenzen, die erhalten bleiben sollten, beschreiben:

- 1) Regeln, die aus der Existenz mehrerer Replikate eines Dokuments entstehen. Solche Regeln sagen z.B. in Bezug auf die gemeinsame Dokumentenbearbeitung, dass Teile von Berichten, die in einem Replikat gelöscht wurden, auch im Original gelöscht werden müssen, Teile, die einem Replikat hinzugefügt wurden, auch dem Original hinzugefügt werden müssen usw. Diese Konsistenz-Regeln heißen **replikatübergreifend**.
- 2) Regeln, die abhängig von der Art des Dokuments sind. So muß z.B. ein Dokument auf Deutsch den Regeln der deutschen Grammatik und Syntax genügen. Sowohl die Regeln der deutschen Grammatik als auch die Regeln der deutschen Syntax können als Konsistenzregeln für ein deutsches Dokument betrachtet werden. Solche Regeln könnten aber z.B. auch die Anzahl der Wörter, der Zeilen, der Absätze usw. eines Dokuments sein, oder dass die Einleitung nicht länger als zwei Seiten sein darf, oder die Anzahl der verschiedenen Wörter eines Dokuments usw. Solche Regeln trifft man bei Veröffentlichungen und Projektanträgen oft an.
- 3) Als letztes gibt es noch Konsistenzregeln zwischen mehreren Dokumenten. Nehmen wir als Beispiel ein Fahrradgeschäft, das auch eine Internet-Präsenz hat. Diese Internet-Präsenz könnte einen Umfang von mehreren Web-Seiten haben. Es könnte z.B. ein Produktkatalog existieren, Werbung für verschiedene Fahrräder, Kunden-Berichte (mit Listen von Produkten, die von verschiedenen Kunden gekauft wurden) und Service-Berichte (Beschreibungen von Problemen, die Kunden hatten). Es ist selbstverständlich, dass diese Informationen in den verschiedenen Dokumenten in enger Beziehung zueinander stehen. Modelle aus der Werbung müssen auch im Produktkatalog enthalten sein und deren Preise sollten in beiden Do-

kumenten übereinstimmen. Die Produkte aus den Kunden-Berichten sollten auch im Produktkatalog auftauchen und defekte Produkte aus den Service-Berichten müssen auch in den Kunden-Berichten der entsprechenden Kunden existieren. Diese Konsistenzregeln heißen **dokumentenübergreifend**.

Diese Liste der Konsistenzregeln erhebt keinen Anspruch auf Vollständigkeit, sie sollte einfach einen ersten Eindruck geben, was es für Konsistenzregeln im Fall von Dokumenten geben könnte. Auch nicht alle diese Regeln sind für diese Arbeit von gleicher Bedeutung.

2.2.2 Besonderheiten einer DDE (Distributed Document Environment)

[BeMi99] unterscheidet zwischen **gekoppelter** und **entkoppelter** Arbeit von Benutzern in Bezug auf deren Verbindung zu einem System. Eine Kopplung kann genauer betrachtet in eine physikalische und eine logische Verbindung unterteilt werden. Eine physikalische Verbindung bedeutet, dass zwischen den Rechnern eine Verbindung auf Netzwerkebene besteht. Bei einer logischen Verbindung funktioniert die Kommunikation auch auf Anwendungsebene. Eine logische Verbindung setzt also eine physikalische Verbindung voraus und ein Benutzer arbeitet gekoppelt mit einem System, wenn er mit dem System logisch verbunden ist. Arbeitet er dagegen ohne logische Verbindung, dann arbeitet er entkoppelt unabhängig davon ob eine physikalische Verbindung vorliegt oder nicht.

Jede Änderung der Verbindung kann deshalb als eine Änderung der logischen Verbindung betrachtet und als **Abkopplung** oder **Ankopplung** bezeichnet werden.

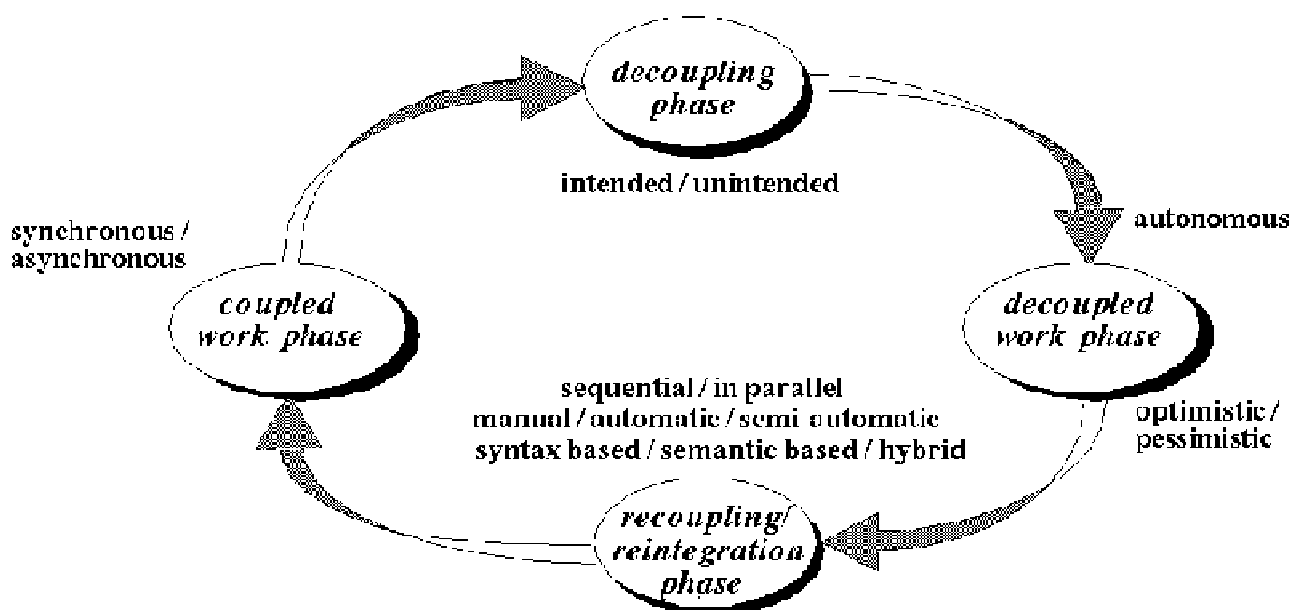


Abbildung 2.2: Gekoppelte/Entkoppelte Arbeit

Durch den Wechsel zwischen gekoppelter und entkoppelter Arbeit entsteht der aus [BeScVö96] stammende Kreislauf in Abbildung 2.2.

Eine DDE sollte in der Lage sein sowohl gekoppelte als auch entkoppelte Arbeit zu unterstützen. Eine solche Unterstützung ist in Bezug auf mobile Rechner unentbehrlich, macht aber auch ansonsten aus den folgenden Gründen Sinn:

- **Räumliche Streuung der Experten.** Das könnte ein Ergebnis der zunehmenden Komplexität der Aufgaben, der höheren Spezialisierung einzelner Personen sowie der verstärkten Globalisierung von Unternehmen und Organisationen sein.
- **Mobilität arbeitender Menschen.** Man erwartet von ihnen nicht nur, dass sie längere Reisen mit sich ständig ändernden Orten antreten, sondern auch, dass sie während ihrer Reisen weiterhin mit allen anderen Leuten zusammenarbeiten können, genauso als ob sie in ihrer Basis geblieben wären.
- **Technische Eigenschaften mobiler Rechner.** Mobile Rechner erleichtern zwar die oben erwähnte Mobilität der Menschen, haben aber z.B. eingeschränkt verfügbare Energie zum Senden und Empfangen von Informationen.
- **Bandbreite und Verbindungsdauer** der Netzwerkanbindungen sind begrenzt, sowohl aus technischen als auch aus wirtschaftlichen Gründen.
- **Kreativität.** Man sollte aus Kreativitätsgründen in der Lage sein, sich auch für längere Zeiträume zurückzuziehen, um sich allein Gedanken zu machen.

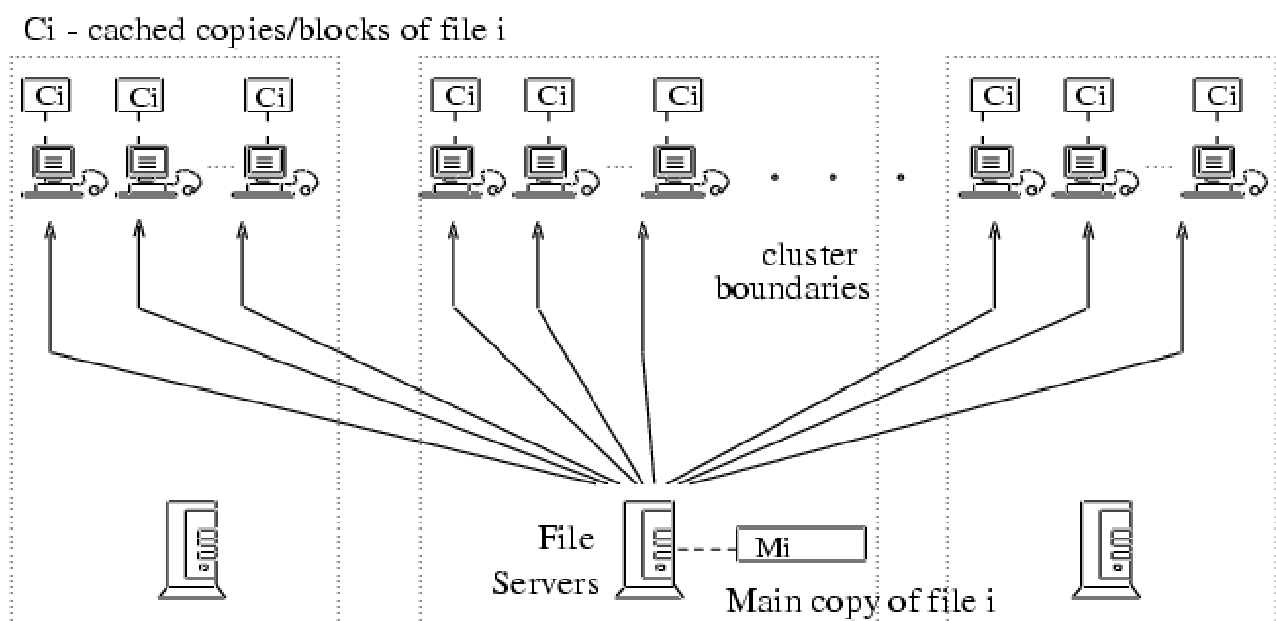


Abbildung 2.3: Caching

Damit eine sinnvolle entkoppelte Arbeit überhaupt möglich ist, muss die Nutzerinstanz benötigte Daten lokal bereithalten [RaRePo97]. Da diese Daten jedoch eventuell auch auf dem ursprünglichen Rechner benötigt werden, reicht ein Verlagern der Daten nicht aus. Stattdessen müssen Replikate der Daten auf den Rechnern der Benutzer erzeugt und verwaltet werden.

In verteilten Dateisystemen z.B. Andrew File System (AFS) oder Network File System (NFS) geschieht etwas sehr ähnliches unter dem Begriff des "Caching". Dabei steht dort allerdings nur die Erhöhung der Zugriffsgeschwindigkeit im Vordergrund. Außerdem wer-

den die Dateien meist nur kurze Zeit im Cache gehalten, so dass von temporären Kopien gesprochen wird. Siehe hierzu auch Abbildung 2.3.

Im Bereich mobiler Rechner hingegen wird eher der Begriff der Replikation von Daten verwendet. Die einzelnen Kopien der Files werden dabei als "Replikat" bezeichnet. Unter Replikation wird dabei nicht nur das Erstellen der Kopien verstanden, sondern auch die Verwaltung der Kopien und die Verbreitung von Änderungsinformationen über geänderte Dateien.

Durch entkoppelte Arbeit können Inkonsistenzen entstehen. Betrachten wir hierfür eine Version x eines Kapitels, die sich ein Benutzer A auf seinen mobilen Rechner kopiert und während einer Geschäftsreise verändert. Während dieser Zeit wird die auf dem DDE-Server liegende Version von anderen Benutzern (B und C) verändert und jeweils abgespeichert. Nach Ende seiner Geschäftsreise will nun Benutzer A seine veränderte Version wieder auf den DDE-Server zurückspeichern. Dabei kann die Version des Benutzers A nicht einfach als neue Version gespeichert werden, denn dabei würden die Veränderungen der Benutzer B und C verloren gehen. Stattdessen gibt es nun zwei Varianten dieses Kapitels die zu einer neuen Version $x+4$ zusammengeführt werden müssen. In Abbildung 2.4 ist solch ein Auseinanderlaufen und wieder Zusammenführen der Versionen dargestellt.

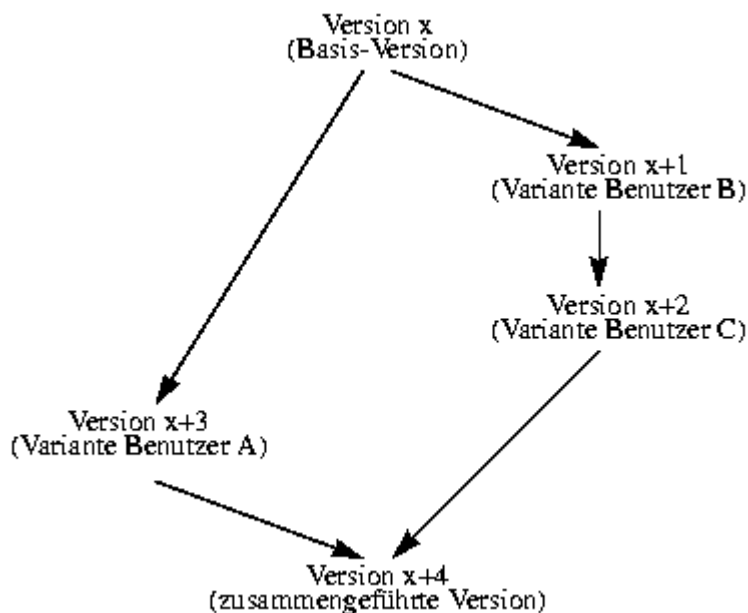


Abbildung 2.4: Entstehung und Behebung von Inkonsistenzen

3. Agententechnologie

Autonom handelnde Rechner, was eigentlich soviel wie autonom handelnde Software bedeutet, bleiben auch heute im Jahr 2002 immer noch eine Utopie, obwohl Menschen bereits in den 60er Jahren in Science-Fiction Filmen davon geträumt haben. Es gibt aber eine relativ neue Software-Technologie, die sich in diese Richtung bewegt. Die Agententechnologie, ein Programmierparadigma, das seinen Ursprung in der künstlichen Intelligenz hat, bildet heute aufgrund der verteilten künstlichen Intelligenz eine Schnittstelle zwischen der künstlichen Intelligenz und den verteilten Systemen. Im folgenden wird ein kurzer Überblick über die Agententechnologie und deren Besonderheiten vorgestellt. Im Mittelpunkt stehen dabei mobile Agentensysteme und insbesondere das Mole-System, ein mobiles Agentensystem implementiert in der Abteilung "Verteilte Systeme" im IPVR der Universität Stuttgart, welches auch die Basis des untersuchten Mole-DDE Systems ist.

3.1 Definitionen

Laut Duden handelt es sich bei einem Agenten um jemanden, "der im Auftrag für einen anderen eine Aufgabe erledigt". Diese Definition beschreibt zwar, aus welchem Hintergrund heraus die Namensgebung für Agenten entstand, im Bereich der Informatik existieren allerdings weitergehende Definitionen von Agenten. Exemplarisch sollen an dieser Stelle zwei Definitionen näher beschrieben werden.

In [WoJe94] ist ein Agent "ein Programm, welches in der Lage ist, seine Entscheidung und sein Handeln basierend auf der Wahrnehmung seiner Umwelt, bei der Verfolgung eines oder mehrerer Ziele selbständig zu kontrollieren".

Dabei muss es die folgenden Eigenschaften erfüllen:

- **Autonomie:** Die Agenten lösen eine ihnen gestellte Aufgabe selbständig, d.h. ohne weitere Benutzereingriffe und mit eigenständiger Kontrolle über ihr Handeln und ihren internen Status.
- **Kooperationsfähigkeit:** Besonders wichtig für ein Multiagenten-System ist, dass Agenten in der Lage sind, mit anderen Agenten und Anwendern zusammenzuarbeiten.
- **Reaktionsfähigkeit:** Agenten beobachten ihre Umgebung und reagieren auf auftretende Veränderungen.
- **Unternehmungsgeist:** Im Gegensatz zu Objekten reagieren Agenten nicht nur auf Veränderungen, sie ergreifen sogar die Initiative bei der Verfolgung ihrer Ziele.

Diese Kriterien geben bei der Entwicklung von Agenten einen relativ engen Rahmen vor. Etwas allgemeiner gehalten ist die Definition von [WoMi99]. Ein Agent ist demnach "ein Computer System, welches sich in einer speziellen Umgebung befindet und zum Erreichen seiner Entwurfziele die Fähigkeit besitzt, autonome Aktionen auszuführen".

Er kann beispielsweise über Sensoren seine Umgebung überprüfen und diese über Aktionen verändern. Abbildung 3.1 zeigt dieses Szenario. Der Agent hat dabei allerdings nicht die totale Kontrolle über seine Umgebung. Es können auch andere Komponenten auf die Umgebung einwirken. Des Weiteren kann der Agent aus Sicherheitsaspekten auch nur partiellen Zugriff auf die Umgebung haben.

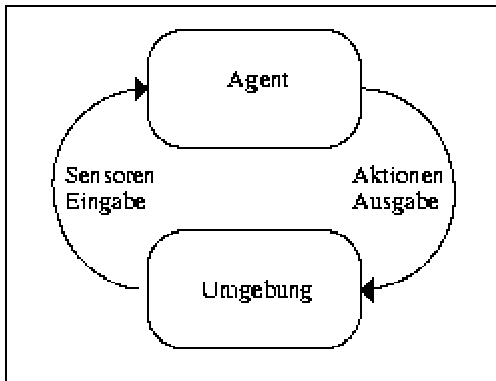


Abbildung 3.1: Umgebung von Agenten

Eine weitere Definition bietet die **Belief-Desire-Intention (BDI)** Architektur für kognitive Agenten [RaGe95]. Das Wissen eines Agents über sich selbst und seine Umwelt ist in Form von Überzeugungen/Meinungen (beliefs) dargestellt. Zusätzlich hat jeder Agent Wünsche (desires), in welchem Zustand seine Umgebung sein sollte. Auf Grund seines Glaubens und seiner Wünsche verfolgt jeder Agent Ziele (goals) und hat Absichten/Pläne (intentions), in welchem Zustand seine Umgebung sein sollte. Die BDI Architektur verbindet also Wissen, Wünsche und Ziele eines Agents mit seiner Art die Umwelt wahrzunehmen und seinen Aktionen ([HuSi98], [LePa97]).

3.2 Multiagenten-System

Bereits oben wurde der Begriff Multiagenten-System (**MAS**) verwendet, in diesem Abschnitt werde ich ohne Formalismen eine möglichst einfache und verständliche Definition für diesen Begriff geben.

Zentraler Punkt aller obigen Definitionen für einen Agent ist das autonome Handeln. Wenn man mehrere Agenten hat, die zusätzlich zu dieser zentralen Eigenschaft, die Eigenschaft aufweisen miteinander zu verhandeln, spricht man von einem MAS. Unter Verhandlung versteht man die Analyse eines Problems mit der Absicht, eine Lösung (Eini-gung) zwischen den beteiligten Menschen oder im konkreten Fall Agenten zu finden [Le-Pa97].

Für die Analyse eines Problems sind Kenntnisse gefragt und diese Kenntnisse könnten ein oder mehrere Agenten besitzen. Ein MAS ist also ein System von mehreren Agenten, die ohne zentrale Koordinierung in der Lage sind, ein Problem zu zerlegen und anschließend zu lösen. Wichtig ist dabei zu begreifen, dass diese Agenten keine zentrale Unterstützung oder Vorgaben bekommen, da keine solche zentrale Instanz existiert und auch, dass jeder entsprechend seiner Kenntnissen zur Problemlösung beiträgt, wobei außer ihren Kenntnissen auch die Verhandlungen unter ihnen eine wichtige Rolle spielen.

3.3 Mobile Agentensysteme

Einen Spezialfall eines Agentensystems stellt ein mobiles Agentensystem dar. Hier ergeben sich durch die Mobilität der Agenten besondere Anforderungen.

Damit in Agentensystemen ein gewisses Maß an Sicherheit herrscht, muss man die lokalen Ressourcen vor unerlaubtem Zugriff schützen. In mobilen Agentensystemen, bei denen Agenten von einer Lokation zur nächsten wandern können, kann ihnen beispielsweise der Zugriff auf die Umgebung komplett untersagt werden. Um an Informationen zu kommen, müssen sie mit ortsfesten Agenten kooperieren.

Deshalb können im Bereich mobiler Agentensysteme die Agenten aufgrund ihrer Befugnisse und Möglichkeiten in zwei Klassen eingeteilt werden:

- **Systemagenten:**
Die Agenten sind ortsfest, können sich also nicht von einer Stelle zu einer anderen bewegen. Sie haben Zugriff auf Systemressourcen.
- **Mobile Agenten:**
Die Agenten können migrieren, d.h. von einer Lokation zu einer anderen wechseln. Es handelt sich hierbei um eine Prozessmigration, bei der die Information über den aktuellen Zustand beibehalten wird. Mobile Agenten dürfen aus Sicherheitsgründen nicht auf alle Systemressourcen zugreifen, sondern müssen sich über ortsfeste Systemagenten Zugang zu Ressourcen verschaffen.

Diese Aufteilung ist für die hier vorliegende Arbeit von besonderer Bedeutung, da auch das als Grundlage dienende MOLE diese Unterscheidung macht.

3.4 Eigenschaften

Agenten und insbesondere Mobile Agenten sind ein relativ neues Programmiermodell. Während die traditionellen objektorientierten Ansätze ein Netz von Komponenten bilden, welche in einem nach außen offenen System arbeiten und dort Dienste anbieten, ist der Ansatz bei einem Mobilen Agentensystem ein anderer.

Ein Agentensystem ist ein in sich geschlossenes System, in dem sich die Agenten frei bewegen können, Dienste anbieten oder vollbringen können und dabei ein gemeinsames kooperatives Ziel verfolgen. Die Agenten können diese Umgebung nicht verlassen, sie stellt eine Art Sandkasten dar, aus dem die mobilen Agenten nicht heraus können.

Diese Agentenumgebung ermöglicht überhaupt erst eine Ausführung der Agenten. Sie stellt sogenannte Lokationen zur Verfügung: Orte, an denen sich die Agenten aufhalten können. So können alle Agenten miteinander kommunizieren. Zusätzlich können Systemagenten auch die externe Umgebung beobachten und verändern.

Die Umgebung teilt sich also in einen Bereich, der für alle Agenten zugänglich ist und in einen, auf den nur die Systemagenten zugreifen können.

Agenten entscheiden selbst, ob eine ihrer Methoden aufgerufen wird oder nicht. Dies unterscheidet sie klar von Objekten, bei denen normalerweise das Objekt nicht darüber entscheidet, ob ein Aufruf seiner Methoden Sinn macht oder nicht.

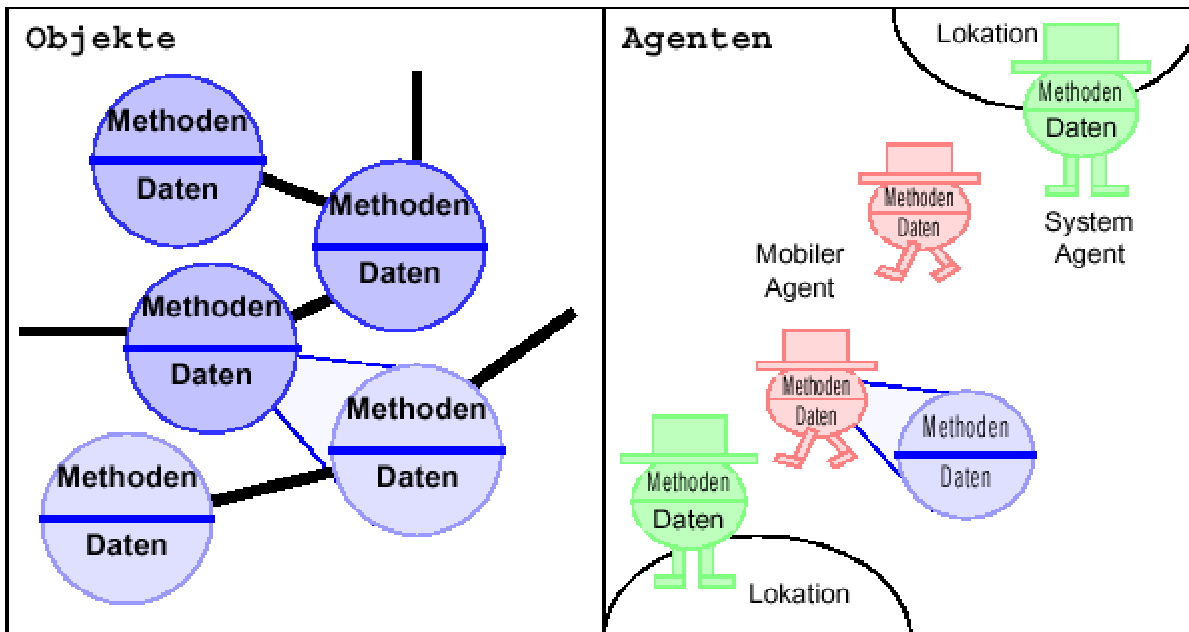


Abbildung 3.2: Darstellung von Objekten und Agenten

3.5 Bewertung

Durch die Unterschiede zu herkömmlichen Programmierparadigmen ergeben sich sowohl Vor- als auch Nachteile. Diese bestehen in der Art der Programmierung und den Möglichkeiten ihrer Ausführung.

- **Vorteile:**
 - Durch die Fähigkeit zur Migration können sich die Agenten zu den Daten begeben und nicht umgekehrt, was bei großen Datenmengen vorteilhaft ist
 - Agenten sind autonom, d.h. sie entscheiden kontextabhängig selbst, ob sie Methoden ausführen oder nicht.
 - Agenten sind dem Menschen nachgebildet, wodurch ihre Arbeitsweise leichter verständlich ist.
- **Nachteile:**
 - Alle Agenten können nur innerhalb eines Agentensystems existieren.
 - Durch den Unterschied zur objektorientierten Programmierung erfordert die Agentenprogrammierung ein Umdenken.
 - Durch die Migration dürfen mobile Agenten nur eingeschränkt auf Systemressourcen zugreifen.
 - Der Mehraufwand der Agenten-Kommunikation bzw. des Agentensystems sorgt für einen Leistungsverlust.

Durch den Einsatz von Agentensystemen ergeben sich also neue Möglichkeiten und Probleme in der Realisierung komplexer Szenarien.

3.6 Das Mole-System

Wie bereits in der Einführung erwähnt, ist das Ziel dieser Arbeit, die Möglichkeiten von mobilen Agenten in Bezug auf Konsistenzerhaltung von Dokumenten in DDE-Systemen zu untersuchen. Zu diesem Zweck wird das bereits existierende Mole-DDE in Betracht gezogen, welches auf dem Mole-System basiert, ein System mobiler Agenten. Deshalb will ich in diesem Kapitel etwas näher auf die Besonderheiten von MOLE eingehen.

3.6.1 Die Struktur von Mole

Abbildung 3.3 zeigt eine etwas vereinfachte Struktur eines auf der Basis von Mole laufenden Systems. Durch die Verwendung von Java als Programmiersprache kann Mole auf verschiedenen Betriebssystemen laufen. Mole selbst stellt dabei pro Rechner beliebig viele Lokationen zur Verfügung und pro Rechner eine "MOLE ENGINE": eine Art übergeordneter Prozess, der die anderen Prozesse überwacht.

Mole stellt auch eine Java Klassenbibliothek zur Verwendung in eigenen Projekten dar. Agenten selbst gibt es in MOLE noch nicht, deren Entwicklung bleibt vollständig dem Benutzer von MOLE überlassen.

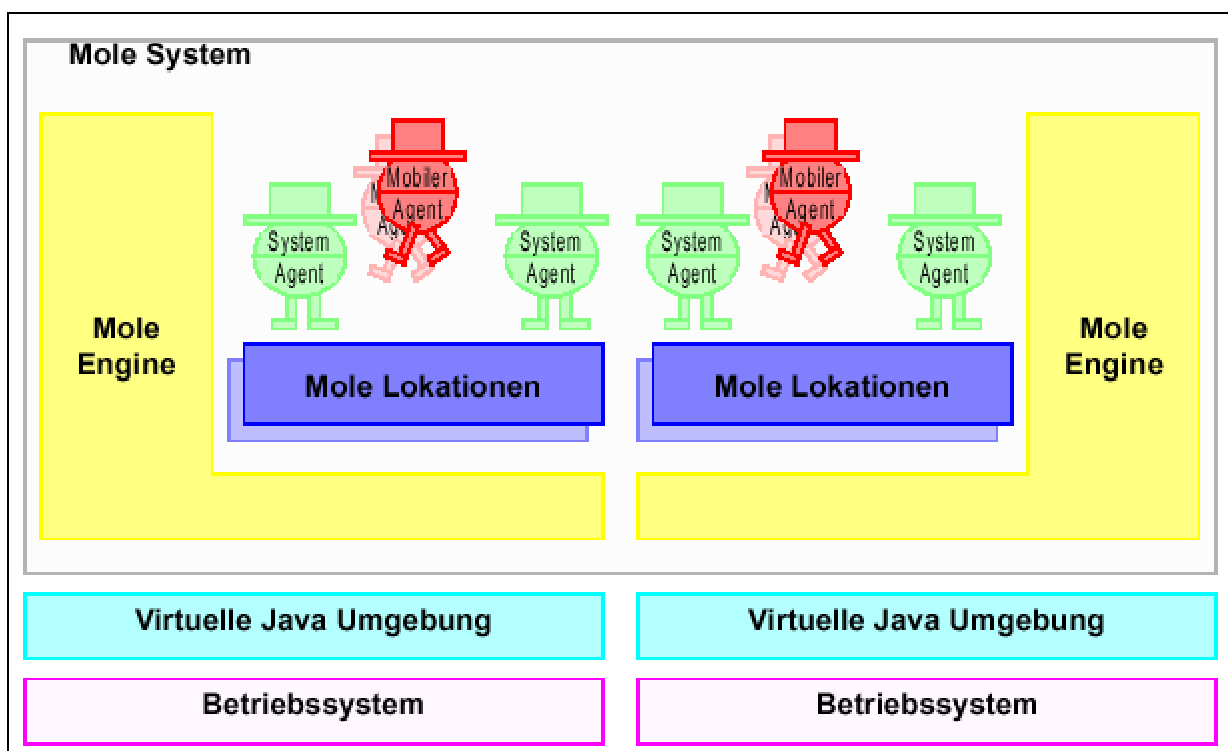


Abbildung 3.3: Das Mole System

Die "MOLE ENGINE" muss zwar auf jedem Rechner separat gestartet werden, das Mole-System aber sorgt für Transparenz in Bezug auf Rechnernamen und deren genaue Position im Internet. Ein Benutzer braucht also nur die Lokationen, an denen sich die Agenten befinden ohne die exakten IP-Adressen der Rechner, auf denen sie laufen, zu kennen. Es können sogar prinzipiell mehrere abgeschlossene Agentensysteme innerhalb eines Mole-Systems laufen, ohne sich gegenseitig zu behindern.

3.6.2 Mole Lokationen und Agenten

Mole ist ein mobiles Agentensystem oder genauer gesagt, Mole stellt sowohl Klassen für mobile Agenten, als auch für Systemagenten zur Verfügung, wobei nur letztere direkt auf die Umgebung außerhalb des Mole-Systems zugreifen können. Agenten haben einen eindeutigen Namen, der der Identifikation innerhalb des Systems dient, wobei die "Engine" darauf achtet, dass die Namen eindeutig bleiben.

Die Lokationen sind Orte, an denen sich die Agenten treffen und lokal Informationen austauschen können. Sie sind vor dem Start des Systems statisch anzulegen. Jede Lokation hat einen eindeutigen Namen, ähnlich einem Rechner im Internet, beispielsweise "moleof-ficel.mole.informatik.uni-stuttgart.de". Jede Lokation kann vor dem Start des Systems mit Agenten "gefüllt" werden. Diese Agenten werden dann automatisch durch die entsprechenden Lokationen gestartet. Es muss also mindestens einen solchen Agenten im gesamten System geben, der statisch angelegt wurde und dann selbst beliebig andere Agenten erschaffen kann.

Jeder Agent hat eine bestimmte Lebenszeit, die mit dem Aufruf der "start-Methode" für die entsprechende Lokation beginnt, und mit dem Aufruf der "die-Methode" endet. Bei der Migration von Agenten werden diese kurzzeitig beendet, über den Serialisierungsmechanismus von Java in einer Objektdatei gespeichert, an die neue Lokation verschoben und dort schließlich erneut durch den Aufruf der "start-Methode" gestartet, ohne dass die Informationen der Daten verloren gehen. Da nach dem erneuten Start des mobilen Agenten sich die Informationen seiner Umgebung verändert haben können, muss darauf geachtet werden, dass globale Variablen, die speziell für die alte Lokation gültig waren, noch immer ihre Gültigkeit besitzen.

3.6.3 Agentenkommunikation

Die Kommunikation zwischen den Agenten kann auf zweierlei Weise geschehen. Zum einen ist es möglich, eine Methode eines Agenten aufzurufen. Dabei wird von der API unterschieden, ob die Agenten sich innerhalb derselben "Engine" befinden oder nicht. Wenn sich die Agenten auf derselben "Engine" befinden, handelt es sich um einen Prozeduraufruf mit Parameterübergabe **call by reference**, ansonsten handelt es sich um einen RPC (Remote Procedure Call), es wird also wie üblich in einem solchen Fall eine Kopie der Daten übergeben (**call by value**).

Die zweite Methode stellt das Versenden von beliebigen Nachrichten dar, wobei der Empfänger selbst entscheiden kann, ob er die Nachrichten gleich verarbeiten will oder sie in einer Mailbox aufbewahrt, um sie zu einem späteren Zeitpunkt abzuarbeiten. Die Nachrichten sind ähnlich wie Emails aufgebaut; sie haben einen Absender, einen Empfänger und Inhalt.

4. Das Mole-DDE-System

Wie bereits in der Einleitung beschrieben, ist das existierende Mole-DDE-System ein Ergebnis mehrerer Arbeiten. Aufbauend auf dem mobilen Agentensystem Mole wurde MoleOffice entwickelt, ein System, welches über eine grafische Benutzeroberfläche den Teamteilnehmern die Möglichkeit bietet, sich über Benutzer und Räumlichkeiten zu informieren. Es gibt Auskunft über aktive Tätigkeiten und die Bereitschaft zur Kommunikation. Es stellt also ein Benachrichtigungssystem zum Austausch von Informationen dar. Die Mole-DDE ist eine Erweiterung des MoleOffice-Systems, erweitert um die Möglichkeiten gemeinsamer Dokumentenbearbeitung. Im weiteren werden Teile der Mole-DDE vorgestellt, die für diese Diplomarbeit relevant sind. Genauer über die Mole-DDE kann man in [WeLu00] finden.

4.1 Überblick über die Struktur des DDE-Systems

Mole-DDE lässt sich prinzipiell in fünf Teilkomponenten aufgliedern:

- **Editor:** Auf oberster Ebene befinden sich die Editoren, die für die eigentliche Arbeit mit dem Dokument und dessen Layout zuständig sind
- **Grafische Benutzeroberfläche:** Zusätzlich zum Editor befindet sich auf der obersten Ebene auch eine eigene grafische Benutzeroberfläche.
- **Wrapper:** Die Editoren werden über einem Datenumsetzer, den Wrapper, an das darunter liegende Agentensystem angeschlossen.
- **Agentensystem:** Das Agentensystem dient zur eigentlichen Verarbeitung der Informationen und stellt die Hauptkomponente des verteilten Dokumentensystems dar.
- **Bücherregalkomponenten:** Auf der untersten Ebene befinden sich die Komponenten des Bücherregals. Diese greifen direkt auf das Dateisystem zu.

Die GUI kommunizieren mit den Agenten über RMI (Remote Method Invocation), während die Wrapper direkt über Sockets kommunizieren.

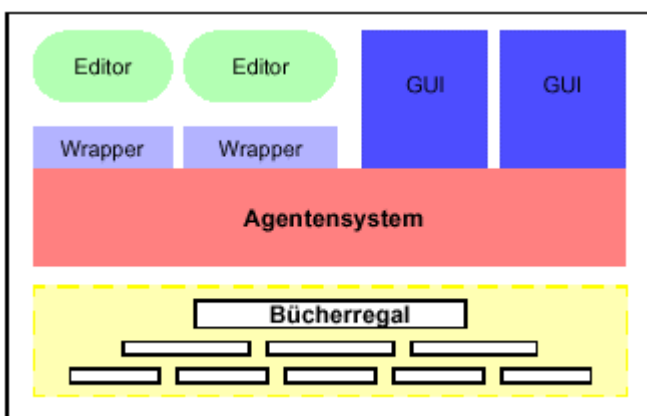


Abbildung 4.1: Komponenten der Mole-DDE

Die Mole-DDE hat eine Client-Server Architektur. Jeder angemeldete Benutzer, genauer gesagt seine GUI, sein Wrapper und sein Editor stellen einen Client dar, während für alle Benutzer ein Server existiert, bestehend aus dem Agentensystem und dem Bücherregal.

4.2 Verwendete Dokumentenstruktur

Mole-DDE unterscheidet verschiedene Bücherregale. Diese enthalten einzelne Bücher, welche wiederum in einzelne Kapitel unterteilt werden. Diese Anordnung ergibt, wie in Abbildung 4.2 verdeutlicht, eine hierarchische Struktur.

Die Gliederung der Bücherregalkomponenten stellt zwar nur eine relativ grobe Struktur dar, dennoch können bei einer Synchronisation auch einzelne Wörter abgeglichen werden. Die vorhandene Objektstruktur erlaubt eine weitere Untergliederung der Kapitel in Abschnitte beziehungsweise Wörter. Sie ist aber bisher noch nicht implementiert. Neben dieser hierarchischen Struktur existiert auch eine sequentielle Reihenfolge der Kapitel innerhalb eines Buches bzw. der Bücher innerhalb eines Bücherregals. Zur eindeutigen Identifikation der Bücher und Kapitel wird eine Kombination aus einer eindeutigen Nummer und einem Namen (Kapitel-, Buch-, Bücherregal-) verwendet. Mole-DDE selbst kann sowohl mit Namen als auch mit Nummern umgehen, die Editorumgebung aber (Emacs und Emacs-Wrapper) kann nur mit einer Kombination aus Buch- und Kapitelnamen umgehen, was eine Doppelvergabe eines Kapitelnamens innerhalb eines Buches zwar ausschließt, dem Benutzer aber nicht die Möglichkeit gibt, die Bücher und Kapitel neu anzuordnen.

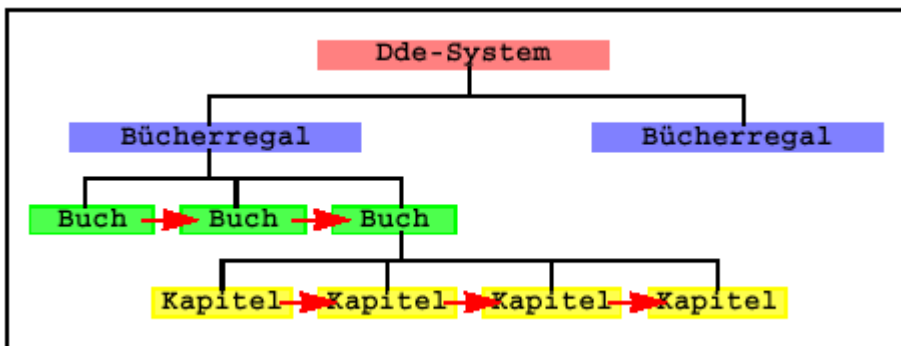


Abbildung 4.2: hierarchische Dokumentenstruktur der Mole-DDE

In Mole-DDE selbst ist dies jedoch möglich. Auch die Umbenennung eines Buches oder Kapitels ist in Mole-DDE möglich, wird aber auf der obersten Ebene im Editor nicht angeboten.

Alle Dokumente dieser Dokumentenhierarchie besitzen eigene Protokollierungsfunktionen. Sämtliche Modifikationen werden festgehalten, so dass sowohl dem System als auch den Teamteilnehmern ersichtlich ist, welche Historie das Dokument aufweist. Es ist z.B. ersichtlich, wer das Dokument angelegt hat, wer es seit seiner Erstellung verändert oder auch wer es gerade bearbeitet.

Um die Datenhaltung der Bücherregalkomponenten effizient zu gestalten, wurde sowohl ein Lese- als auch ein Schreibcache innerhalb der Dokumentenstruktur integriert. Dokumente werden bei einem Lesezugriff erst in den Speicher geholt und verharren dort, bis auf sie eine gewisse Zeit lang nicht mehr zugegriffen wurde. Ein Schreibzugriff wird unmittelbar nur im Speicher vorgenommen. Erst eine automatisch ablaufende Autospeicherung

schreibt veränderte Daten wieder zurück auf das Speichermedium, damit sie auch persistent sind und bei einem erneuten Start des Systems dem Bücherregal vorliegen.

4.3 Informationsaustausch

Innerhalb von Mole-DDE wird man über die Aktivitäten der anderen Teilnehmer, bezüglich ihrer Arbeit an den Dokumenten informiert, so beispielsweise auch, welche Kapitel gerade geöffnet sind. In MoleOffice sieht man, welche Aktivität ein Teilnehmer prinzipiell gerade vornimmt und dessen Kommunikationsbereitschaft. In beiden Informationssystemen (MoleOffice und Mole-DDE) kann der Informationsaustausch über die Konfiguration von Filtern gesteuert werden.

4.4 Aktionen

Mole-DDE verwendet zum Austausch dokumentenspezifischer Informationen ein spezielles Objekt, in dem die folgenden für die Kommunikationspartner wichtigen Informationen über die Aktion beinhaltet sind:

- **Das betroffene Objekt der Aktion:** Dies kann z.B. ein Kapitel oder ein Buch sein.
- **Der Erzeuger der Aktion**
- **Die erzeugte Aktion:** Dies ist eine eindeutige Beschreibung der Aktion, wie z.B. „Kapitel lesen“.

Bisher existieren die folgenden Aktionen, die über ein DDE-Aktionsobjekt transportiert werden:

- **Buch erzeugen**
- **Buch löschen**
- **Kapitel erzeugen**
- **Kapitel löschen**
- **Kapitel öffnen**
- **Kapitel schließen**
- **Kapitel lesen**
- **Kapitel schreiben**

Das Bücherregal wird, falls es noch nicht existiert, automatisch erzeugt. Beim Öffnen und Schließen von Kapiteln handelt es sich um einen einmaligen Vorgang, wenn man ein Kapitel bearbeiten will. Auch mehrmaliges Lesen und Schreiben öffnet und schließt das Dokument zwischenzeitlich nicht. Durch das Öffnen und Schließen wird der Arbeitsbereich des entsprechenden Dokuments also betreten oder verlassen, d.h. es wird festgestellt, ob bei gegebener Situation ein Zugriff erlaubt ist oder nicht.

Folgende weitere Aktionen sind noch möglich:

- **Inhaltsverzeichnis des Bücherregals lesen**
- **Inhaltsverzeichnis eines Buches lesen**
- **Anmeldung an das System**
- **Abmeldung vom System**

Indem ein Benutzer den Inhalt eines Inhaltsverzeichnisses im Editor sieht, erkennt er automatisch den Erfolg seiner Aktion. Anfragen nach dem Inhaltsverzeichnis werden auch stets erfolgreich ausgeführt¹.

4.5 Editor

Als Editor wird der XEmacs verwendet, was allerdings nicht bedeutet, dass die Mole-DDE nur mit dem XEmacs als Editor möglich ist. Der Grund dafür, dass hier der XEmacs verwendet wird, ist, dass die an einem Einbenutzer-Editor nötigen Erweiterungen, um ihn an die Mole-DDE anzuschließen, zur Zeit nur für den XEmacs existieren. Diese zusätzliche Funktionalität, wurde nicht als ganzes in den Editor integriert, sondern ein Teil davon existierte bereits in der grafischen Benutzungsoberfläche von MoleOffice und ein anderer Teil wurde in Form einer weiteren Komponente, des Wrappers, hinzugefügt.

Innerhalb des Editorbereichs wird man leider nur über seine eigenen Aktionen unterrichtet. Da die Erweiterungen am XEmacs keine informationsgesteuerte Kommunikation mit Rückmeldungen erlaubt, muß der Benutzer über den Erfolg oder Mißerfolg direkt unterrichtet werden und selbst agieren. Falls beispielsweise das Öffnen eines Kapitels fehlschlägt, wechselt der XEmacs trotzdem in den Kapitelmodus. Der Benutzer bekommt lediglich eine Mitteilung, daß das Kapitel nicht geöffnet werden kann und der Inhalt des Kapitels besteht aus einer Warnung, es nicht zu benutzen. Der Benutzer muß nun selbst dafür sorgen, wieder zurück in den Büchermodus zu wechseln. Alle darüber hinaus gehenden Informationen sind in die grafische Oberfläche integriert.

In Verbindung mit *Mole-DDE* bietet XEmacs folgende Modi an:

- **Bücherregalmodus:** Es wird das aktuelle Inhaltsverzeichnis des Bücherregals angezeigt, d.h. alle vorhandenen Bücher werden aufgelistet. Durch Auswahl eines Buches gelangt man in den
- **Büchermodus:** Die einzelnen Kapitel des entsprechenden Buches werden der Reihenfolge nach aufgelistet. Es kann entweder zurück in den Bücherregalmodus gewechselt werden oder durch die Auswahl eines Kapitels in den
- **Kapitelmodus:** Im Gegensatz zu den anderen beiden Modi wird hier nicht ein Inhaltsverzeichnis zur Auswahl eines Dokuments gezeigt, sondern der Inhalt selbst geöffnet.

4.6 Wrapper

Der Wrapper ist wie bereits erwähnt die andere zusätzliche Komponente für den Anschluß eines Einbenutzer-Editors an die Mole-DDE. Wrapper ist einfach gesagt eine Software-Hülle², die für das „eingewickelte“ Programm die nötige fehlende Funktionalität anbietet.

Die erste wichtige Funktion eines Wrappers ist die der Schnittstelle zwischen eines Editors und der Mole-DDE. Man kann also verschiedene Editoren in einem DDE-System verwenden, aber für jeden von diesen Editoren braucht man einen dazu geeignet programmierten Wrapper.

¹ Lediglich im Falle einer mißlungenen Anmeldung kann es nicht gelesen werden.

² Aus dem englischen Wrapper=Hülle

Die zweite wichtige Funktion des Wrappers ist die Informationsumsetzung und damit ist gemeint, daß die Informationen in beiden Richtungen sowohl Editor → DDE-System als auch DDE-System → Editor entsprechend manipuliert werden, damit der Sender dieser Informationen die fehlende Funktionalität der anderen Seite nicht bemerkt. Um es leichter zu formulieren, fehlt die entsprechende Funktionalität auf der anderen Seite, werden die gesendeten Daten, die diese Funktionalität ansprechen, entweder entfernt, oder die Funktionalität wird von dem Wrapper simuliert.

Als Beispiel für die Richtung Editor → DDE-System dient der Versuch des Editors ein Kapitel zu lesen ohne es vorher zu öffnen, oder der Versuch es zu verlassen ohne aber es zu schließen. Für den Editor sind „öffnen“ und „schließen“ Fremdwörter, daher übernimmt diese Aufgabe der Wrapper.

In der weiteren Arbeit meint der Begriff Editor sowohl den Editor als auch seinen Wrapper, es sei denn, es wird explizit zwischen den beiden unterschieden.

4.7 Schnittstelle zwischen Editor und Mole-DDE

Die Schnittstelle zwischen Mole-DDE und Editor ist fest definiert. Diese Definition muss zum Zweck der Integration eines beliebigen Editors mit der Mole-DDE eingehalten werden.

Sowohl hereinkommende Nachrichten, also vom Editor Richtung Mole-DDE, als auch ausgehende Nachrichten, also von der Mole-DDE Richtung Editor, haben eine sehr konkrete strenge Syntax. Diese Nachrichten heißen **Informations-Einheiten (IE)** und deren Syntax kann man auf der Abbildung 4.3 betrachten. Die zur Trennung einzelner Blöcke notwendigen Sonderzeichen "<" und ">" sind nicht als normale Zeichen zugelassen (ASCII Zeichen 60 und 62). Wie man aus der Abbildung 4.3 erkennen kann, hat eine IE immer die Form <id><cmd><content> gefolgt von einem "Newline" ASCII Zeichen. Das Token <id> zeigt die Richtung der Nachricht und (oder) den Teil der Struktur des Dokuments, auf den sich die Nachricht bezieht. Das Token <cmd> steht für "Command", es handelt sich um eine Anweisung, die genau beschreibt, was der Editor vom DDE-System erwartet (im Fall Editor→Mole-DDE) oder was passierte (im Fall Mole-DDE→Editor). Das letzte Token <content> ist zuständig für zusätzliche Informationen, wie z.B. den Namen eines Buches, eines Kapitels oder den Inhalt eines Kapitels.

Für die Kommunikation zwischen dem DDE System und dem Editor ist neben der Syntax der IE auch ein Kommunikationsprotokoll nötig. Es beschreibt in welcher Reihenfolge und unter welchen Bedingungen die Informationseinheiten eingesetzt werden dürfen.

Die Kommunikation zwischen XEmacs und Mole-DDE ist dabei nach dem Client/Server Modell konzipiert. Mole-DDE stellt den Server dar. Von seiner Sicht aus existieren beliebig viele Wrapper als Klienten. Für den XEmacs als Klient arbeitet der Wrapper hingegen transparent. Er kommuniziert quasi direkt mit dem Server. Somit ist der Wrapper in Wirklichkeit Klient und Server.

```

<IE> ::= "<" <ID> ">" <CMD> ">" <
        <CONTENT> ">" <NEWLINE>

<ID> ::= <DDE2WRAPPER> | <WRAPPER2DDE>

<DDE2WRAPPER> ::= "BookShelf" | "Minibuffer" | <OBJECT>

<WRAPPER2DDE> ::= "0" | "BookShelf" | <OBJECT>

<OBJECT> ::= <BOOKNAME> |
             <BOOKNAME> "/" <SECTIONNAME>

<BOOKNAME> ::= <BOOKNAME> <ZEICHEN> | <ZEICHEN>

<SECTIONNAME> ::= <SECTIONNAME> <ZEICHEN> | <ZEICHEN>

<CMD> ::= msg |
           init | close | bookShelfToc |
           bookToc | openBook | closeBook |
           createBook | destroyBook |
           openSection | closeSection |
           readSection | writeSection |
           createSection | destroySection

<CONTENT> ::= <CONTENT> | <CONTENT> <ZEICHEN>

<NEWLINE> ::= ASCII Zeichen 10

<ZEICHEN> ::= ASCII Zeichen 32-127 außer 60,62

```

Anmerkung:

Die genaue Definition der einzelnen Kommandos von <CMD> können der Datei *dde/wrapper/IE.java* entnommen werden .

Abbildung 4.3: Definition einer IE in BNF

Die Initiative geht üblicherweise vom XEmacs aus. Dieser meldet sich beim Start über den Wrapper bei Mole-DDE an und kann im folgenden mit diesem fast beliebig kommunizieren. Beim Beenden meldet er sich schließlich auf dem gleichen Weg bei Mole-DDE wieder ab. Geschieht dies nicht, sorgt ein spezieller Mechanismus dafür, dass nach einer gewissen Zeit die Verbindung automatisch unterbrochen wird.

Die Kommunikation zwischen XEmacs, Wrapper und Mole-DDE kann zum Teil parallel erfolgen. Der Editor fordert beispielsweise direkt nach dem Versenden der Anmeldeinformationen den Inhalt des Bücherregals an, noch bevor er eine positive Rückmeldung von Mole-DDE bekommt. Mole-DDE speichert die ankommenden Anfragen in einer Warteschlange und arbeitet sie sequentiell ab. Beim Editor kommen hintereinander zwei Nachrichten an. Die erste enthält die Anmeldebestätigung und die zweite den Inhalt des Bücherregals. Nachdem der Benutzer ein Buch ausgewählt hat, schickt der Editor auch hier eine Anfrage nach dem Inhaltsverzeichnis an Mole-DDE. Wichtig ist zu wissen, dass die Mole-DDE die Nachrichten speichert und sie sequentiell bearbeitet, die Reihenfolge bleibt also erhalten. Es kann allerdings vorkommen, dass durch Netzwerkfehler Nachrichten verloren gehen oder nur noch fehlerhaft am anderen Ende ankommen.

4.8 Grafische Benutzungsoberfläche

Die grafische Benutzungsoberfläche von Mole-DDE stammt zum größten Teil von Mole-Office, wurde aber auch erweitert, um der zusätzlichen Funktionalität von Mole-DDE gerecht zu werden.

Sie stellt eine der beiden Schnittstellen der Mole-DDE zur Außenwelt dar. Die andere Schnittstelle ist der Editor, also im konkreten Fall XEmacs, der bereits vorgestellt wurde. Obwohl XEmacs relativ einfach zu programmieren ist, ist es trotzdem nicht so einfach, den Komfort, den man von einer Mensch-Maschine-Schnittstelle im konkreten Fall erwartet, in den XEmacs zu integrieren. Eine solche Lösung wäre darüber hinaus nicht die bestmögliche, weil die grafische Benutzungsoberfläche mit jedem Editor verwendet werden kann, andernfalls müsste alles wieder erneut für jeden Editor programmiert werden.

Für jeden Benutzer kann, muss aber nicht, eine solche Schnittstelle laufen. Sie kann genauso wie der Editor gestartet und beendet werden, wann es jeder Benutzer für sinnvoll hält und stellt genauso wie der Editor einen Client dar, während das Mole-DDE-System als Server ununterbrochen laufen muss.

Die grafische Benutzungsoberfläche kommuniziert mit der Mole-DDE über den Systemagenten des entsprechenden Benutzers durch Java-RMI (**R**emote **M**ethod **I**nvocation). Die Systemagenten der Benutzer sind, wie bereits im Kapitel 3 beschrieben, an Lokationen gebunden und sie werden als Threads gestartet, sobald die für diese Lokationen zuständige Engine auch gestartet wird. Alles in allem laufen die Systemagenten der Benutzer unabhängig von den grafischen Benutzungsoberflächen, wartend auf Verbindung mit den entsprechenden Benutzern.

5. Aspekte eines Konsistenzerhaltungssystems (KES)

Im vorigen Abschnitt wurde das DDE-System vorgestellt, wofür geeignete Konzepte zur Konsistenzerhaltung der Dokumente untersucht, entworfen und prototypisch implementiert werden sollen. Es soll also ein neues KES entwickelt werden. Bevor noch konkrete Anforderungen an dieses KES formuliert werden, erachte ich es für sinnvoll, eine kleine Reise durch die relevante Literatur zu machen und die wichtigsten Aspekte eines solchen Systems vorzustellen.

Ziel dieser Untersuchung ist es, eine Antwort auf die Frage zu finden, welches von den Konzepten, die vorgestellt werden, im Fall des KESs für das bereits existierende DDE-System am besten geeignet ist, aus welchen Gründen und unter welchen Bedingungen.

Das Kapitel untergliedert sich in drei Abschnitte. Im ersten Abschnitt werden einige Transparenzaspekte erläutert. Abschnitt 5.2 widmet sich den Konsistenzerhaltungsverfahren, während Abschnitt 5.3 dann die Replikation und die dabei auftretenden Inkonsistenzen behandelt.

Die einzelnen Algorithmen sollen jeweils nur kurz angerissen werden, für genauere Betrachtungen sei auf die jeweils angegebene Literatur verwiesen. Ferner wird jeweils eine kritische Stellungnahme klarmachen, inwiefern diese Algorithmen für die konkrete Mole-DDE in Frage kommen oder nicht.

5.1 Transparenzaspekte

Dieser Abschnitt betrachtet die bei Replikation und Reintegration wichtige Frage der Transparenz für den Benutzer und für die Anwendungen. Hierbei werden Argumente für und gegen die Transparenz und die besonderen Anforderungen mobiler Benutzer aufgezeigt.

5.1.1 Anforderungen mobiler Benutzer und Replikate

In [RaPoRe96] und [RaRePo97] werden fünf besondere Anforderungen mobiler Benutzer und der hierfür nötigen Replikation aufgeführt:

- optimistische Replikationsalgorithmen mit peer-to-peer support
- Granularität der Replikation
- Skalierbarkeit der Replikation
- Mobil- Transparent und mit niedrigem Wartungsaufwand
- ohne pre-motion-actions auskommend

Zum einen fordern die Autoren optimistische Replikationsalgorithmen mit peer-to-peer Support für die Reintegration. Optimistisch bedeutet, dass jedes der Replikate unabhängig von den anderen geändert werden kann. Entsprechend sind Update-Konflikte zu erkennen und zu beheben. Untersuchungen der Autoren haben ergeben, dass Konflikte eher selten auftreten und dass diese dann oftmals auch leicht zu beheben sind.

Die Forderung nach peer-to-peer Support bedeutet, dass eine Reintegration von zwei beliebigen Replikaten möglich ist. Dies setzt voraus, dass alle Replikate gleichberechtigt sind und keine Master-/Slave-Hierarchie besteht. Eine ausführlichere Diskussion zu der Hierarchie der Replikate folgt noch im Abschnitt 5.3.2.

Die zweite Anforderung nach Replikation feiner Granularität wird in der Literatur auch als 'selective replication' bezeichnet. Hintergrund ist, dass eine präzise Auswahl der zu replizierenden Daten möglich sein sollte. Der Benutzer sollte in der Lage sein, nur die Daten, die er auch wirklich benötigt, ohne viel zusätzlichen "Datenballast" zu replizieren. Die Dokumentenstruktur der existierende Mole-DDE wurde im vorigen Kapitel vorgestellt. Demnach kann man nach Öffnen des entsprechenden Kapitels einzelne Abschnitte oder auch Absätze und Sätze von Abschnitten bearbeiten, also auch lokal speichern. Folglich ist der Feinheit der Granularität der Replikation fast keine Grenze gesetzt. Nach einer Wiederverbindung aber mit dem Server auf dem die Mole-DDE läuft (siehe Abschnitt 4.1) muss das ganze Kapitel, das geändert werden soll, geladen werden und die Änderungen müssen dort eingetragen werden. Schliesslich muss sogar das Kapitel, in dem auch nur ein einziges Wort geändert wurde, gespeichert werden.

Die Granularität der Replikation für die Mole-DDE ist also fein, was die Mitnahme der Daten angeht, was die Reintegration aber der Replikate in die bereits existierende Mole-DDE Struktur angeht, ist die Granularität sehr grob.

Die dritte Forderung nach Skalierbarkeit beruht auf Erfahrungen mit bestehenden statischen Replikationsstrategien, die auf nur wenige Replikate ausgelegt sind. Im mobilen Umfeld ist jedoch mit einer großen Anzahl an Replikaten zu rechnen, da auf jedem Laptop oder später auch jedem Palmtop ein Replikat benötigt wird. Die statischen Replikationsstrategien skalieren jedoch nur noch sehr schlecht, wenn viele Replikate benötigt werden und sind somit nicht sinnvoll anwendbar im mobilen Bereich. Die besser skalierenden Systeme mit Read-only Strategie würden hier zwar helfen, schränken jedoch die Benutzer zu sehr ein. Die Mole-DDE setzt, was die Skalierbarkeit der Replikation angeht, theoretisch keine Grenzen. Es können gleichzeitig sehr viele Replikate existieren. DDE-Systeme sollen die Arbeit von Teams (siehe Abschnitt 2.1.2) unterstützen. Da diese selten aus mehr als zehn Personen bestehen, sind meiner Meinung nach Replikationsstrategien, die eine solche Größenordnung unterstützen, vollkommen ausreichend.

Unter der Anforderung Mobil-Transparent und mit niedrigem Wartungsaufwand verstehen die Autoren folgendes, die Benutzer sollten nämlich nichts davon merken, dass für die Mobilität zusätzlicher Aufwand aufgebracht werden muss. Das System sollte sich soweit wie möglich so verhalten, als ob der mobile Rechner nicht mobil wäre. Es sollte auch kein regelmäßiges Eingreifen eines Administrators nötig sein, da diese in der Offline-Zeit nicht zur Verfügung stehen. Das System muss deshalb ohne Eingriffe laufen können. Die Anforderung, dass die Replikation ohne promotion-actions auskommen sollte, beruht darauf, dass nach Meinung der Autoren nicht immer im Voraus geplant werden kann, wann man offline arbeitet oder den Ort wechselt. In [RaRePo97] wird dies auch als 'Chaos of real life' bezeichnet. Der Begriff promotion-actions bedeutet dabei, dass vor einer Abkopplung explizit zusätzlicher Aufwand getrieben wird, um die mobile Arbeit zu ermöglichen. Zu diesen Forderungen möchte ich anfügen, dass sie teilweise gerechtfertigt, teilweise aber auch ungerechtfertigt scheinen. Dass kein regelmäßiges Eingreifen des Administrators nötig sein sollte, ist unbestritten. Offline gehen (siehe auch Abschnitt 2.2.2) aber ohne promotion-actions würde einen sehr hohen technischen Aufwand mit hoher Komplexität bedeuten

und das Ergebnis wäre dabei auch sehr zweifelhaft. Dieser Aspekt wird noch im nächsten Abschnitt genauer beleuchtet.

5.1.2 Gewollte und ungewollte Trennung vom System

An mehreren verschiedenen Stellen in der Literatur wird darauf hingewiesen, dass zwischen gewollten und ungewollten Trennungen vom System zu unterscheiden ist. Bei gewollten Trennungen entscheidet sich der Benutzer, dass er sich jetzt vom System abkoppeln und autonom weiterarbeiten möchte. Dies trifft vor allem für mobile Benutzer zu, die im Voraus wissen, dass sie die nächsten Stunden, Tage oder Wochen mobil und ohne Verbindung zum System arbeiten wollen. Unter ungewollten Trennungen versteht man hingegen Trennungen des Benutzers vom System aufgrund von technischen Problemen, wie Netzwerkunterbrechungen oder Serverausfällen.

Damit ungewollte oder spontane Trennungen ohne negative Folgen für den Benutzer bleiben, ist laut [BeMi99] eine Replikation der notwendigen Daten auf jedem Rechner während der gekoppelten Arbeit nötig. Dies bedeutet einen viel höheren Aufwand, da viel mehr Replikate dauernd existieren müssten und die Konsistenz wäre viel schwieriger zu erhalten. Die Mole-DDE beruht jedoch in der bisherigen Version auf einer Ein-Server-Lösung ohne Replikate und eine diesbezügliche Änderung wäre eine tiefgreifende Konzeptänderung, die nicht gewollt ist. Deshalb wird die Mole-DDE auch weiterhin nur für gewollte Trennungen eine Möglichkeit zur Replikation der Daten während der Abkopplung zur Verfügung stellen. Dies bedeutet, dass der Benutzer, bevor er autonom arbeiten kann, die benötigten Bücher oder Kapitel auf seinen mobilen Rechner kopieren muss. Der Nachteil hiervon ist natürlich, dass eine optimale Vorhersage der benötigten Bücher oder Kapitel nicht immer möglich sein wird und der Benutzer deshalb im Zweifelsfall mehr kopieren muss als er später wirklich benötigt.

5.1.3 'Single System Image' und Transparenz

Verteilte Systeme versuchen meist, dem Benutzer ein sogenanntes 'Single System Image' zu bieten. Dies bedeutet, sie versuchen die Verteilung der Daten und Ressourcen für den Benutzer unsichtbar zu halten. Der Benutzer greift auf alle Daten zu, als wären sie lokal verfügbar. Das System erledigt die eventuell notwendigen Zugriffe über das Netzwerk auf andere Rechner ohne Zutun des Benutzers. Für die Benutzer erscheint das Netzwerk somit transparent. Im Idealfall merken die Benutzer nicht, ob die Daten wirklich lokal vorhanden sind oder von anderen Rechnern stammen. 'Single System Image' legt auch sehr großen Wert darauf, dass keine unterschiedlichen Versionen entstehen können. Hierfür wird vom System ein eventuell sehr großer Aufwand aufgebracht, damit der Benutzer keinesfalls zwei verschiedene Versionen eines Files sieht. Wiederum bereitet dies bei mobilen Rechnern große Probleme, da während der mobilen Arbeit unterschiedliche Versionen entstehen können.

'Single System Image' funktioniert, solange das Netzwerk ständig verfügbar ist und genügend Bandbreite bietet, sehr gut. Im Bereich von mobilen Benutzern ist dies jedoch unmöglich. [EbSa97] sprechen deshalb davon, dass für mobile Rechner die Illusion der Transparenz nicht aufrecht erhalten werden kann. Stattdessen plädieren sie dafür, dass die Replikation der Daten für den Benutzer 'lichtdurchlässig' gemacht werden soll, d.h. der

Benutzer soll Einfluss nehmen können, welche Daten vom System auf dem mobilen Rechner repliziert werden.

[HoJo93] vertritt deshalb einen, wie er es bezeichnet, anderen philosophischen Ansatz. Hierbei soll der Benutzer wissen, dass es Replikate gibt und dass diese eventuell unterschiedlich sein können. Er begründet dies mit drei Argumenten gegen das 'Single System Image':

- 'Single System Image' - Systeme benötigen einen hohen Kommunikationsaufwand, um das 'Single System Image' zu erhalten. Dies bedeutet, dass eine gute Netzanbindung nötig ist, was im Fall von mobilen Rechnern aber unmöglich ist.
- 'Single System Image' - Systeme bieten 100%ige Konsistenz auch wenn dies eventuell gar nicht nötig ist. Und der Benutzer hat darüber keinerlei Kontrolle.
- Riesiger Aufwand um eventuell auftretende Verbindungsfehler zu behandeln.

Stattdessen sieht [HoJo93] das Gesamtsystem als eine Menge von autonom arbeitenden Rechnern, die hin und wieder miteinander kommunizieren, um die entstandenen unterschiedlichen Versionen der Files wieder zu einer Version zu vereinen. Dabei soll der Benutzer, der ja von den verschiedenen Versionen wissen soll, auch aktiv an deren Wiedervereinigung mitarbeiten. Die Philosophie von [HoJo93] ist auch meine Philosophie und wird im Mittelpunkt für die Entwicklung des neuen KESS stehen.

5.1.4 Anwendungs-transparent oder 'Application-aware'

In [NoSa95] wird zwischen 'application-transparent' und 'application-aware-adaption' unterschieden. Dies bedeutet, dass im ersten Fall die Anwendung, in unserem Fall also der Editor, nicht wissen muss, ob der Benutzer momentan mobil arbeitet oder nicht. Wenn sich die Applikation jedoch unterschiedlich verhalten soll, je nachdem ob der Benutzer gerade mobil arbeitet oder nicht, dann muss sie diese Zustände unterscheiden können, indem sie Informationen über den aktuellen Zustand aus dem System abfragt und darauf entsprechend reagiert. [NoSa95] sieht den größten Vorteil von application-transparent in dessen Abwärtskompatibilität. Alle bestehenden Applikationen können hierbei ohne Änderungen weiter funktionieren.

In Mole-DDE muss eine 'application-aware-adaption' existieren, um den Editor an den Mole-DDE Server anzuschließen. Gemeint ist hiermit der Wrapper (siehe Abschnitt 4.6), ohne dessen Funktionalität die Verbindung jeglichen Editors und DDE-Systems nicht möglich ist. Für die Verbindung muss aber zuerst der Benutzer sorgen, der bewusst den Editor mit dem Wrapper starten muss, falls er gekoppelt mit dem DDE-System arbeiten möchte.

5.2 Konsistenzerhaltungsverfahren

Dieser Abschnitt fasst die in der Literatur gefundenen Aspekte zum Thema der Replikation und Konsistenzerhaltung zusammen. Dabei werden sowohl verschiedene Replikationsverfahren als auch Verfahren zur Erhaltung der Konsistenz oder wenn dies nicht möglich ist, zur Erkennung von Inkonsistenzen beschrieben.

5.2.1 Replikationsstrategien

Es wird allgemein ([ImBa93], [ReHeRa94], [Sör96], [BeMi99]) zwischen zwei grundsätzlich verschiedenen Replikationsstrategien, den pessimistischen und den optimistischen, unterschieden. Genau genommen wird dabei weniger die eigentliche Replikation betrachtet, als vielmehr das Verhalten der Systeme bezüglich eventuell konkurrierender Schreibzugriffe auf die einzelnen Replikate.

Abbildung 5.1 stellt einige wichtige Replikationsstrategien im Überblick dar. Nach [ReHeRa94] führt die Replikation änderbarer Files praktisch automatisch zu Konsistenzproblemen. Durch die Möglichkeit der Partitionierung von Replikaten kommt es zu Konflikten bei Änderungen an verschiedenen Replikaten.

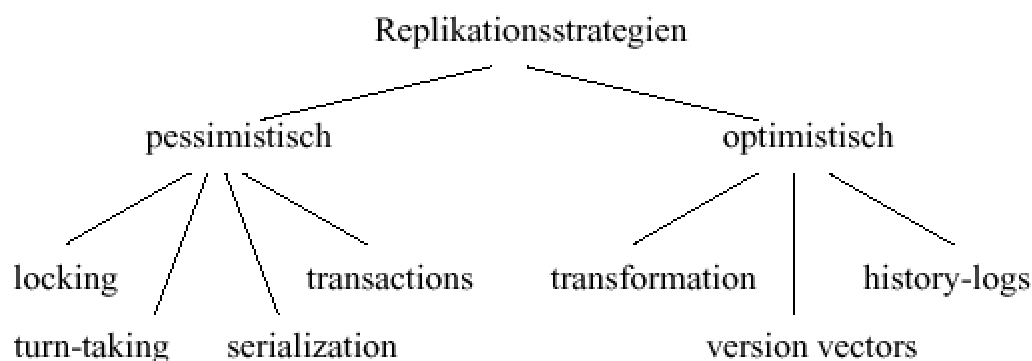


Abbildung 5.1: Replikationsstrategien

Die pessimistischen Replikationsstrategien betrachten das Auftreten von Inkonsistenzen als häufig auftretendes kritisches Ereignis und versuchen dieses unter allen Umständen zu vermeiden. Dies gelingt ihnen jedoch nur durch rigorose Maßnahmen, die schon im Falle der Gefahr, dass eventuell eine Inkonsistenz entstehen könnte, eine Änderung nicht zulassen.

Im Gegensatz dazu haben optimistische Strategien als erstes Ziel die Verfügbarkeit. Diese nehmen für die jederzeitige Verfügbarkeit der Replikate eventuelle Inkonsistenzen in Kauf und sehen Möglichkeiten zur Auflösung der Inkonsistenzen in Form einer Reintegration vor. Der Vorteil der erhöhten Verfügbarkeit muss abgewogen werden gegen die Zahl der Konflikte und die Kosten, um diese zu beheben [ReHeRa94]. Dies ist insbesondere deshalb wichtig, weil der Vorteil der autonomen Arbeit nicht durch den Aufwand der Reintegration zunichte gemacht werden darf [BeScVö96].

[ReHeRa94] gehen anhand von Messungen von einer geringen Zahl an Konflikten aus und bezeichnen deshalb die Kosten der optimistischen Strategien in vielen Umgebungen als gering. Sie begründen das unter anderem damit, dass beim typischen Zugriffsverhalten ein File nur von wenigen Benutzern geändert wird und somit wenige Konflikte auftreten. Sie sehen es deshalb als großen Nachteil an, wenn das Ändern eines Files verboten wird, nur weil vielleicht jemand anders dieses File gerade ändern könnte.

Der Vollständigkeit halber sei hier noch kurz darauf hingewiesen, dass zwischen den beiden Extremen pessimistischer oder optimistischer Replikationsstrategien auch Zwischenlösungen existieren. So präsentiert [Sör96] eine Zwischenlösung bei der die Applikation

oder der Benutzer eventuelle Konflikte als verschieden kritisch einstufen kann und entsprechend optimistische oder pessimistische Wege gewählt werden können.

Dabei wissen die Applikation oder der Benutzer von den möglichen Inkonsistenzen und können fallweise zwischen optimistischem oder pessimistischem Vorgehen entscheiden. Im Idealfall weiß die Applikation wie genau die Daten sein müssen und kann ein entsprechendes Vorgehen wählen. Oftmals reicht es beispielsweise aus, mit veralteten Daten zu arbeiten. In einem solchen Fall ist Konsistenz nicht unbedingt nötig und entsprechend ein optimistisches Vorgehen angebracht.

5.2.2 Konflikterkennung

Im Rahmen der Literaturrecherche wurden auch Artikel über die Erkennung von Konflikten untersucht. Die hierbei aufgefundenen Algorithmen sind sehr unterschiedlich, da sie für spezielle Einsatzgebiete entworfen oder optimiert wurden.

[BeMi99] unterscheidet drei Möglichkeiten, über die Inkonsistenzen erkannt werden können:

- Über einen Vergleich der Daten selbst
- Über einen Vergleich von zusätzlich gehaltenen Informationen und
- Über einen Vergleich der ausgeführten Operationen.

Im folgenden wird auf die Erkennung anhand einer Informationsbasis am Beispiel der 'Version Vectors' und auf die Erkennung anhand einer Historien-Aufzeichnung eingegangen. Die verschiedenen Möglichkeiten unterscheiden sich außerdem darin, ob sie die Inkonsistenzen nur erkennen oder auch Anhaltspunkte für die Reintegration liefern.

Als Vertreter der reinen Erkennungs-Algorithmen sollen hier die sogenannten 'version vectors' erwähnt werden. 'Version vectors' oder Varianten davon werden in vielen optimistischen Replikationsstrategien zur Erkennung von Inkonsistenzen verwendet [RaRePo97b]. Sie werden in Systemen eingesetzt, in denen jedes Replikat die Änderungen, die auf ihm ausgeführt werden, über eine Änderungsmitteilung an alle anderen Replikate verteilt. Dabei zählt jedes Replikat mit, wieviele Änderungen es von allen anderen Replikaten bereits erhalten hat. Wenn alle Replikate jeweils gleich viele Änderungen der jeweils anderen Replikate empfangen und nach eventueller Transformation der Position in das eigene Replikat eingearbeitet haben, dann ist die Konsistenz gewährleistet.

Abbildung 5.2 zeigt die Wirkungsweise der Version Vectors anhand dreier Benutzer, die verschiedene Replikate eines File editieren. Jeder Benutzer besitzt ein Replikat des Files und einen daneben dargestellten Versions-Vektor. Das erste Feld des Versions-Vektors gibt dabei die Zahl der Änderungen an, die von Benutzer 1 ausgehend beim jeweiligen Replikat angekommen sind. Entsprechendes gilt für die beiden anderen Felder. Jeder Pfeil stellt die Übertragung einer Änderung dar. Der kurze Pfeil stellt eine verloren gegangene Änderung dar.

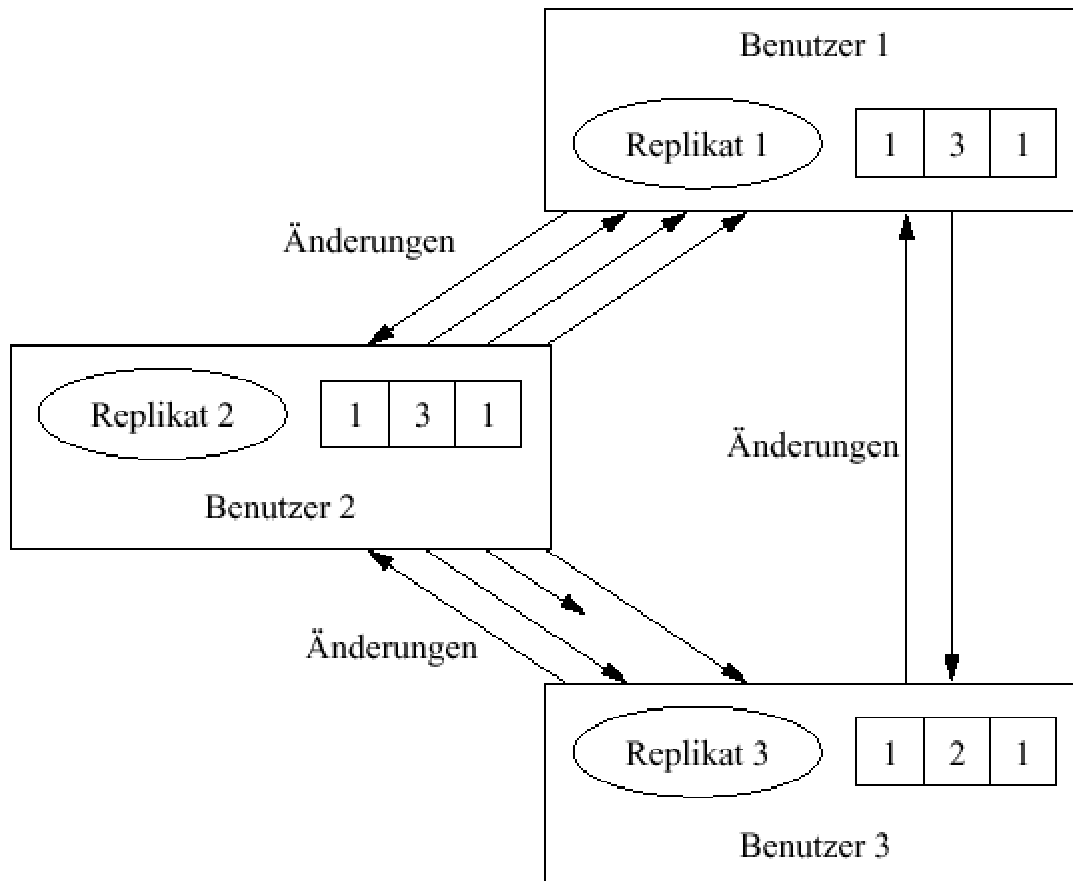


Abbildung 5.2: Konflikterkennung durch Version-Vektors

Benutzer 1 und 3 haben bisher je eine Änderung vorgenommen, die auch bereits an die jeweils anderen Benutzer weitergeleitet wurden. Benutzer 2 hat 3 Änderungen vorgenommen, die auch bei Benutzer 1 angekommen sind. Bei Benutzer 3 hingegen sind erst 2 dieser 3 Änderungen angekommen. Wenn zu diesem Zeitpunkt nun geprüft werden soll, ob die Replikate von Benutzer 1 und 2 konsistent sind, dann werden die Versionsvektoren dieser beiden Benutzer verglichen. Da sie beide 1-3-1 lauten, sind die Replikate konsistent. Wenn jedoch ein Vergleich zwischen den Benutzern 1 und 3 gemacht wird, dann fällt auf dass die Versionsvektoren einmal 1-3-1 und einmal 1-2-1 lauten und somit nicht übereinstimmen. An der niedrigeren Anzahl an Änderungen bei Benutzer 3 kann man außerdem erkennen, dass dieser veraltet ist.

Version Vectors besitzen den Vorteil, dass mit ihrer Hilfe sehr schnell entschieden werden kann, ob zwei Replikate konsistent sind oder nicht. Bei Inkonsistenzen erlauben sie jedoch keine Aussage über die Art der Inkonsistenz und geben daher keinen Ansatzpunkt für die Reintegration.

Die Historie Aufzeichnung stellt einen anderen Algorithmus dar, mit dem nicht nur Inkonsistenzen erkannt werden können, sondern sie liefert zusätzlich Anhaltspunkte für die Reintegration der Replikate. Beim Vergleich anhand von Historie-Aufzeichnungen (siehe [BeScVö96] oder [BeMi99]) kann aus den Aufzeichnungen direkt entnommen werden, was eingefügt oder gelöscht wurde. Somit können hieraus auch sofort die nötigen Maßnahmen zur Reintegration bestimmt werden.

Das Prinzip hierbei ist, dass jedes Replikat die Änderungen, die von lokalen Benutzern vorgenommen wurden, an alle anderen verteilt. Statt jedoch nur zu zählen, wieviele Änderungen von welchem Replikat empfangen wurden, führt nun jedes Replikat ein Protokoll, in dem alle Änderungen aufgezeichnet werden. Dabei werden sowohl die lokal entstandenen als auch die von anderen Replikaten empfangenen Änderungen erfasst. Durch den Vergleich zweier solcher Historie-Aufzeichnungen kann dann entschieden werden, ob zwei Replikate konsistent sind. Dies ist jedoch rechen- und speicherintensiver als die version vectors, weil die Positionen, an denen Änderungen vorgenommen werden, abhängig sind von schon vorgenommenen Änderungen, so dass die Positionen häufig neu bestimmt werden müssen.

Abbildung 5.3 stammt aus [BeScVö96] und stellt die Verwendung von Historie-Aufzeichnungen anhand zweier Benutzer dar. Ausgehend von der ganz links dargestellten Version 0 nehmen die Benutzer A und B unabhängig voneinander Änderungen vor, die zu den inkonsistenten Versionen A1 und B1 führen. Im Rahmen der Reintegration erstellt der Reintegrations-Algorithmus aus den Historie-Aufzeichnungen SA1 und SB1 zwei Listen SA2 und SB2 von Änderungen, die auf die jeweiligen Replikate angewandt, zur wieder konsistenten Version 2 ganz rechts führen.

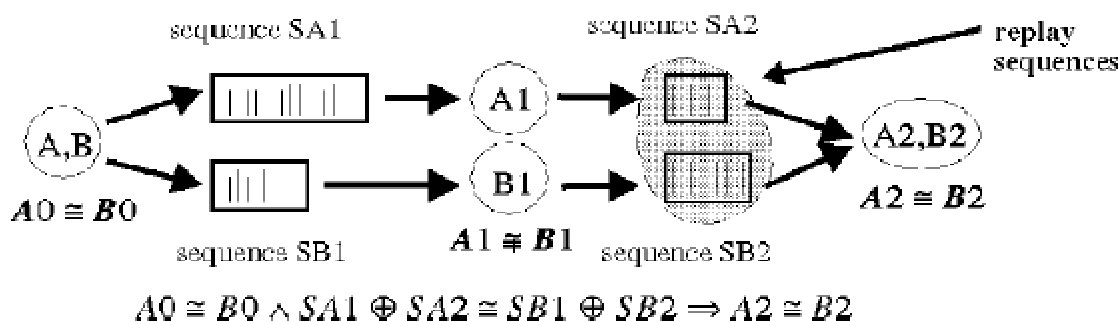


Abbildung 5.3: Historie-Aufzeichnung

In [BeScVö96] werden vier Vorteile der Historie-Aufzeichnung gegenüber den anderen beiden Algorithmen aufgezählt:

- effizienteres Erkennen von Inkonsistenzen, da weniger Daten verglichen werden müssen,
- Problem der Erkennung von remove oder insert existiert nicht
- letzte gemeinsame Version wird nicht benötigt
- Historie-Aufzeichnungen können verkürzt werden, wenn Teile der Replikate zusammengeführt wurden.

In der existierenden Mole-DDE ist eine Art von Anwendung der Historie-Aufzeichnung möglich. Der Editor und der Wrapper unterstützen zwar einen solchen Mechanismus nicht, durch die Kombination aber der Mensch-Editor- und Editor-DDE-Schnittstelle wird ein Pseudo-Historie-Mechanismus angeboten. Um das ganze verständlicher zu gestalten, schauen wir uns ein einfaches Beispielszenario an.

Ein Teamteilnehmer, der in den nächsten zwei Wochen ein Konferenz besuchen soll, will in dieser Zeit an einem Buch arbeiten. Er verbindet sich mit der DDE und kopiert das Buch lokal auf seinen mobilen Rechner. Wenn er von seinem Konferenzort aus seine Änderun-

gen wirksam machen möchte, geht er ins Netz und stellt wieder die Verbindung zur DDE her. Der Benutzer selber ist jetzt derjenige, der die Änderungs-Historie aufgezeichnet hat und sie der DDE überträgt. Hat er z.B. einen Abschnitt gelöscht und einen neuen angelegt, muss er in der gleichen Reihenfolge seine Änderungen dem DDE-System bekannt machen. Die entsprechenden Informationen werden über die Schnittstelle zwischen dem Editor und der Mole-DDE übertragen (siehe Abschnitt 4.7). Einen Abschnitt löschen, würde als Ergebnis eine Informations-Einheit der Art:

<BOOKNAME><E2WDESTROYSECTION><SECTIONNAME> haben.

Entsprechend hat das Anlegen eines neuen Abschnitts als Ergebnis die Informations-Einheit: <BOOKNAME><E2WCREATESECTION><SECTIONNAME>.

Diese Informationen der Schnittstelle zwischen Editor und DDE-System kann man als Historie-Aufzeichnung verwenden.

5.3 Reintegration

Die Erkennung von Inkonsistenzen bei optimistischen Replikationsstrategien ist die Voraussetzung zur Behebung von Inkonsistenzen. Wie für die Erkennung, so existieren auch für die Behebung der Inkonsistenzen verschiedene Möglichkeiten. Ziel ist jedoch jedes Mal, die einzelnen inkonsistenten Replikate zu einem File zusammenzufassen, welches dann wiederum als Ursprungsreplik verwendet werden kann.

In diesem Abschnitt wird untersucht welche Replikate ihre Änderungen zusammenführen dürfen und ob dabei eine Benutzerbeteiligung erlaubt oder sogar notwendig ist. Des Weiteren wird eine Klassifikation der auftretenden Inkonsistenzen vorgestellt und die Schwierigkeitsgrade der einzelnen Klassen werden erläutert.

5.3.1 An Reintegration beteiligte Replikate

Als erstes soll eine Unterscheidung der Reintegrationsalgorithmen anhand der beteiligten Replikate getroffen werden. [RePoGu96] unterscheidet hier zwischen Master-Slave- und Peer-to-Peer-Reintegration.

Eine Master-Slave-Struktur schreibt eine hierarchische Trennung der Replikate vor. Es existieren ein oder mehrere als Master bezeichnete Replikate, auf denen keine Änderungen erlaubt sind. Diese sind die Empfänger der auf den sogenannten Slave-Replikaten stattgefundenen Änderungen. Eine Reintegration ist nur zwischen ein Master- und einem Slave-Replikat möglich. Nach [RePoGu96] bedeutet dieses Verfahren einen minimalen Aufwand für den Reintegrationsalgorithmus besonders, wenn nur ein zentrales Master-Replikat vorhanden ist. Falls mehrere Master-Replikate existieren, wird ein Algorithmus benötigt, welcher die Änderungen der gleichberechtigten Master-Replikate zusammenführen kann.

Peer-to-peer Reintegration verwendet nur gleichberechtigte Replikate, so dass jedes Replikat mit jedem anderen eine Reintegration vornehmen kann. Die hierzu nötigen Algorithmen zur Kontrolle der Reintegration und der Updates sind deutlich komplexer als bei der Master-Slave-Reintegration. Dem steht jedoch gegenüber, dass auch ohne Verbindung zu einem Master-Replikat eine Reintegration der Änderungen zweier Replikate direkt stattfinden kann. Es fällt dabei nicht nur der Flaschenhals des Servers, auf dem sich das Master-Replikat befindet, weg, sondern es kann eventuell auch Kommunikationsaufwand

gespart werden. Ich möchte hier betonen, dass eine Master-Slave-Struktur nicht mit der Client-Server-Architektur fest verbunden ist. Eine Client-Server-Architektur kann sowohl die Master-Slave-, als auch die Peer-to-Peer-Reintegration unterstützen. Dieser Sachverhalt wird im folgenden Beispielszenario erläutert.

5.3.2 Beispielszenario

Diesen geringeren Kommunikationsaufwand beschreibt [RePoGu96] in einem Szenario, in dem zwei Benutzer während ihrer Reisen auf ihrem mobilen Rechner jeweils ein Replikat eines Files verändert haben. Diese Benutzer treffen sich nun auf einer Tagung weit entfernt vom ursprünglichen Server des Master-Replikats. Im Falle der Master-Slave-Struktur müsste jeder dieser Benutzer eine Verbindung zum weit entfernten Rechner aufbauen und über diese wahrscheinlich langsame Verbindung eine Reintegration der Replikate vornehmen. Dabei müssten die Änderungen beider Benutzer zum Server geschickt werden.

Änderungen eines dritten Benutzers müssten jeweils für die Benutzer 1 und 2 über die langsame Leitung transportiert werden. Außerdem müsste jeder der Benutzer 1 und 2 eventuell zweimal eine Verbindung zum Server aufbauen, um beim zweiten Vergleich die Änderungen des jeweils anderen Benutzers zu erhalten, nachdem dieser sie an den Server übertragen hat. Alternativ können die Benutzer 1 und 2 die Verbindung natürlich auch offen halten, während die Änderungen des jeweils anderen Benutzers übertragen werden. Allerdings entstehen dadurch wieder höhere Online-Kosten. Abbildung 5.4 stellt dieses Beispielszenario graphisch dar.

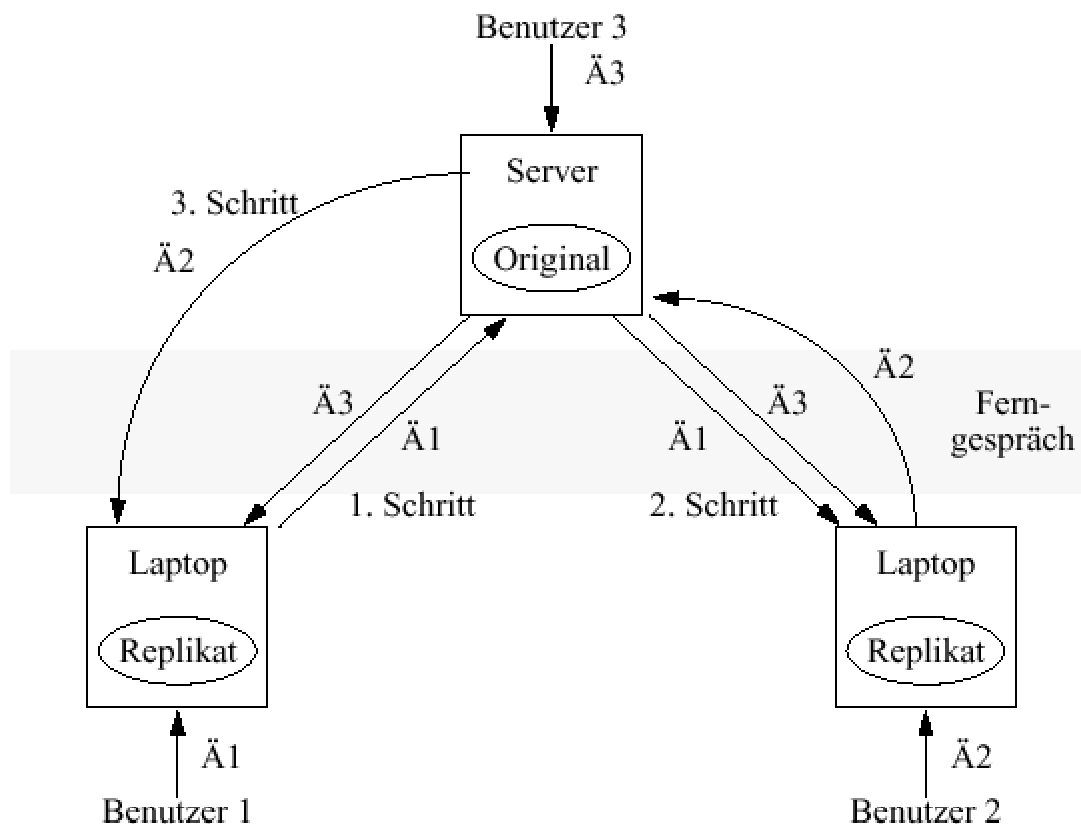


Abbildung 5.4: Client-Server Reintegration

Bei einer Peer-to-Peer Struktur hingegen kann jeder Benutzer direkt die Änderungen des anderen in sein Replikat reintegrieren und es müssen nur einmal ihre gesammelten Änderungen an den Server übertragen werden. Entsprechend müssen auch nur einmal die Änderungen des dritten Benutzers zu den beiden Benutzern über die langsame Leitung übertragen werden. Dafür ist dann auch hier wieder ein zweites Update diesmal zwischen Benutzer 1 und 2 nötig, um die Änderungen von Benutzer 3 dem jeweils anderen bekannt zu machen. Abbildung 5.5 stellt die hierfür nötigen Schritte dar.

Wie aus den Abbildungen deutlich zu sehen ist, müssen im zweiten Fall deutlich weniger Daten zum weit entfernten Rechner über die langsame und teure Leitung übertragen werden.

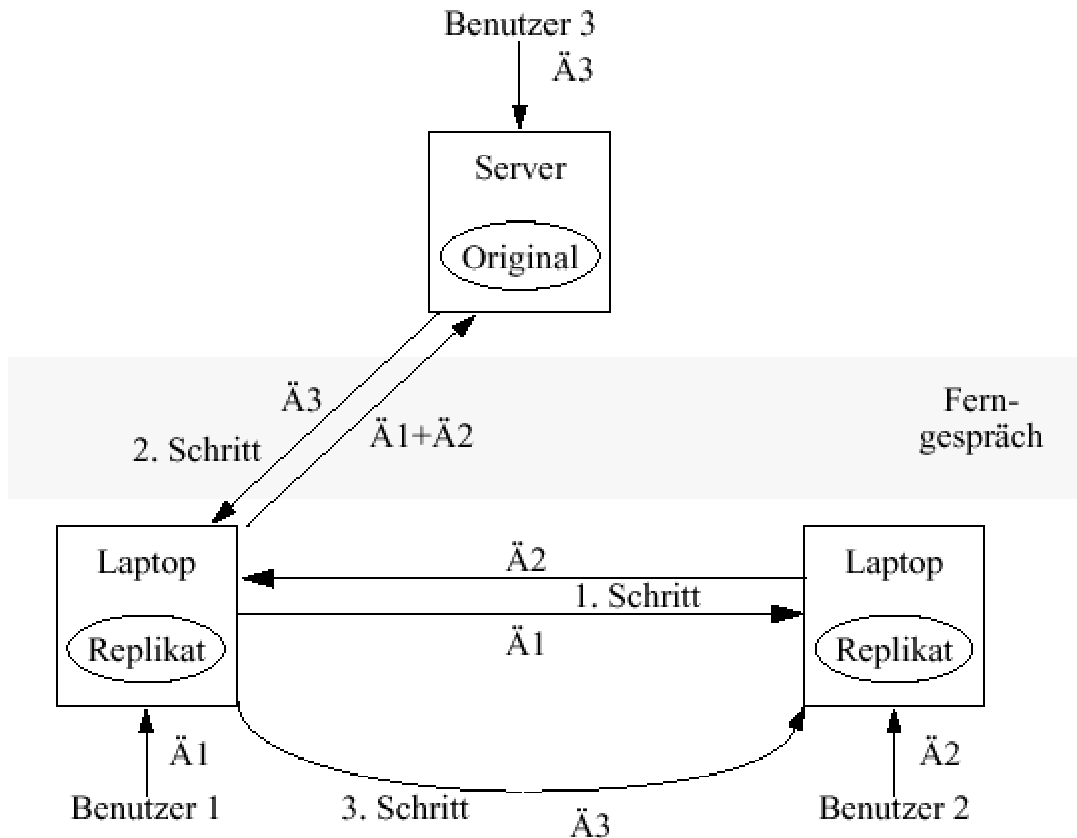


Abbildung 5.5: Peer-To-Peer Reintegration

Wie man noch aus der Abbildung 5.5 entnehmen kann, existiert auch in diesem Fall eine Client-Server-Architektur. Sie verpflichtet uns aber nicht zu einer Master-Slave-Reintegration.

Unsere Mole-DDE unterstützt zwar nicht direkt, verbietet aber auch keine Peer-to-Peer-Reintegration. Es ist genauso möglich, dass die beiden Benutzer 1 und 2 ihre Änderungen gegenseitig austauschen und sie in ihre Replikate einbringen, um sie danach dem Server bekanntzugeben. Umgekehrt brauchen nicht beide ein Replikat vom Server zu verlangen, sondern es kann sich nur der eine die Daten holen und dann auch zu dem anderen übertragen.

5.3.3 Zeitpunkt der Reintegration

Als nächstes unterscheidet wieder [RePoGu96] zwischen 'reconciliation-based' und 'update-propagation-based' Reintegration. Bei update-propagation-based Reintegration werden die Änderungen sofort nach dem Entstehen an alle Replikate verteilt. Wenn einige davon nicht erreichbar sind, bedeutet dies einen zusätzlichen Aufwand und Inkonsistenzen. Dieser zusätzliche Aufwand stellt für mobile Umgebungen natürlich ein Problem dar, da die mobilen Rechner häufig nicht erreichbar sind.

Bei reconciliation-based Reintegration werden Updates nicht sofort verteilt, sondern entweder regelmäßig, oder sie werden bei Bedarf als Batch aller Änderungen seit der letzten Reintegration verbreitet. Hierbei können zurückgenommene Änderungen oder bereits wieder geänderte Änderungen zusammengefasst werden, was die Netzwerklast reduziert, Zeit und letztlich Geld spart. Allerdings führt dies dazu, dass die anderen Replikate zwischenzeitlich mit veralteten Daten arbeiten müssen. Zur Entscheidung, zu welchem Zeitpunkt die Reintegration stattfinden soll, muss man abwägen, was für den konkreten Anwendungsfall sinnvoller ist.

Mole-DDE unterstützt mobile Rechner. Daher kommt eine update-propagation-based Reintegration nicht in Frage.

5.3.4 Benutzerbeteiligung bei Reintegration

Das letzte Unterscheidungskriterium ist das, ob eine Beteiligung der Benutzer bei der Reintegration nötig ist oder nicht. Bei Systemen mit automatischer Reintegration wird versucht, die Reintegration ohne Beteiligung des Benutzers möglichst vollständig automatisch durchzuführen.

Ein Beispiel hierfür sind die Resolver im Ficus-Filesystem [ReHePa94]. Diese Resolver beheben für eine kleine Gruppe von Files Inkonsistenzen völlig ohne Zutun des Benutzers. Möglich ist dies, weil diese Gruppe von Files eine klare Syntax und Semantik hat und für eine Reintegration einfache Regeln existieren. In diese Gruppe fallen z.B. Highscore-Files von Spielen oder auch die '.newsrc' Files von Newsreadern. Das Problem dieser Algorithmen ist, dass sie auf ein ganz spezielles Anwendungsgebiet optimiert sind und dabei zusätzliches Wissen oder Regeln aus dem jeweiligen Themengebiet verwenden können.

Das Gegenstück zu diesen automatisch arbeitenden Reintegrationsalgorithmen bilden Algorithmen, die eine Mitarbeit des Benutzers bei der Reintegration erfordern. So überlässt z.B. das oben erwähnte Ficus-Filesystem die Reintegration allgemeiner Files dem Benutzer, da hierfür kein Resolver geschrieben werden konnte.

Das zu entwickelnde KES ist für allgemeine Texte gedacht. In diesem Fall ist die Formulierung von Regeln keinesfalls trivial. Trotzdem werden wir im weiteren Verlauf dieser Arbeit versuchen, solche Regeln zu formulieren. In Fällen, in denen keine Regel formuliert werden kann, wird ein Eingreifen der beteiligten Benutzer zugelassen.

5.4 Zusammenfassung

In diesem Kapitel wurden die wichtigsten Aspekte eines Konsistenzerhaltungssystems vorgestellt, ergänzt durch kritische Stellungnahmen bezüglich der bereits existierenden Mole-DDE.

Wir haben gesehen, dass für das zu entwerfende KES:

- nur optimistische Replikationsalgorithmen in Frage kommen,
- eine möglichst feine Granularität der Replikation angestrebt werden sollte,
- die Skalierbarkeit der Replikate gewährleistet sein muss, obwohl wir nicht an einer besonders hohen Zahl von Replikaten interessiert sind,
- ungewollte Trennungen vom System nicht unterstützt werden können,
- ein 'Single System Image' nicht möglich ist und auch nicht das Ziel sein sollte, sondern das System wird als eine Menge autonom arbeitender Rechner betrachtet,
- die verschiedenen Editoren, die zum Einsatz kommen, sollten über den gekoppelten oder entkoppelten Status informiert werden,
- eine Konflikterkennung durch die ausgeführten Operationen möglich sein sollte, ähnlich zur Historie-Aufzeichnung,
- Peer-to-Peer-Reintegration möglich sein sollte,
- der Zeitpunkt der Reintegration frei wählbar sein sollte und
- eine Benutzerbeteiligung bei der Reintegration unvermeidbar ist.

Im nächsten Kapitel folgen eine Anforderungsanalyse und eine Spezifikation unseres Konsistenzerhaltungssystems. Die oben erwähnten Punkte sind Anhaltspunkte für unser weiteres Vorgehen in dieser Arbeit, sie stellen bei kritischen Fragen bezüglich der Entwicklung unseres Systems die Weichen.

6. Anforderungsanalyse und Spezifikation

In diesem Kapitel werden wir uns mit der Anforderungsanalyse und der Spezifikation unseres Konsistenzerhaltungssystems beschäftigen. Die Anforderungsanalyse dient dazu, die existierende Situation (Ist-Zustand) zu untersuchen und festzulegen. Im Gegensatz dazu zielt die Spezifikation darauf ab, die erwünschte Situation (Soll-Zustand) möglichst genau zu spezifizieren.

Dieses Kapitel unterteilt sich daher in zwei Teile:

1. **Ist-Zustand:** es wird eine Bestandaufnahme des existierenden Systems gemacht, dabei werden uns insbesondere Fragen der Konsistenzerhaltung beschäftigen.
2. **Spezifikation (Soll-Zustand):** hier wird die Wunsch-Situation geschildert. Wie soll das DDE-System in Bezug auf verschiedene Inkonsistenz-Szenarien aussehen? Dabei werden uns insbesondere die Inkonsistenz-Szenarien, die bereits im Abschnitt 2.2.2. beschrieben wurden, beschäftigen.

6.1 Ist-Zustand

Die bereits existierende Mole-DDE wurde schon im Kapitel 4 beschrieben. Hier werden wir unseren Blick speziell auf Konsistenzfragen richten. Möchte ein Benutzer mit der Mole-DDE gekoppelt arbeiten, muss er lokal auf seinem Rechner über die grafische Benutzungsoberfläche den XEmacs-Editor starten. XEmacs läuft jetzt verbunden mit dem DDE-Server.

Man kann hier zwei Fälle unterscheiden:

- 1) der Benutzer unternimmt Änderungen an der Dokumentenstruktur oder an dem Dokumenteninhalte direkt auf dem Server,
- 2) der Benutzer war bereits davor mit dem System verbunden, hat sich die Sachen, die ihn interessieren lokal auf seinen Rechner heruntergeladen und bearbeitet und möchte sie jetzt auch auf dem Server wirksam machen.

Im ersten Fall könnten gleichzeitig mit ihm auch ein oder mehrere andere Benutzer Änderungen vornehmen. Manche von diesen Änderungen machen seine unwirksam oder auch umgekehrt macht seine Arbeit die Arbeit der anderen unwirksam. Wenn die Benutzer ihren XEmacs herunterfahren, sind sie fest davon überzeugt, dass ihre Arbeit jetzt gespeichert auf dem Server liegt. Solange sie mit dem Server in Verbindung waren, konnten sie zwar auf dem Hauptfenster von Mole-Office verschiedene Meldungen über Änderungen an Dokumenten von anderen Benutzern sehen, diese Meldungen waren jedoch nicht genügend aufschlussreich, um ihnen den Erfolg bzw. Misserfolg ihrer Arbeit aufzuzeigen.

Im zweiten Fall gilt wieder, was bereits geschildert wurde. Zusätzlich könnten aber ein oder mehrere andere Benutzer die Dokumentenstruktur bereits geändert haben (siehe auch Abschnitt 2.2.2), solange der Benutzer die Dokumente lokal auf seinem Rechner bearbeitet hat.

Die obige Beschreibung verdeutlicht, dass die existierende Mole-DDE die Arbeit von mehreren Personen nicht optimal unterstützt. Sowohl für den Fall der Existenz von mehreren Replikaten (entkoppelte Arbeit) als auch für den Fall des gleichzeitigen Zugriffs auf den DDE-Server könnten Teile der Arbeit der Benutzer verloren gehen. Aus diesem Grund ist ein zusätzliches System nötig, das solche unerwünschten Szenarien, wie die in diesem Abschnitt beschriebenen, ausschließt.

Im Abschnitt 2.2.1 habe ich bereits verschiedene Konsistenz-Arten von Dokumenten mit Hilfe von Regeln definiert. Entsprechend müssen auch für unsere Mole-DDE Regeln definiert werden, die einzuhalten sind. Das für die Einhaltung dieser Regeln zuständige System stellt nach der Diskussion im Abschnitt 2.2.1 ein KES dar. Wie ein solches System aussehen könnte, werden wir im weiteren Verlauf dieses Kapitels sehen.

6.2 Soll-Zustand

Man kann die Anforderungen an ein KES aus verschiedenen Blickwinkeln betrachten, abhängig davon, an welchen Eigenschaften des Systems man interessiert ist. In diesem Kapitel werden wir die Anforderungen aus zwei verschiedenen Sichten betrachten, einmal aus der Sicht der Software-Qualität und einmal aus der Sicht der rein funktionellen Anforderungen. Die ersten zwei Abschnitte sind daher etwas allgemeiner und beschäftigen sich mit allgemeinen Anforderungen (qualitativen und funktionellen). Der zweite Abschnitt stellt eine Zusammenfassung der funktionellen Anforderungen an Konsistenzerhaltungssysteme, die in der Literatur gefunden wurden, dar. Im dritten Abschnitt dieses Kapitels, in der Spezifikation, werden dann die Anforderungen für unser System spezifiziert. Die Wunschsituation wird möglichst genau beschrieben.

6.2.1 Anforderungen aus der Sicht der Software-Qualität

Man könnte in diesem Abschnitt jedes Merkmal einer qualitativen Software aufzählen. Dazu möchte ich aber auf die relevante Software-Engineering-Literatur verweisen und hier nur einige wichtige Qualitätseigenschaften, die ein Konsistenzerhaltungssystem aufweisen soll, erwähnen.

Die Konsistenzerhaltung sollte von einer **Software-Schicht** realisiert werden, die zwar auf der Funktionalität der bereits existierenden Software aufbaut, bei der aber auf der anderen Seite ihre eigene Funktionalität möglichst von ihren eigenen Modulen und Prozeduren implementiert ist. Das bedeutet die Software-Schicht, die die Konsistenzerhaltung realisieren sollte, sollte die typischen Eigenschaften von Software-Modulen aufweisen, nämlich möglichst hohen Zusammenhalt ihrer eigenen Teile, das sind in diesem Fall ihre Module und Prozeduren, und eine möglichst geringe Kopplung mit der bereits existierenden Software.

Aus dieser Forderung heraus folgen noch einige Vorteile, die man auch als Anforderungen betrachten könnte. Das sind:

- **Wiederverwendbarkeit.** Eine solche in sich geschlossene Schicht könnte man mit relativ niedrigem Aufwand auch für andere Anwendungen verwenden.

- **Wartbarkeit.** Da möglichst viel der Funktionalität von der Schicht selber realisiert ist, kann man künftige Änderungen relativ einfach durchführen.
- **Erweiterbarkeit.** Diese beinhaltet, dass auch nach der Erstellung, notwendig werdende Funktionalitätserweiterungen in dieser Schicht vorgenommen werden können.

Der Vollständigkeit halber könnte man hier noch einige andere Merkmale, wie z.B. Testbarkeit, Struktur, Simplizität, Knappheit usw. erwähnen, diese sind aber für mich selbstverständlich und ich werde nicht näher darauf eingehen.

6.2.2 Anforderungen aus funktioneller Sicht

Anforderungen an ein KES, die aus funktioneller Sicht entstehen, haben mit der Erkennung und Behebung von verschiedenen Arten von Inkonsistenzen zu tun. Das sind aber nicht die einzigen Aspekte, die man betrachten sollte. Weitere wichtige Möglichkeiten, die ein solches System anbieten sollte, sind die Festlegung der Konsistenzkonditionen, des Zeitpunktes, wann Verletzungen entdeckt werden sollten usw. Laut [TaCI98] sollte ein KES mindestens folgende Anforderungen erfüllen:

- 1) **Definition von Konsistenzkonditionen:** Konsistenz-Management beginnt mit der Definition, was man überhaupt unter Konsistenz verstehen sollte. Anhand dieser Definition kann man mehrere mögliche Objekt-Zustände unterscheiden. Die gängigsten Zustände sind der "konsistente" und der "inkonsistente" Zustand, es könnten aber auch unterschiedliche Grade von Konsistenzen existieren. Ein Konsistenz-Management System sollte eine solche Unterscheidung, die je nach Fall sinnvoll erscheint, zulassen.
- 2) **Festlegen, wie und wann Verletzungen entdeckt werden sollten:** Entdeckungen von Verletzungen sind meistens operation-driven, d.h. sie finden während der Änderung der Objekte im Rahmen dieser Änderungen statt. Es sollte möglich sein solche Verletzungen auch auf eine andere Art und Weise zu entdecken; z.B. aufgrund einer Anfrage eines Klienten oder auch während des Verlaufs eines anderen Prozesses. Für verschiedene Arten von Objekten existieren unterschiedliche Konsistenz-Definitionen, deren Verletzungen auch zu unterschiedlichen Zeitpunkten entdeckt werden sollten. Es gibt z.B. Konsistenzbedingungen, die niemals verletzt werden sollten, was bedeutet, dass sogar eventuelle Verletzungen von solchen Regeln entdeckt werden sollten, bevor sie noch auftreten. Daneben gibt es auch solche Konsistenz-Regeln, die gleich nach ihrer Verletzung wieder gut gemacht werden sollten und solche, bei denen eine Verletzung für einige Zeit toleriert werden kann.
- 3) **Maßnahmen zur Konsistenzerhaltung spezifizieren:** Es sollte möglich sein, geeignete Maßnahmen für eventuelle Konsistenzverletzungen zu definieren. Diese Maßnahmen haben ein sehr weites Spektrum, das von der Ablehnung einer Inkonsistenz verursachenden Änderung über die Wiederherstellung des Zustands der Objekte, die von dieser Änderung betroffen waren, über weitere Änderungen dieser Objekte zu einem neuen konsistenten Zustand bis zur Zulassung des inkonsistenten Zustands reicht. Zusätzlich sollte man auch Maßnahmen für den Fall des Scheiterns von Wiedergutmachungsaktionen berücksichtigen.

- 4) **Inkonsistenz-Management:** Auch mit einer Fülle an Maßnahmen gegen Verletzungen von Konsistenzbedingungen kann eine sofortige Rückkehr zu einem konsistenten Zustand nicht möglich oder vielleicht sogar nicht erwünscht sein. Verwaltung der Inkonsistenz bedeutet daher, man sollte in der Lage sein, Inkonsistenzen zu entdecken, sinnvoll mit den inkonsistenten Objekten umzugehen und letztendlich die gewünschte konsistente Situation zu erreichen. Die Toleranz gegenüber inkonsistenten Objekt-Zuständen verlangt Unterstützung von Werkzeugen und Prozessen, d.h. sie müssen "wissen", wie mit inkonsistenten Objekten umzugehen ist. Das Sicherstellen eines sinnvollen Umgangs kann z.B. auch den Austausch der auf den Objekten definierten Operationen beinhalten.
- 5) **Dynamische Änderung der Konsistenz-Spezifikationen:** Die letzte allgemeine Anforderung an Konsistenzerhaltungssysteme verlangt eine dynamische Änderung der Konsistenz-Spezifikationen. Darunter sollte man verstehen, dass auch das Hinzufügen von neuen Konsistenz-Regeln, wie wir sie im Abschnitt 2.2.1 gesehen haben, möglich sein sollte. Das würde zusätzlich bedeuten, dass ein solches System, was die Anzahl der Konsistenz-Regeln angeht, skalierbar sein sollte. Die Änderung von Konsistenz-Regeln im Laufe der Zeit kann auch sinnvoll sein, wenn man feststellt, dass irgendwelche Regeln nicht mehr unsere Zwecke erfüllen.

Nach diesen allgemeinen funktionellen Anforderungen an Konsistenzerhaltungssysteme sind wir bereit, diese für unser eigenes erwünschtes System zu konkretisieren. Im nächsten Abschnitt wird jeder von den Punkten, die hier erwähnt wurden, durchgegangen, um unser System zu spezifizieren.

6.2.3 Spezifikation

Wir fangen hier mit dem ersten Punkt des vorigen Abschnitts, mit der Definition von Konsistenzbedingungen an. Konsistenter Zustand ist der Zustand, in dem alle Benutzer die gleiche Dokumentenstruktur mit dem gleichen Dokumenteninhalte vor sich haben. Dies umschließt auch die Tatsache, dass die Benutzer nicht ständig mit dem DDE-Server verbunden sind (siehe Abschnitt 2.2.2). D.h. falls sie etwas lokal kopiert haben, um es zu ändern, müssen sie über jede von einem anderen Benutzer durchgeführte Änderung informiert werden. Jeder Zustand, in dem zwei oder mehrere Benutzer eine unterschiedliche Dokumentenstruktur oder einen unterschiedlichen Dokumenteninhalte vor sich haben, ist inkonsistent, wobei keine weiteren Konsistenzgrade dazwischen existieren.

Zum zweiten Punkt des vorigen Abschnitts, dem Festlegen, wie und wann Verletzungen entdeckt werden sollten, möchte ich sagen, dass eine Entdeckung von Inkonsistenzen nur während der Zeit der Ankopplung eines Benutzers an das System stattfinden kann. Sie basiert auf den Operationen, mit deren Hilfe ein Benutzer entweder die Struktur oder den Inhalt des Dokuments ändern möchte (siehe Abschnitt 5.2.5).

Das erweiterte DDE-System sollte auch mobile Benutzer, also Benutzer ohne ständige Netzwerkverbindung unterstützen. Hierfür ist, wie bereits im Kapitel 5 beschrieben, eine Replikation der Kapitelinhalte und der Dokumentenstruktur auf den mobilen Rechnern nötig. Diese Replikat können unabhängig voneinander und vom Dokument auf dem DDE-Server geändert werden. Inkonsistente Zustände werden also vom System durchaus ge-

duldet, aber nur solange, bis einer der Benutzer seine Änderungen in das Dokument, das auf dem DDE-Server liegt, integrieren möchte.

Und somit kommen wir zum vierten Punkt des vorigen Abschnittes, dem Inkonsistenz-Management. Sobald ein Benutzer seine Änderungen auf dem Server wirksam machen möchte und es zusätzlich noch andere Benutzer gibt, die mit demselben Dokument arbeiten, müssen diese informiert werden, damit möglichst viele Replikate gleichzeitig in das Ursprungsreplikat reintegriert werden. Falls zwei oder mehrere Benutzer ihre Replikate ohne Beteiligung des DDE-Servers reintegrieren möchten, ist es ihnen durchaus erlaubt. Der DDE-Server aber wird es nicht wissen und die Reintegration wird wieder in der gewohnten Weise verlaufen. Zur Beseitigung von Inkonsistenzen muss auch eine Benutzerbeteiligung erlaubt sein.

Zur dynamischen Änderung der Konsistenz-Spezifikationen möchte ich noch erwähnen, dass das Hinzufügen von weiteren, Entfernen oder Ändern von bereits existierenden Konsistenzregeln vom System möglichst einfach gestaltet sein sollte.

Bei unserem System kann nicht davon ausgegangen werden, dass sich die mobilen Benutzer regelmäßig bei den ortsfesten Servern über Wählleitungen einloggen, um eine Synchronisation zu ermöglichen. Jeder Benutzer sollte jederzeit in der Lage sein, Kapitelinhalte oder die Dokumentenstruktur zu ändern. Entsprechend ist mit einem Auseinanderlaufen der Versionen zu rechnen und eine entsprechende Reintegration nach Wiederanbindung der mobilen Benutzer vorzunehmen. Hierzu müssen dabei auftretende Konflikte zuverlässig erkannt und automatisch behoben werden, sofern dies möglich ist. Ansonsten muss eine Unterstützung für die Reintegration durch den Benutzer geboten werden.

Wie bereits die ursprüngliche DDE, soll auch das KES mit theoretisch beliebigen Editoren funktionieren. Es kann also nicht davon ausgegangen werden, dass die Editoren von sich aus Gruppenarbeit unterstützen oder neben ihrer normalen Arbeit zusätzliche Informationen liefern, die zur Konfliktvermeidung oder Konfliktbehebung verwendet werden könnten. Da eine Anbindung an die DDE bisher nur für XEmacs existiert, ist diese Erweiterung auch für XEmacs zu implementieren. Es ist jedoch darauf zu achten, dass eine Anbindung anderer Editoren möglich bleiben muss.

Desweiteren ist die Erweiterung in einer zur bisherigen DDE konformen Art und Weise zu gestalten. Ein vollständiges Re-Design des Grundkonzeptes soll nicht stattfinden, die für die Erweiterung nötigen Änderungen im Design sollen sich möglichst nahtlos in die bestehende Struktur einfügen.

7. Entwurf

Im vorigen Kapitel "Anforderungsanalyse und Spezifikation" haben wir in Bezug auf Konsistenz gesehen, wie die jetzige Situation ist und wie die erwünschte Situation aussehen soll. In diesem Kapitel werden wir die erwünschte Situation etwas näher betrachten und werden einen Entwurf erstellen, wie die Ziele der Spezifikation zu realisieren sind. Mit anderen Worten, hier wird uns die Frage beschäftigen, wie unser KES zu realisieren ist. Dabei werden uns weniger technische Fragen der Agenten beschäftigen, als vielmehr die Dokumentenstruktur und die auf ihr erlaubten Operationen, Konfliktfälle bei der Änderung eines Dokuments von mehreren Benutzern und letztendlich Regeln, mit denen man diese Konflikte beheben kann.

7.1 Dokumentenstruktur und die auf ihr erlaubten Operationen

Im Abschnitt 2.1.1 wurde allgemein über Dokumentenstruktur und Granularität der Dokumentenstruktur gesprochen und im Abschnitt 4.2 wurde die Dokumentenstruktur der existierenden Mole-DDE beschrieben. In diesem Kapitel wird das Thema noch einmal aufgegriffen, sowohl weil ich für das zu entwerfende KES eine Dokumentenstruktur mit feinerer Granularität für nötig halte, als auch weil ich die Operationen, die einzelne Komponenten der Dokumentenstruktur ändern, erläutern möchte.

Zur Erinnerung möchte hier noch einmal die Dokumentenstruktur der existierenden Mole-DDE beschreiben. Die Mole-DDE unterscheidet die Komponenten Bücherregal, Buch und Kapitel. Obwohl man theoretisch auch mehrere Bücherregale definieren könnte, existiert in der Praxis nur ein Bücherregal, in dem man mehrere Bücher und in jedem Buch mehrere Kapitel anlegen kann. Darin, dass nur ein Bücherregal existiert, sehe ich keine Einschränkung und es wird uns im weiteren nicht mehr beschäftigen, d.h. das Wort Bücherregal wird in unserer Diskussion nicht mehr auftauchen.

Die Komponente Kapitel behält ihre bisherige flache Struktur, d.h. sowohl ein Kapitel als auch alle seine Unterkapitel befinden sich auf der gleichen Ebene. Zwischen Kapiteln und deren Unterkapiteln unterscheidet man nur durch eine sinnvolle Nummerierung. Sie ist Aufgabe der Verfasser der Dokumente. Zur Erläuterung muss hier noch erwähnt werden, dass unter den Begriffen Buch und Kapitel nur deren Titel, nicht aber deren Inhalte zu verstehen sind. Zuständig für den Inhalt ist eine zusätzliche neue Komponente, der **Absatz**, die hier in dieser Arbeit eingeführt wird. Unter Absatz verstehe ich einen Teil eines Kapitels ohne eigenen Titel genauso, wie man ihn von den meisten Textverarbeitungsprogrammen her kennt, nämlich den Teil zwischen zwei Wagenrückläufen ([ENTER]). Diese zusätzliche Komponente hat nur intern für die Mole-DDE eine Bedeutung, die Editorumgebung (Wrapper und XEmacs) weiß nichts davon, d.h. die neue Komponente lässt die Schnittstelle zwischen Editor und Mole-Server unverändert, wie wir sie im Abschnitt 4.7 kennengelernt haben. Da ein Absatz keinen Titel hat, braucht man zu seiner Identifikation die Kombination von Buch und Kapitel und eine Zahl, die jeden Absatz innerhalb eines Kapitels eindeutig identifiziert.

Die Schnittstelle zwischen Editor und DDE-Server definiert unter anderem die Anweisungen `createBook`, `destroyBook`, `createSection`, `writeSection`, und `destroySection` (siehe Abbildung 4.3). Alle diese Anweisungen können ein Dokument ändern. Allgemeiner betrachtet, sind Änderungen an einem Dokument das Ergebnis von **insert**, **update** und **dele-**

te Operationen, die für Komponenten eines Dokuments eventuell definiert sind. Die Anweisungen `createBook` und `destroyBook` sind z.B. äquivalent zu den Operationen "insert Book" bzw. "delete Book". Um die Änderungen also an einem Dokument zu verfolgen ist es sinnvoll, die drei Operationen `insert`, `update` und `delete` im DDE-Server zu definieren. Sobald der DDE-Server eine von den oben erwähnten Anweisungen empfängt, sollte er überprüfen, welche der möglichen Änderungsoperationen ausgeführt wurde. Das ist im Fall von `createBook` und `destroyBook` sehr einfach, wie wir bereits in unserem Beispiel gesehen haben. Auch die Anweisungen `createSection` und `destroySection` können sehr leicht mit den Operationen "insert Kapitel" und "delete Kapitel" identifiziert werden. Im Fall der Anweisung `writeSection` aber muss im DDE-Server überprüft werden, ob das zu speichernde Kapitel geändert wurde oder nicht und falls es geändert wurde, welche Absätze des Kapitels betroffen sind.

In der existierenden Mole-DDE wurde in der Editorumgebung für die Komponenten Buch und Kapitel keine Operation für deren Umbenennung definiert. Weil Buch und Kapitel, wie bereits erwähnt, für den Titel des Buchs bzw. Kapitels stehen und nicht für den Inhalt, ist auch eine Definition der `update`-Operation im DDE-Server überflüssig, da `update` ja nichts weiter als eine Umbenennung wäre. Für die Komponenten Buch und Kapitel werden also nur die Operationen `insert` und `delete` definiert. Für die Komponente Absatz werden alle drei Änderungs-Operationen definiert. Die Operationen `insert` und `delete` für einen Absatz braucht man nicht weiter zu erläutern, die Operation `update` dagegen schon, sie hat nämlich die Bedeutung der Änderung des Inhalts eines Absatzes.

Die drei Änderungsoperationen, die hier für Komponenten unserer Dokumente definiert wurden, dienen als Basis für unsere weitere Diskussion über Konsistenz von Dokumenten. Im nächsten Abschnitt werden wir untersuchen, welche Kombinationen von diesen Operationen zu einem inkonsistenten Dokument führen. Diese Kombinationen werden Konflikte genannt, weil der Änderungswunsch eines Benutzers im Konflikt zu Änderungswünschen anderer Benutzer steht. Ziel ist die Ausführung solcher Kombinationen von Operationen von unserem KES aus nicht zuzulassen.

7.2 Von Änderungsoperationen verursachte Konflikte

Unabhängig von den Komponenten eines Dokuments kann man allgemein betrachtet die Konflikte, die in der Tabelle 7.1 erläutert werden, unterscheiden. Da die Reihenfolge der Operationen aber für die Betrachtung der Konflikte keine Rolle spielt, sind die Konflikte, die symmetrisch zu den Diagonalen dieser Tabelle sind, äquivalent und aus diesem Grund reduziert sich die Anzahl der zu betrachtenden Fälle ungefähr auf die Hälfte.

	B delete	B insert	B update
A delete	1	2	3
A insert	4	5	6
A update	7	8	9

Tabelle 7.1: Konflikte bei Änderungsoperationen auf Dokumentkomponenten

Aus Einfachheitsgründen wurden die Operationen von zwei Benutzern A und B betrachtet.

Der Fall 1 stellt keinen Konfliktfall dar. Wenn zwei oder mehrere Benutzer eine Komponente löschen möchten, wird sie einfach gelöscht. Die Fälle 2 und 4, 3 und 7, 6 und 8 sind äquivalent, weshalb jeweils nur ein Fall betrachtet wird, genauer gesagt die Fälle 2, 3 und 6. Die Fälle 5, 6 und 9 können, müssen aber nicht Konfliktfälle sein. Im Weiteren werden wir sehen, unter welchen Bedingungen diese Fälle Konfliktfälle darstellen.

Wie wir bereits im vorigen Abschnitt gesehen haben, sind, wenn von Büchern oder Kapiteln die Rede ist, deren Titel gemeint, während in Bezug auf Absätze der Inhalt gemeint ist. Aus diesem Grund wird ab jetzt auch explizit zwischen zwei Komponentenebenen unterschieden. Die Komponentenebene Buch-Kapitel und die Komponentenebene Absatz.

Auf der Komponentenebene Buch und Kapitel greifen die Operationen auf die Titel der entsprechenden Komponente, während sie auf der Komponentenebene Absatz den Inhalt betreffen. Mit der Operation delete wird allerdings nicht nur der Titel eines Buches oder Kapitels gelöscht, sondern auch deren Inhalte. Diesen Fall (Löschen eines ganzen Buchs oder Kapitels) betrachte ich als einen in der Praxis selten auftretenden Randfall. Wenn es um inhaltliche Veränderungen eines Dokuments geht, sollte die genaue Position der Änderungen angegeben werden können. Daher wird die Komponente Absatz eingeführt und es wird zwischen zwei verschiedenen Ebenen unterschieden.

7.2.1 Komponentenebene Buch-Kapitel

In diesem Abschnitt werden wir die Konfliktfälle auf der Komponentenebene Buch-Kapitel untersuchen. Nachdem ein oder auch mehrere Konfliktfälle beschrieben wurden, wird eine Regel gegeben, mit der man den Konflikt oder die Konflikte lösen kann. Diese Regel muss aber nicht als die einzige Möglichkeit betrachtet werden, den Konflikt oder die Konflikte zu lösen. Sie stellt nur eine Möglichkeit dar. Unser KES sollte die Möglichkeit offen lassen, auch andere Regeln definieren zu können

Konfliktfall delete-insert 2: Ein Benutzer löscht ein Buch oder ein Kapitel, während ein oder mehrere andere Benutzer demselben Buch oder Kapitel neue Komponenten hinzufügen. Neue Komponenten hinzuzufügen, bedeutet im Fall eines Buches, dass neue Kapitel oder auch neue Absätze bereits existierenden Kapiteln hinzugefügt werden. Im Fall eines Kapitels bedeutet es einfach das Hinzufügen von neuen Absätzen.

Konfliktfall delete-update 3: Ein Benutzer löscht ein Buch oder ein Kapitel, während ein oder mehrere andere Benutzer in demselben Buch oder Kapitel Änderungen vornehmen. Im Falle eines Buches können die Änderungen einen oder mehrere Absätze von verschiedenen Kapiteln betreffen, während im Falle eines Kapitels alle geänderten Absätze zu demselben Kapitel gehören.

Zur Lösung beider Konfliktfälle kann man die folgende Regel definieren:

Regel 1: Kein Benutzer sollte in der Lage sein, ein Buch, oder ein Kapitel ohne die Zustimmung aller anderen Benutzer zu löschen.

Jetzt kommen wir zu den Fällen 5, 6 und 9. Die Konfliktfälle 6 und 9 kann es auf dieser Komponentenebene gar nicht geben, weil, wie wir bereits im Abschnitt 7.1 gesehen ha-

ben, die Operation update für die Komponenten Buch und Kapitel nicht definiert ist. Der Fall 5 kann nur unter bestimmten Bedingungen ein Konflikt sein:

Konfliktfall insert-insert 5: Zwei oder mehrere Benutzer fügen ein neues Buch oder ein neues Kapitel mit demselben oder einem anderen Namen hinzu. Es kann sein, dass sie alle "dasselbe" Buch oder Kapitel meinen, aber unterschiedliche Titel ausgewählt haben, oder andersherum unterschiedliche Bücher oder Kapitel meinen, aber trotzdem denselben Titel ausgewählt haben.

Für diesen Konfliktfall muss man die folgende, im Prinzip alternativlose Regel definieren:

Regel 2: Alle Benutzer sollen über alle Änderungen informiert werden. Sie können nur untereinander den Konflikt lösen.

7.2.2 Komponentenebene Absatz

Der Fall 2 kann unter bestimmten Bedingungen ein Konfliktfall sein:

Konfliktfall delete-insert 2: Ein oder mehrere Benutzer löschen Absätze in einem Kapitel, während ein oder mehrere andere Benutzer demselben Kapitel Absätze hinzufügen. Wenn die Absätze, die hinzugefügt werden, in semantischem Zusammenhang mit den gelöschten stehen, liegt ein Konflikt vor.

Auch für diesen Konfliktfall kann man nur die Regel 2 definieren. Wir haben hier von Absätzen des gleichen Kapitels gesprochen. Es könnte natürlich auch zwischen Absätzen verschiedener Kapitel ein semantischer Zusammenhang bestehen. Da die Wahrscheinlichkeit hierfür aber eher gering ist, werden wir diesen Fall hier ignorieren.

Der Fall 3 stellt generell einen Konfliktfall dar:

Konfliktfall delete-update 3: Ein Benutzer A löscht einen Absatz, während andere Benutzer denselben Absatz ändern.

Regel 3: Änderungen bleiben weiterhin bestehen. Benutzer A soll bei der nächsten Verbindung mit dem DDE-Server darüber informiert werden, wie jetzt der Absatz aussieht und von wem die Änderungen vorgenommen wurden. Wenn A mit dem neuen Zustand nicht zufrieden ist, sollte er das Problem in direktem Kontakt mit den anderen direkt betroffenen Benutzern lösen.

Fälle 5, 6 sind genauso wie Fall 2 nur unter bestimmten Bedingungen Konfliktfälle:

Konfliktfall insert-insert 5: Zwei oder mehrere Benutzer fügen einem Kapitel neue Absätze hinzu. Es ist sehr schwer zu sagen, wann in diesem Fall ein Konfliktfall vorliegt. Die Absätze könnten syntaktisch unterschiedlich, aber semantisch äquivalent sein und somit zu Redundanzen führen, oder sie könnten semantisch konträr sein und somit zu Widersprüchen führen.

Konfliktfall insert-update 6: Einer oder mehrere Benutzer fügen einem Kapitel neue Absätze hinzu, während ein oder mehrere andere Benutzer bereits existierende Absätze ändern. Wie bereits zu Konfliktfall 5 erläutert, ist es schwer zu entscheiden, wann ein Konfliktfall vorliegt.

Für diese beiden Fälle kann als Lösung nur die Regel 2 definiert werden.

Und kommen wir jetzt zu unserem letzten Konfliktfall:

Konfliktfall update-update 9: Einer oder mehrere Benutzer ändern denselben Absatz.

Und eine mögliche Regel zur Konfliktlösung:

Regel 4: Wenn die geänderten Absätze vollständig identisch sind, sollte man sie einfach so lassen wie sie sind. Wenn sie allerdings unterschiedlich sind, liegt ein Konflikt vor, den nur die entsprechenden Benutzer, die die Änderungen vorgenommen haben, lösen können.

7.3 Eine Diskussion über Konsistenzregeln

Im Abschnitt 2.2.1 wurde Konsistenz als die Einhaltung von Regeln definiert und Regeln, die festlegen, welcher Zustand einer oder mehrerer Dokumente als konsistent bezeichnet wird, wurden Konsistenzregeln genannt. Im jetzigen Kapitel wurde deutlicher, dass die Nicht-Einhaltung von derartigen Konsistenzregeln zu Konflikten führen kann, deren Lösung man nur mit der Einführung von weiteren Regeln erzielen kann. Mit anderen Worten, die Regeln, die in den Abschnitten 7.2.1 und 7.2.2 vorgestellt wurden, stellen Meta-Konsistenzregeln dar, die die Einhaltung von replikatübergreifenden Konsistenzregeln, wie sie im Abschnitt 2.2.1 vorgestellt wurden, erst ermöglichen.

Diese Meta-Konsistenzregeln, die vorgeben, was man in einem Konfliktfall zur Auflösung dieses Konflikts tun sollte, sind auch Konsistenzregeln. Indem man nämlich festlegt, wie man einen inkonsistenten Zustand vermeiden kann, legt man indirekt fest, was ein konsistenter Zustand ist. Die Konsistenzregeln der vorigen beiden Abschnitte sind laut Abschnitt 2.2.1 replikatübergreifende Regeln. In demselben Abschnitt (2.2.1) haben wir gesehen, dass man für Dokumente auch weitere Arten von Konsistenzregeln definieren kann. Aber auch für einige der replikatübergreifenden Konsistenzregeln der Abschnitte 7.2.1 und 7.2.2 gibt es Alternativen, wie bereits dort angedeutet.

In der Spezifikation haben wir die Anforderung gestellt, dass sowohl das Hinzufügen weiterer, als auch das Entfernen oder Ändern bereits existierender Konsistenzregeln mit unserem KES möglich sein sollte. Um eine solche Möglichkeit anbieten zu können, sollten die Konsistenzregeln nicht im System hardcodiert sein, sondern als Variablen von der grafischen Benutzungsoberfläche (siehe Abschnitt 4.8) aus, der Mole-DDE, frei änderbar sein. Aus diesem Grund ist eine Erweiterung der existierenden Benutzungsoberfläche notwendig.

Viel wichtiger aber noch als die Erweiterung der grafischen Benutzungsoberfläche, ist die Art und Weise, wie man eine solche Flexibilität eines KES erreichen kann. Da Inkonsistenzen zu Konfliktfällen führen, ist ein Mechanismus erforderlich, der Konfliktfälle lösen kann. Die Konfliktlösungsfähigkeit dieses Mechanismus sollte nicht auf Spezialfälle beschränkt sein, sondern sie sollte für jede Art von Konflikt gegeben sein. Ein solcher Mechanismus wird in den nächsten Abschnitten vorgestellt.

7.4 Das REVISE-System

Bei dem REVISE-System [DaPeSc97], handelt es sich um ein NMR-System, das Konflikte lösen kann. Um Konflikte automatisch zu lösen, muss man sie zuerst in geeigneter Form darstellen, damit sie dann von einem solchen Mechanismus erkannt werden können. Aus diesem Grund folgen im nächsten Abschnitt einige Definitionen, die den Grundstein für das weitere Verständnis unseres Vorgehens legen.

7.4.1 Erweitertes logisches Programm

Die folgenden Definitionen sind grundlegender Teil der Theorie der **Well Founded Semantics with eXplicit negation** [PeAl92] (WFSX), einer logischen "Sprache", die als Erweiterung von Prolog, viel bessere Möglichkeiten als Prolog anbietet, sowohl Fakten als auch Argumente als logische Terme auszudrücken. Hier geht es nicht um eine Einführung in die WFSX, sondern es werden einige wichtige Begriffe vorgestellt. Für ein tiefergehendes Verständnis der WFSX wird auf die umfangreiche Literatur am Ende dieser Arbeit verwiesen.

Anfangen möchte ich mit der Definition des erweiterten logischen Programms:

Definition 1. Erweitertes logisches Programm

Ein erweitertes logisches Programm ist eine (möglicherweise auch unendliche) Menge von Regeln der Art $L_0 \leftarrow L_1, \dots, L_m, \text{not}L_{m+1}, \dots, \text{not}L_n$ ($0 \leq m \leq n$), wobei jedes von den L_i ($0 \leq i \leq n$) ein objektives Literal ist. Ein objektives Literal ist entweder ein Atom A oder dessen explizite Negation $\neg A$. Literale der Art $\text{not}L$ werden default Literals genannt. Literale sind also entweder objektiv oder default, beides ist nicht möglich.

Ein erweitertes logisches Programm ist demnach eine Sammlung von logischen Regeln, wobei jede dieser logischen Regeln auf der rechten Seite sowohl Atome, also einfache Aussagen der Art: "Benutzer A hat Kapitel 1 gelöscht", als auch explizit negierte Atome, also Aussagen der Art: "Benutzer A hat Kapitel 1 nicht gelöscht", als auch implizit (default) negierte Atome, also Aussagen der Art: "falls Benutzer A Kapitel 1 nicht gelöscht hat", enthalten kann.

Die **implizite Negation** bedeutet also kein gesichertes Wissen. Die Aussage "falls Benutzer A Kapitel 1 nicht gelöscht hat", impliziert weder, dass Benutzer A Kapitel 1 gelöscht hat, noch, dass er es nicht getan hat. Sie ist nur eine Annahme, die man macht, um etwas anderes zu implizieren, wie z.B. "falls Benutzer A Kapitel 1 nicht gelöscht hat, werden die Änderungen von Benutzer B übernommen".

Ein weiterer wichtiger Punkt für das Verständnis dieser Thematik ist, dass die **explizite Negation**, die implizite bewahrheitet. So gilt beispielsweise, dass falls "Benutzer A Kapitel 1 nicht gelöscht hat" wahr ist, dann ist auch "falls Benutzer A Kapitel 1 nicht gelöscht hat" wahr. Die rechte Seite solcher Regeln kann leer sein. In diesem Fall spricht man, wie auch in der üblichen Logik, von einem Fakt.

Definition 2. Integritätsbedingung

Eine Integritätsbedingung hat die Form $\perp \leftarrow L_1, \dots, L_m, \text{not}L_{m+1}, \dots, \text{not}L_n$ ($0 \leq m \leq n$), wobei jedes von den L_i ($0 \leq i \leq n$) ein objektives Literal ist und \perp für Falsch steht.

Eine Integritätsbedingung stellt eine besondere Art von Regel dar. Syntaktisch gesehen ist der einzige Unterschied zwischen einer üblichen Regel und einer Integritätsbedingung der Kopf der Regel. Eine normale Regel hat als Kopf ein objektives Literal, während eine Integritätsbedingung ein \perp hat, das Symbol für Konflikt oder Fehler, je nachdem in welchem Kontext die WFSX verwendet wird. Semantisch gesehen ist der Unterschied zwischen den beiden, dass eine Regel den Lösungsraum erweitert, während eine Integritätsbedingung ihn beschränkt.

Außer Fakten, Regeln und Integritätsbedingungen enthält ein erweitertes logisches Programm noch Revisables:

Definition 3. Revisable, Revision

Die Revisables R eines Programms P sind eine Untermenge der default Literale. Die entsprechenden Literale, dürfen nicht als Köpfe von Regeln auftauchen. Die Menge $R' \subseteq R$ heißt Revision, wenn sie eine minimale Menge ist, so dass $P \cup R'$ keine Konflikte enthält. In der Praxis sind die Revisables nichts weiter als Fakten, die man ändern sollte, um Konflikte in einem logischen Programm zu lösen.

Mit dem Ausdruck “die Revisables R eines Programms P sind eine Untermenge der default Literale“ ist gemeint, dass die Revisables Literale sind, über die man nicht genau weiß, ob sie wahr oder falsch sind. Man macht eine Annahme über deren Wahrheitswert, falls aber diese Annahme Grund für einen Konflikt ist, wird der Wahrheitswert geändert, um den Konflikt zu lösen.

Um das ganze etwas verständlicher zu machen, habe ich auf der Abbildung 7.1 ein kleines erweitertes logisches Programm dargestellt. Es enthält zwar keine “übliche“ Regel, dafür aber zwei Revisables und eine Integritätsbedingung. Die zwei Revisables setzen die Fakten “insert(buch, kapitel)” und “delete(buch)” auf wahr, während die Integritätsbedingung “ $\perp \leftarrow \text{delete}(\text{buch})$ “, die Ansicht vertritt, dass ein Löschen des Buches zu einem Konflikt führen würde. Um diesen Konflikt zu lösen, muss der Wert von delete(buch) auf falsch gesetzt werden, er wird also revidiert. Die Revision besteht also in diesem Fall nur aus dem Fakt “delete(buch) ist falsch“. Man könnte auch den Wert von “insert(buch, kapitel)” auf falsch setzen, die Revision würde aber weiterhin nur aus dem Fakt “ $\neg \text{delete}(\text{buch})$ “ bestehen.

```
revisable(insert(buch, kapitel), true)
revisable(delete(buch), true)

 $\perp \leftarrow \text{delete}(\text{buch})$ 
```

Abbildung 7.1: Ein kleines erweitertes logisches Programm

7.4.2 Das REVISE-System als Teil eines Konfliktlösungsmechanismus

Im vorigen Abschnitt haben wir gesehen, was ein erweitertes logisches Programm ist und wie man damit Konflikte lösen kann. Das REVISE-System nimmt als Eingabe ein solches erweitertes logisches Programm und gibt die Revision zurück, also die Lösungsmenge. Falls kein Konflikt existiert, oder mit den vorgegebenen Revisables die existierenden Konflikte nicht behoben werden können, gibt es einfach eine leere Menge zurück.

Es ist also sehr wichtig, in einem Problem die Konfliktfälle zu erkennen, sie sinnvoll in einem erweiterten logischen Programm darzustellen und die entsprechenden Revisables hinzuzufügen, wenn man die richtigen Lösungen für die Konfliktfälle vom REVISE-System zurückbekommen möchte.

Das REVISE-System selber ist ein größeres (ca. 100 kB) SICStus-Prolog-Programm, das aus mehreren Dateien besteht. Die größere davon stellt einen Interpreter (oder einen Compiler, beide Optionen sind implementiert worden) für erweiterte logische Programme dar. Das REVISE-System allein kann nicht als Konfliktlösungsmechanismus verwendet werden. Es muss in die in Abbildung 7.2 dargestellte Architektur eingebettet werden.

Wie man aus der Abbildung 7.2 entnehmen kann, ist das REVISE-System eine Komponente des Inferenzmechanismus, stellt sogar das Herzstück dar. Die anderen Komponenten fügen weitere Funktionalität hinzu, die für einen solchen Mechanismus nötig ist.

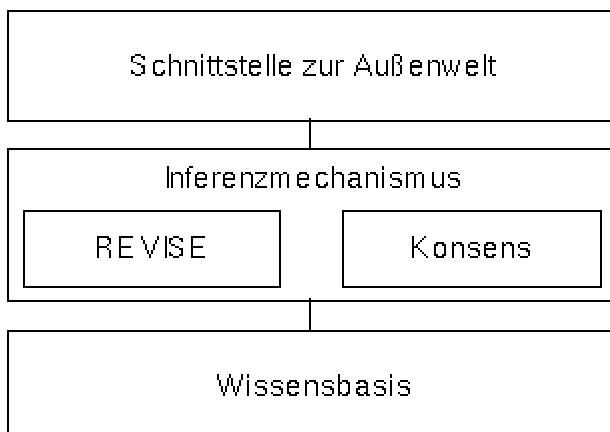


Abbildung 7.2: Komponenten des Konfliktlösungsmechanismus

Die **Wissensbasis** ist der Ort, der erweiterte logische Programme enthält. Sie kann während des Betriebs um weitere Regeln erweitert werden, oder Regeln können auch aus ihr entfernt werden. Die Wissensbasis unterliegt also ständigen Änderungen, abhängig vom aktuellen Wissenstand. Sie kann entweder als eine Datenbank, was alle Vorteile einer Datenbank mit sich bringt, oder als eine einfache Textdatei realisiert werden.

Die **Schnittstelle** zur Außenwelt, kann verschiedene Formate haben, je nachdem, was die Außenwelt ist. So kann sie z.B. im Falle von Benutzern eine grafische Benutzungsoberfläche sein, die sowohl die Eingabe von Daten als auch deren Ausgabe erleichtert und in sinnvoller Weise darstellt. Falls die Außenwelt ein Programm ist, ist die Schnittstelle eine Bibliothek, geeignet für die Verbindung mit Programmen in der entsprechenden Sprache.

Eng verbunden mit der Schnittstelle zur Außenwelt ist auch das **Konsens**-Modul. Es sorgt für Konsens zwischen mehreren unterschiedlichen Benutzern oder Programmen, die gleichzeitig mit dem System verbunden sind, daher auch der Name. Es handelt sich dabei um ein Programm, das sowohl für die Weiterleitung der Daten verschiedener Eingabequellen an die REVISE-Maschine zuständig ist, als auch für die Benachrichtigung der entsprechenden Benutzer oder Programme mit den Ergebnissen. Das Konsens-Modul wird erst im Kapitel Feinentwurf genauer beschrieben.

Um unseren Konfliktlösungsmechanismus besser zu verstehen sollte man an ein Expertensystem denken. Expertensysteme oder "wissensbasierte Systeme" simulieren die Fähigkeiten eines Experten [BüSc88]. Wie schon der Name besagt, ist das Wesentliche an diesen Systemen, dass sie eine Wissensbasis beinhalten, also Wissen in einem solchen System abgespeichert werden kann. Mit Hilfe eines entsprechenden Abarbeitungsprogramms können dann aus diesem Wissen Schlüsse gezogen werden, ähnlich wie auch Experten Lösungsvorschläge auf Grund ihres Wissens erarbeiten. Ein Expertensystem kann allerdings neben seinem Herzstück, dem Inferenzmechanismus, viele andere Komponenten umschließen, wie z.B. allgemeines Wissen, externes Wissen, aufbereitetes Wissen, erfragtes Wissen, eine Fragekomponente, eine Erklärungskomponente usw. In unserem Konfliktlösungsmechanismus existieren nicht alle diese Komponente (wie z.B. die Erklärungskomponente), oder sie existieren zusammengefasst in einer einzigen Komponente unseres Systems (z.B. steht die Wissensbasis der Abbildung 2.6 für alle Wissenskomponenten eines Expertensystems). Unsere "minimalistische" Architektur sollte aber trotzdem für die Realisierung eines Konsistenzerhaltungssystems vollkommen ausreichend sein.

7.4.3 Möglichkeiten, das REVISE-System in die Mole-DDE zu integrieren

Zunächst haben wir in diesem Kapitel gesehen, wie in unserer DDE Konflikte entstehen könnten. Anschließend wurde ein Konfliktlösungsmechanismus, das REVISE-System, beschrieben. Als nächstes wird ein Konzept für die Interaktion zwischen dem REVISE-System und der Mole-DDE vorgestellt.

Die einfachste Lösung ist die direkte Verbindung des Editors mit dem REVISE-System. Eine solche direkte Verbindung ist aber unmöglich, wie aus der Diskussion im Abschnitt 7.1 hervorgeht. Zuerst muss nämlich vom DDE-System geprüft werden, ob und welcher Konfliktfall vorliegt. Erst dann können die für die Konfliktlösung nötigen Daten an das REVISE-System weitergeleitet werden.

Wie bereits im Abschnitt 4.1 beschrieben, ist das Agentensystem die Kernkomponente der Mole-DDE (siehe auch Abbildung 4.1). Interaktion zwischen dem REVISE-System und der Mole-DDE ist also Interaktion zwischen dem REVISE-System und den Agenten der Mole-DDE. Unser Konzept sollte also die Interaktion zwischen Agenten und dem REVISE-System ermöglichen.

Ein Konzept für die Verbindung von Agenten mit dem REVISE-System existiert bereits. Es stellt eine sehr gute theoretische Basis für unser KES dar und man kann auch einige Ideen davon übernehmen. Aus diesem Grund wird es in den folgenden Abschnitten beschrieben.

7.5 Kooperierende und argumentierende Agenten

Bereits im Abschnitt 3.2, als eine Definition für den Begriff Multiagenten-System vorgestellt wurde, war davon die Rede, dass die Agenten eines Multiagenten-Systems dazu fähig sein sollen, miteinander zu verhandeln. Die Verhandlungsfähigkeit ist eine elementare Voraussetzung dafür, dass ein Agent in einem Multiagenten-System autonom agieren kann. Während eines Verhandlungsprozesses kommuniziert eine Gruppe von Agenten miteinander, um zu einer allseits akzeptablen Übereinstimmung zu kommen. Beispielsweise kann ein solcher Prozess der Austausch von Preisen zwischen Käufer und Verkäufer gemäß eines konkreten Protokolls sein. Verhandlung kann aber auch einen komplizierteren Prozess der Argumentation beinhalten, nämlich das Festlegen bzw. Ändern von Überzeugungen/Meinungen von Agenten. Im Kontext des Begriffs Verhandlung hat die Theorie der Argumentation in den letzten Jahren an Bedeutung gewonnen ([PaSiJe98], [KrSyEv98]).

Da, wie wir bereits wissen, Agenten nichts weiter als Software-Programme sind, kommt schnell die Frage auf, wie diese Agenten miteinander argumentieren oder kooperieren können. Die Antwort auf diese Frage kann nur eine intelligente Komponente, wie der Konfliktlösungsmechanismus, den wir in den vorigen Abschnitten kennengelernt haben, sein. Wenn nämlich die Argumente in einer für den Konfliktlösungsmechanismus konformen Form ausgedrückt sind und eine Schnittstelle für die Agenten existiert, könnten die Agenten ihre Argumente eingeben und entsprechende Lösungen zurückbekommen, was den Eindruck erweckt, dass die Agenten miteinander kooperieren oder argumentieren.

Wie das genau im Falle unseres KES funktioniert, werden wir im nächsten Kapitel, dem Feinentwurf, sehen. In den Abschnitten 7.5.1-7.5.3 dieses Kapitels werden wir jedoch zunächst einen theoretischen Rahmen für kooperierende und argumentierende Agenten schaffen.

7.5.1 Argumentation in der Philosophie

Seitdem Leibniz sein *calculus ratiocinator* im Jahre 1679 entwickelt hat, waren und sind noch heute mehrere Forscher mit der Automatisierung der Argumentation beschäftigt. Das größte Problem dabei ist, dass sich viele Argumente nicht formal beschreiben lassen. Die Enzyklopädie Britannica unterscheidet zwischen **semantischen** und **syntaktischen** Argumenten, wobei sich die syntaktischen Argumente formal beschreiben lassen und daher einfacher automatisiert werden können als die semantischen. Es gibt mehrere Formen von Argumenten, sowohl semantische als auch syntaktische. Wir werden unser Interesse auf zwei beschränken, nämlich auf Unterbietung (auf Engl. undercut) und auf Widerlegung (auf Engl. rebut). Undercuts sind Argumente, welche Prämissen von anderen Argumenten attackieren, Rebutts sind Argumente, die direkt andere Argumente attackieren.

Wir beschränken uns auf diese zwei Arten von Argumenten, weil sie ausreichen, um die Semantik von logischen Programmen zu definieren, was auch der Grund ist, unsere Argumentations-Tools als logische Programme zu implementieren [KrSyEv98].

Die Relevanz eines Arguments, d.h. sollte ein Argument akzeptiert oder abgelehnt werden, ist ein weiteres wichtiges Thema in der klassischen Theorie der Argumentation. [Co-

Co94] z.B. unterscheiden 17 verschiedene Trugschlüsse, was die Relevanz der Argumente angeht, wovon nur 3 formal ausgedrückt werden können:

1. *argument from ignorance* auf Deutsch ein *Argument aus Unwissenheit*; bedeutet, Argumentieren für die Richtigkeit einer Aussage einfach auf der Basis, dass ihre Falschheit noch nicht bewiesen wurde, oder andersherum, Argumentieren für die Falschheit einer Aussage einfach, weil ihre Richtigkeit nicht bewiesen ist,
2. *begging the question*, bedeutet das zu Beweisende als feststehend betrachten mit anderen Worten, aus einer erst zu beweisenden Voraussetzung einen Schluss zu ziehen; heißt auch zyklische Argumentation und
3. *Division*; macht die Annahme, wenn etwas als Ganzes wahr ist, müssen alle seine Bestandteile auch wahr sein.

Interessant ist, dass alle drei Beispiele mit nicht-monotoner-Schlussfolgerung (engl. non-monotonic reasoning oder NMR) verbunden sind und zwei verschiedene Arten von Negation verlangen: Implizite und explizite (siehe Abschnitt 2.3.1). Mit diesen beiden Arten von Negation, kann man die drei verschiedenen Trugschlüsse ausdrücken:

1. *Argument from Ignorance* hat die Form: $\alpha \leftarrow \text{not } \neg\alpha$, oder $\neg\alpha \leftarrow \text{not } \alpha$
2. *Begging the question* hat die Form einer positiven Schleife $\alpha \leftarrow \alpha$ oder $\neg\alpha \leftarrow \neg\alpha$
3. *Division* verlangt NMR und Konfliktlösung (siehe Abschnitt 2.3.1). Ein typisches Beispiel in der Literatur ist der Konflikt zwischen fliegenden Vögeln und nicht fliegenden Pinguinen.

In diesem Abschnitt habe ich versucht, einige wichtige Begriffe in Bezug auf Argumentation zu erläutern. Auf der Basis dieser Begriffe wird in den nächsten zwei Abschnitten ein Rahmen für argumentierende und kooperierende Agenten geschaffen.

7.5.2 Argumentation eines Agenten

Die Theorie der Argumentation eines Agenten ist sehr eng verbunden mit der Theorie des erweiterten logischen Programms, die wir bereits im Abschnitt 2.3.1 gesehen haben. Hier werden auch eine Reihe von Definitionen vorgestellt, die man auch als eine Fortsetzung der Theorie im Abschnitt 2.3.1 betrachten kann. Die Theorie hier hat keinen Anspruch auf Vollständigkeit, sondern soll nur einen ersten Einblick vermitteln. Für weitere Einzelheiten wird auf die umfangreiche Literatur am Ende dieser Arbeit verwiesen.

Was ein erweitertes logisches Programm ist, haben wir bereits gesehen, weshalb ich hier mit der Definition eines Arguments für eine Schlussfolgerung beginnen kann [ScMi99].

Definition 1. Sei P ein erweitertes logisches Programm. Ein Argument für eine Schlussfolgerung L , ist eine endliche Reihe von Regeln $A=[r_n, \dots, r_m]$, wobei $r_i \in P$ und:

1. $\forall n \leq i \leq m \wedge \forall$ objektives Literal L_j auf der rechten Seite von $r_i, \exists k < i: L_j$ ist Schlussfolgerung von r_k .
2. L ist die Schlussfolgerung von einer Regel von A und
3. Es gibt keine zwei verschiedenen Regeln des Arguments mit der gleichen Schlussfolgerung.

Eine Reihe von Regeln, die eine Untermenge von A sind und auch ein Argument darstellen, heißt Unterargument von A .

Wie wir bereits im vorigen Abschnitt gesehen haben, gibt es zwei Arten von Argumenten, hier noch ihre formelle Definition:

Definition 2. Seien A_1 und A_2 zwei Argumente, dann A_1 "unterbietet" A_2 , falls A_1 ein Argument für L und A_2 ein Argument mit der Annahme $\text{not } L$ ist. D.h.

$\exists r: L_0 \leftarrow L_1, \dots, L_l, \text{not } L_{l+1}, \dots, \text{not } L_m \in A_2$ und ein $l+1 \leq j \leq m$ so dass $L = L_j$.

A_1 widerlegt A_2 , falls A_1 ein Argument für L und A_2 ein Argument für $\neg L$ ist.

A_1 attackiert A_2 , falls A_1 A_2 unterbietet oder widerlegt.

Definition 3. Ein Argument heißt kohärent (zusammenhängend), falls es keine Unterargumente enthält, die sich gegenseitig attackieren.

Kern des Rahmens der Argumentation ist die Definition der Akzeptanz: Ein Agent akzeptiert ein Argument, falls er in der Lage ist, alle Attacken gegen das in Frage kommende Argument abzuwenden.

Und somit kommen wir zu einer Definition, die eine Relation zwischen zwei Argumenten festlegt:

Definition 4. Seien A_1 und A_2 zwei Argumente, dann A_1 besiegt $A_2 \Leftrightarrow$

- A_1 leer und A_2 nicht kohärent ist,
- A_1 unterbietet A_2 ,
- A_1 widerlegt A_2 und A_2 unterbietet nicht A_1 .

A_1 besiegt absolut A_2 gdw. A_1 besiegt A_2 aber nicht andersherum. A_1 ist akzeptabel in Bezug auf eine Menge von Argumenten $Args$, gdw. jedes Argument, das A_1 unterbietet, absolut besiegt wird von einem Argument in $Args$.

Wir haben in diesem Abschnitt die wichtigsten Punkte der Argumentation aus Sicht eines Agenten kennengelernt. Im nächsten Abschnitt werden wir diese Theorie für ein Multiagenten-System erweitern.

7.5.3 Multiagenten-Argumentation

Ein Agent wird im Rahmen eines Multiagentensystems als ein Tupel betrachtet, bestehend aus der Menge seiner Argumente, seinem Wissensgebiet, einer Flagge, die zeigt ob er skeptisch oder leichtgläubig (credulous) ist, und zwei Mengen, eine für die Agenten, mit denen er kooperieren und eine für die Agenten, mit denen er argumentieren darf.

Definition 5. Sei $n > 0$ die Anzahl der Agenten und $0 \leq i \leq n$ ein Index für einen Agenten. Sei P_i ein erweitertes logisches Programm, $F_i \in \{s,c\}$ eine Flagge, die zeigt, ob es sich um einen skeptischen oder leichtgläubigen Agenten handelt, $Arg_i \subset \{1, \dots, n\}$ und $Coop_i \subset \{1, \dots, n\}$, Mengen von Indizes und Dom_i eine Menge von Prädikaten, die den Kompetenzbereich des Agenten definiert. Dann ist der Tupel $Ag_i = \langle P_i, F_i, Arg_i, Coop_i, Dom_i \rangle$ ein Agent. Eine Menge von Agenten $\mathcal{A} = \{Ag_1, \dots, Ag_n\}$, heißt Multiagenten-System.

Diese Definition legt fest, dass zwei Arten von Interaktion zwischen Agenten existieren: Die Kooperation und die Argumentation. Ein Agent, der nichts über ein konkretes Literal weiß, kooperiert mit anderen, welche ihm mit ihren Kenntnissen weiterhelfen. Im Falle

der Argumentation glaubt ein Agent an etwas, er hat also die entsprechenden Argumente dafür und argumentiert mit anderen Agenten, um herauszufinden, ob seine Überzeugung wahr ist, oder ob er sie revidieren muss. Wenn man über argumentierende Agenten spricht, sollte man zwischen zwei verschiedenen unterscheiden, nämlich zwischen skeptischen und leichtgläubigen, wobei die ersten kritischer eingestellt sind gegenüber ihren eigenen Argumenten. Genauer gesagt akzeptieren skeptische Agenten sowohl Unterbietungen als auch Widerlegungen ihrer Argumente, wobei leichtgläubige Agenten nur Unterbietungen akzeptieren.

Um die Anzahl der Nachrichten zu reduzieren, die ein Agent auf dem Weg zur Wahrheit mit anderen Agenten austauschen muss, wurden diese drei Mengen definiert (Arg_i , $Coop_i$, Dom_i). Ein Agent wird nur die mit ihm kooperierenden Agenten nach Hilfe fragen und von ihnen wieder nur diejenigen, deren Kenntnisbereich für seine Frage relevant ist. Entsprechendes gilt auch für die Argumentation.

7.6 Unterstützung für entkoppelte Arbeit

Im Abschnitt 2.2.2 wurde zwischen gekoppelter und entkoppelter Arbeit unterschieden. Dabei wurde auf die Besonderheit von Inkonsistenz-Problemen, die durch Unterstützung von entkoppelter Arbeit entstehen könnten, hingewiesen. In diesem Abschnitt werden wir uns noch einmal dem Thema widmen, wobei der Schwerpunkt hier auf einer Lösung für die Vermeidung von solchen Inkonsistenzen liegt.

Wenn ein Benutzer seine Änderungen auf dem DDE-Server wirksam machen möchte, kann man folgendes unterscheiden:

- Es gibt gleichzeitig mit ihm auch andere Benutzer, die ihre Änderungen auf dem DDE-Server wirksam machen möchten und
- Es gibt Benutzer die während dieser Zeit entkoppelt mit dem System arbeiten.

Für Benutzer, die gleichzeitig mit dem DDE-System arbeiten, kann man anhand der Änderungen und den gleichzeitig geöffneten Büchern oder Kapiteln feststellen, ob ein Konflikt vorliegt oder ob ein potenzieller Konflikt möglich wäre. Über Benutzer aber, die gerade entkoppelt arbeiten, weiß die Mole-DDE in der existierenden Form nichts.

Eine Erweiterung der Mole-DDE ist also nötig, damit sie über Benutzer, die gerade entkoppelt arbeiten und über den von ihnen bearbeiteten Teil des Dokuments informiert ist. Wie genau eine solche Erweiterung möglich ist, wird im nächsten Kapitel genauer beschrieben. Hier wollte ich nur darauf aufmerksam machen, dass ein Öffnen von Büchern und Kapiteln nicht nur über die Schnittstelle des Editors zum DDE-Server möglich sein sollte. Benutzer, die entkoppelt arbeiten möchten, kopieren das gewünschte Material lokal, und die entsprechenden Bücher und Kapitel werden wieder geschlossen, sobald sie den Editor verlassen. In der Mole-DDE existiert keine Information über ihre Absichten. Eine weitere Möglichkeit, Bücher oder Kapitel für die Bearbeitung zu markieren, sollte also über die grafische Benutzungsoberfläche ermöglicht werden. Benutzern, die abgekoppelt arbeiten möchten, sollte es möglich sein, über die grafische Benutzungsoberfläche ihre Absichten dem DDE-Server bekannt zu machen. Die Markierung von Büchern und Kapiteln, die diese Benutzer bearbeiten möchten, kann die bereits existierende Form der "geöffnet-

Markierung“ haben. Wenn ein Benutzer dann seine Änderungen auf den Server übertragen möchte, werden auch die entkoppelten Benutzer berücksichtigt. Im nächsten Kapitel möchte ich die Konzepte die hier grob vorgestellt wurden, näher erläutern. In diesem Zusammenhang werden wir uns auch mit der Agentenstruktur der existierenden Mole-DDE beschäftigen. Praktische Fragen bezüglich der Realisierung unseres KES stehen dabei im Mittelpunkt.

8. Feinentwurf

In den folgenden Abschnitten soll der Aufbau des Konsistenzerhaltungssystems vorgestellt werden. Es wird diskutiert, welche von den bereits existierenden Komponenten der Mole-DDE als Vorlage dienen können, welche Veränderungen an ihnen nötig sind, um sie unseren Zielen anzupassen und welche neuen Komponenten eventuell hinzukommen müssen. Der Schwerpunkt liegt dabei auf der Funktionsweise des KES. Was tun die bereits existierenden geänderten und die neu hinzugekommenen Komponenten, um die Konsistenzerhaltung von Dokumenten zu gewährleisten? Wie interagieren diese Komponenten zu diesem Zweck?

8.1 Randbedingungen

Ziel dieser Arbeit ist es nicht, ein komplettes KES von Grund auf neu zu entwerfen. Die bereits existierende MOLE-DDE stellt die Basis dar, auf der es aufgebaut wird. Im Abschnitt 4.1 wurde die Struktur der Mole-DDE vorgestellt. Den Kern der Mole-DDE stellt das Agentensystem, mit dem wir uns hauptsächlich beschäftigen werden, dar. Aus diesem Grund werden die Agentenklassen von MOLE-DDE in diesem Abschnitt kurz vorgestellt.

Die bestehende Agentenstruktur wird in zwei Teile untergliedert:

- Agentenklassen, die unverändert übernommen werden und
- Agentenklassen, die verändert werden, um den zusätzlichen Anforderungen gerecht zu werden.

Die Agentenstruktur von Mole-DDE kann man auf der Abbildung 8.1 betrachten.

Werfen wir zuerst einen Blick auf die Agentenklassen, die unverändert übernommen werden:

- **Administrator-Agent:** Im gesamten MOLE-DDE existiert genau ein zentraler Systemagent, der Informationen über alle am System beteiligten Benutzer, Benutzeragenten und deren Aufenthaltsorte hat.
- **Mobile Beobachtungsagenten:** Um Informationen von anderen Benutzeragenten zu erhalten, schickt ein Benutzeragent mobile Beobachtungsagenten an alle Benutzeragenten, von denen er Informationen wünscht. Dort empfängt er alle Ereignisse und filtert diese entsprechend den Benutzerwünschen, um sie schließlich an seinen Benutzeragenten weiterzureichen.
- **Mobile DDE-Filteragenten:** Ausgehend von jedem Benutzer-Agenten wird ein zusätzlicher mobiler Agent zum DDE-Agenten geschickt, um dort von auftretenden Ereignissen unterrichtet zu werden. Es liegt in seiner Verantwortung, die Informationen entsprechend seinen Filtern an andere Filteragenten und seinen Benutzer-Agenten weiterzuleiten.

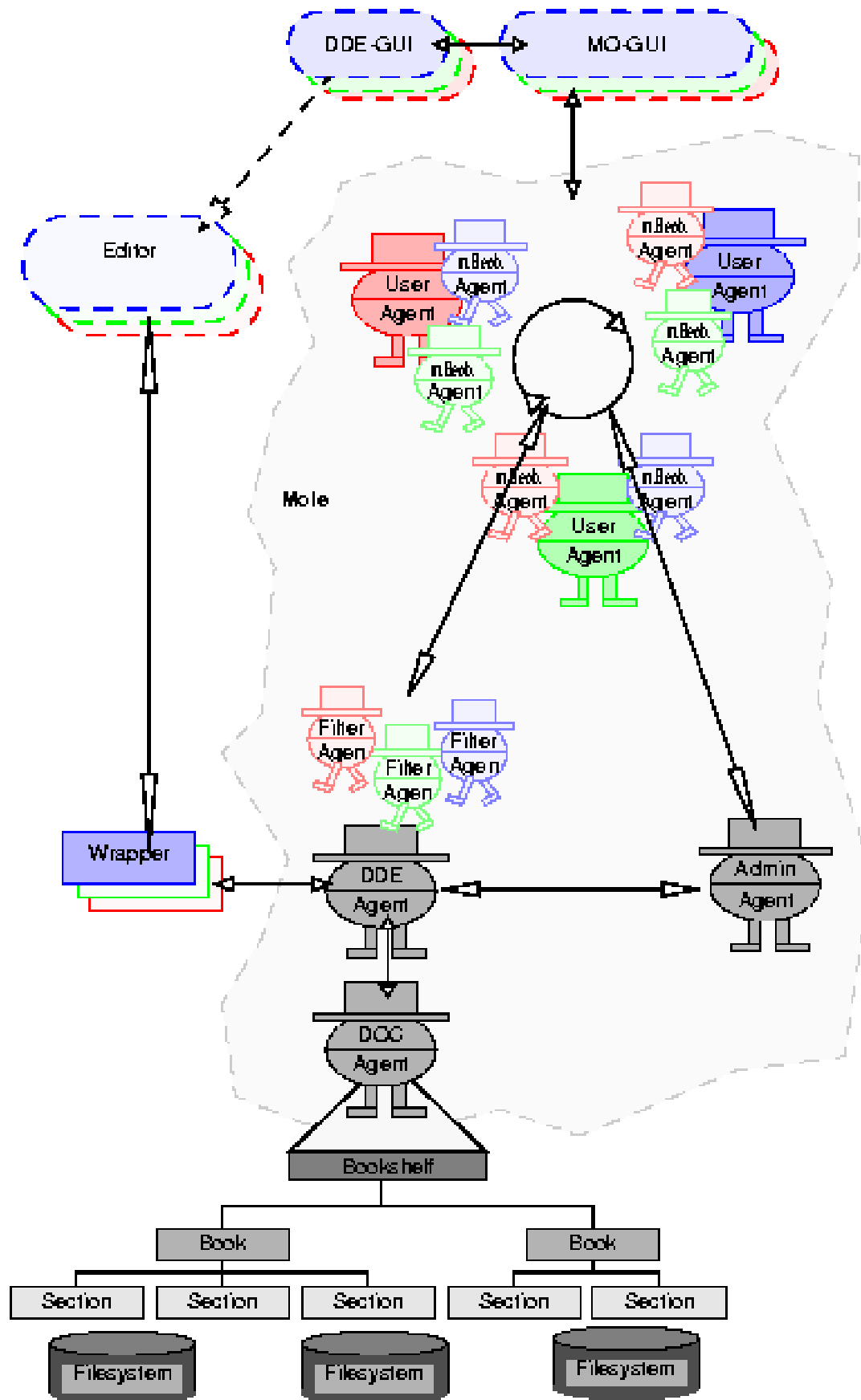


Abbildung 8.1: Mole-DDE Agentenstruktur

Agentenklassen, die geändert werden sollen, sind die folgenden:

- **DDE-Agent:** Der DDE-Agent ist von den bereits bestehenden Agenten derjenige, der am meisten geändert wird. Der DDE-Agent bildet die Schnittstelle zur externen Editorumgebung. Bei ihm melden sich die Wrapper an. Er leitet die Zugriffe auf das Bücherregal koordiniert an den Dokumenten-Agenten weiter und bekommt von diesem eine Rückmeldung über den Erfolg. Vom Administrator-Agenten bekommt er die Benutzerinformationen, um die Benutzerinformationen, die er vom Wrapper bekommt zu kontrollieren. Er nimmt daraufhin eine Zuordnung der Wrapper zu den vorhandenen Benutzern vor und überprüft, ob diese angemeldet sind. Gibt der Dokumenten-Agent eine positive Rückmeldung, wird diese vom DDE-Agent an den Wrapper weitergeleitet. Darüber hinaus generiert der DDE-Agent aus dieser Information ein DDE-Aktions-Objekt (siehe Abschnitt 4.4). Diese Aktion schickt er dem zugehörigen mobilen Filteragenten.
- **Dokumenten-Agent:** Der Dokumenten-Agent verwaltet alle Dokumente. Als zentrales Objekt enthält er das Bücherregal. Er selbst benötigt keine Informationen über die Editoren oder die Wrapper. Auch die Darstellung der Informationen ist aus seiner Sichtweise unwichtig. Bei Zugriffen sorgt er dafür, dass die vorgenommenen Modifikationen korrekt gespeichert werden. Er stößt auch die periodische Speicherung an.
- **Benutzeragenten:** Jeder Benutzer benötigt einen eigenen Systemagenten, der ihn im MOLE-Office und MOLE-DDE vertritt. Er verwaltet alle Zustandsinformationen und sendet Informationen an die bei ihm angemeldeten mobilen Agenten anderer Benutzer. Er besitzt die Schnittstelle zur grafischen Oberfläche und ist für die Interaktion zwischen System und Benutzer verantwortlich.

Von den restlichen Komponenten der DDE-Architektur, also Editor, grafische Benutzungsoberfläche, Wrapper und Dokumentenspeichersystem (siehe Abschnitt 4.1), bleiben alle bis auf eine unverändert. Geändert wird die grafische Benutzungsoberfläche. Sie wird genauer gesagt erweitert, um die zusätzliche Funktionalität des KES zu ermöglichen.

Ziel unseres Vorgehens ist es, die Funktionalität des KES möglichst unabhängig von dem darunter liegenden System zu gestalten. Dadurch könnte man sowohl seine Wiederverwendbarkeit als auch seine Wartbarkeit und Erweiterbarkeit verbessern (siehe Abschnitt 6.2.1). Um die Funktionalität des KES aber möglichst unabhängig von dem bereits existierenden System zu gestalten, sollte die Anzahl der Änderungen an dem bereits existierenden System möglichst gering gehalten werden. Dafür sollte die zusätzliche Funktionalität von zusätzlichen Software-Modulen implementiert werden.

Fazit: Änderungen an dem bereits existierenden System möglichst vermeiden.

8.2 Struktur des MOLE-KES

Im vorigen Abschnitt haben wir die Agentenstruktur des existierenden MOLE-DDE Systems gesehen und sie mit manchen Bemerkungen bezüglich Änderungen, die für das KES notwendig sind, ergänzt. In diesem Abschnitt wird die angestrebte Agentenstruktur des KES geschildert und die Änderungen an den bereits existierenden Agenten näher erklärt.

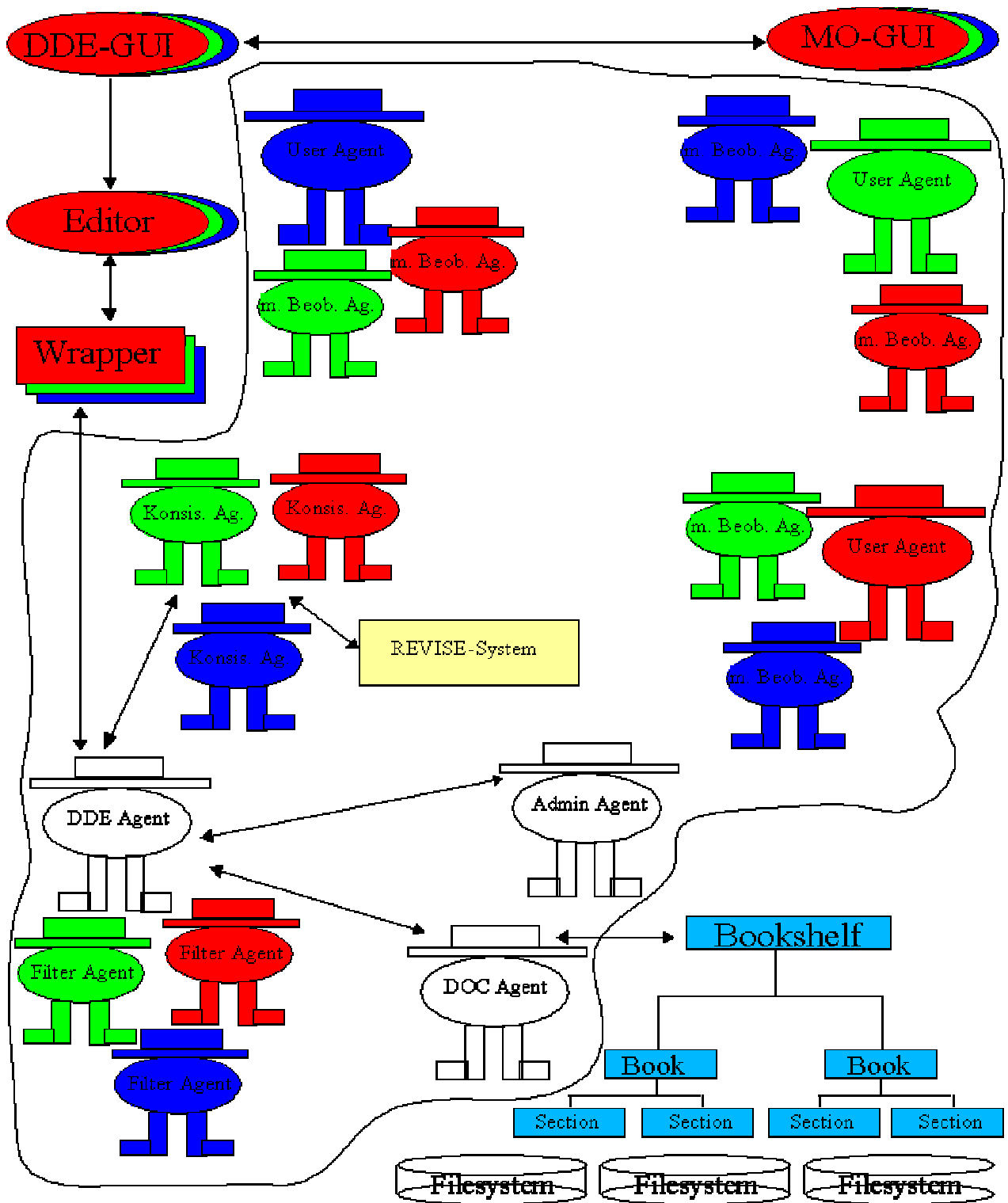


Abbildung 8.2: Agentenstruktur der Mole-DDE, nach der Einführung des KA

Die Agentenstruktur der Mole-DDE soll um die Einführung einer neuen Klasse von mobilen Agenten, den **Konsistenzagenten** (im weiteren KA), erweitert werden. Auf der Abbildung 8.2 kann man sehen, wie die um die KA erweiterte Agentenstruktur der Mole-DDE aussieht.

Konsistenzagenten sind auch mobile Agenten, genauso wie die Filter- oder die mobilen Beobachtungsagenten. Es wird ein KA für jeden Benutzer gestartet, sobald die Lokation, an der sich der entsprechende Benutzeragent befindet, gestartet wird (siehe Abschnitt 3.6.2). Danach wandern sie wie auch die Filteragenten an die Lokation des DDE-Agenten und bleiben dort für den Rest ihres Lebens.

Man sollte sich die Konsistenzagenten als eine Art Berater in Konsistenzfragen auf der Seite des DDE-Agenten vorstellen. Der DDE-Agent weiß nichts von Konsistenz. Er leitet einfach alle Daten, die er über die Schnittstelle zur Editorumgebung bekommt, an den KA des Benutzers weiter, von dem diese Daten stammen. Der KA hat zu überprüfen, ob ein Konfliktfall vorliegen könnte (siehe Abschnitt 9.1) und entsprechend agiert er auch.

Sehr wichtig ist die Rolle der Konsistenzagenten auch für Benutzer, die entkoppelt arbeiten möchten. Ein solcher Benutzer muss seine Absicht über die grafische Benutzungsoberfläche der Mole-DDE melden (siehe Abschnitt 7.6). Der entsprechende Benutzeragent informiert "seinen" KA, welche Bücher und Kapitel zur Bearbeitung markiert werden sollen. Anschließend öffnet der KA diese Bücher und Kapitel im Namen des entsprechenden Benutzers über den DDE- und den Dokumenten-Agenten.

Betrachten wir den Fall eines Benutzers, der über die grafische Benutzungsoberfläche seinen Editor startet. Falls dieser Benutzer entkoppelt gearbeitet hat, sind bereits Dokumententeile als von ihm geöffnet markiert. Gleich nach seiner Anmeldung wird er über die Editor-Schnittstelle darauf aufmerksam gemacht. Solche Dokumententeile brauchen nicht erneut als geöffnet markiert zu werden, mit Büchern oder Kapiteln aber, die er während seiner Sitzung öffnet, wird nach der üblichen Weise verfahren.

Ob und von welchen Benutzern ein Buch oder ein Kapitel geöffnet ist, ist eine Information, die jedes von den Objekten `ddeBook` und `ddeSection` selber trägt. Möchte ein Benutzer einen Teil des Dokuments ändern, überprüft der KA des Benutzers, ob derselbe Teil auch von anderen Benutzern geöffnet ist. Wenn kein anderer Benutzer mit demselben Teil des Dokuments arbeitet, ist es dem DDE-Agenten erlaubt, die Änderungen über den Dokumenten-Agenten zu speichern. Wenn es ein oder mehrere Benutzer gibt, die denselben Dokumententeil geöffnet haben, wird der entsprechende Benutzer über die Editor-Schnittstelle darüber informiert. Das passiert allerdings erst am Ende seiner Sitzung, dann wird er auch darüber informiert, dass seine Änderungen noch nicht definitiv gespeichert werden können. In diesem Fall wird ein anderes Verfahren gestartet, das wir im nächsten Abschnitt kennenlernen werden.

Die Informationseinheiten (siehe Abschnitt 4.7) werden in der Zwischenzeit vom Konsistenzagenten auf der Festplatte gespeichert. Der Grund für eine solche Maßnahme ist, dass die Haltung von Daten im Flüchtigkeitsspeicher eine hohe Gefahr des Datenverlustes mit sich bringt.

Aus der obigen Beschreibung folgt, dass die Klassen `ddeBook` und `ddeSection`, die Agenten DDE-Agent, Dokumententen-Agent und Benutzeragent und die grafische Benutzungsoberfläche geändert werden müssen, wobei die Änderungen nur geringfügig sind.

Die Klassen `ddeBook` und `ddeSection` werden um zwei Vektoren erweitert. Der eine Vektor dient zur Registrierung der Benutzer, die gekoppelt arbeiten, der andere zur Registrierung der Benutzer, die entkoppelt arbeiten.

Es wird jetzt also zwischen gekoppelten und entkoppelten Benutzern unterschieden. Auch der DDE- und der Dokumententen-Agent sollten diese Unterscheidung machen können. Sie sollten auch in der Lage sein, die zusätzlichen Anfragen der Konsistenzagenten über geöffnete Kapitel und Bücher zu behandeln. Der DDE-Agent ist noch von weiteren Änderungen betroffen. Anstatt wie bisher Informationseinheiten, die er über die Editor-Schnittstelle bekommt, als Anweisungen an den Dokumententen-Agenten weiterzugeben, liefert er diese Daten an den KA des entsprechenden Benutzers weiter. Die Informationseinheiten behalten ihre Form auch nach deren Bearbeitung von den Konsistenzagenten. Das bedeutet, dass der DDE-Agent weiterhin derjenige ist, der anhand der Informationseinheiten die entsprechenden Anweisungen an den Dokumententen-Agenten weitergibt.

Die Funktionalität des Benutzeragenten soll erweitert werden, um den KA über abgekoppelt zu bearbeitende Dokumententeile zu informieren. Die grafische Benutzungsoberfläche soll zusätzlich den Benutzern die Möglichkeit geben, ihre Absicht entkoppelt zu arbeiten, dem System zu melden.

8.3 Reintegration

Die Frage, die sich als nächstes stellt, ist, wie lange man warten sollte, bis alle Benutzer, die denselben Dokumententeil bearbeiten, ihre Ausarbeitung zugeschickt haben. Auf Benutzer, die gerade online arbeiten, wartet man solange, bis sie ihre Sitzung beendet haben. Benutzer, die entkoppelt arbeiten, sollten gleich, nachdem der erste Benutzer seine Ausarbeitung abgeschlossen hat, über Email von dem entsprechenden Konsistenzagenten aufgefordert werden, innerhalb der nächsten 24 Stunden auch ihre Ausarbeitung zu schicken. Falls sich vor dem Ablauf dieses Zeitraums die Mehrheit der Bearbeiter gemeldet hat, können die Konsistenzagenten mit der Integration der Änderungen anfangen. Der KA, der die Emails schickt, ist auch derjenige, der die Zeitmessung startet, der die anderen Konsistenzagenten (gekoppelte und entkoppelte) darüber informiert, dass sie an einer Reintegration beteiligt sind und derjenige, der den Konfliktlösungsmechanismus startet (siehe Abschnitt 8.3.2). Falls sich vor dem Ablauf des vorgegebenen Zeitraums die Mehrheit der Benutzer gemeldet hat, meldet der KA, der den Reintegrations-Prozess gestartet hat, die restlichen Benutzer ab. Die Benutzer, die sich noch nicht gemeldet haben, dürfen jetzt auch nicht mehr ihre Änderungen integrieren, bis die Reintegration abgeschlossen ist. Das kann man leicht über einen Flag in den Objekten erreichen. Anschließend an die Reintegration werden diese Benutzer über Email von demselben KA informiert, damit sie sich die neue Version der Dokumententeile holen.

Wie bereits oben erwähnt, werden alle Informationseinheiten vom DDE-Agenten an den KA des entsprechenden Benutzers weitergeleitet. Dieser kann sehr leicht anhand der IE feststellen, welche Änderungen der Benutzer an der Struktur des Dokuments vornehmen

möchte. Für Änderungen, die den Inhalt betreffen, ist es etwas komplizierter, da der KA den neuen Inhalt mit dem bereits existierenden vergleichen muss. Zu diesem Zweck darf der KA über den DDE- und den Dokumenten-Agenten den Inhalt von einzelnen Kapiteln verlangen (wir haben bereits im Abschnitt 7.1 gesehen, dass in unserer Dokumentenstruktur der Begriff Kapitel sowohl Kapitel als auch Abschnitt bedeuten kann). Er vergleicht dann die beiden Inhalte um Änderungen festzustellen. Konflikterkennung basiert also sowohl auf einem Historie-Mechanismus, als auch auf einem Datenvergleich (siehe auch Abschnitt 5.2.5).

Bislang wurden Absätze in diesem Kapitel noch nicht erwähnt, obwohl im vorigen Kapitel davon die Rede war. Absätze sind nur den Konsistenzagenten bekannt. Sie können in einem Kapitel Absätze erkennen und dann die Inhaltsänderungen als Änderungen von Absätzen definieren. Alle anderen Komponenten der Mole-DDE wissen weiterhin nichts von Absätzen.

Änderungen sowohl der Dokumentenstruktur, als auch des Dokumenteninhalts werden von jedem einzelnen KA, der an der Reintegration beteiligt ist, in WFSX ausgedrückt. Wie wir im nächsten Abschnitt sehen werden, bildet der KA aus diesen Änderungen in WFSX und den Konsistenzregeln, die er von der grafischen Benutzungsoberfläche bekommen hat, seine Argumente.

In den nächsten zwei Abschnitten werden wir sehen, wie Konflikte entdeckt und gelöst werden. Hier wird nur noch der Abschluss der Reintegration beschrieben. Konsistenzagenten, die in keine Konfliktfälle verwickelt waren, speichern die Ausarbeitungen ihrer Benutzer definitiv über DDE- und Dokumenten-Agent ab. Gleich danach löschen sie alle von ihnen übergangsweise gespeicherten Daten. Konsistenzagenten, die in Konfliktfälle verwickelt sind, müssen entweder selber die Ausarbeitung ihrer Benutzer ändern oder falls für die Reintegration Benutzerbeteiligung nötig ist, ihre Benutzer benachrichtigen. Die geänderten Ausarbeitungen können dann von den Konsistenzagenten in die Dokumentenstruktur integriert werden und die übergangsweise von ihnen gespeicherten Daten können gelöscht werden. Im Falle einer Inkonsistenz, die nur mit Benutzerbeteiligung behoben werden kann, werden vom Initiatoragent der Reintegration auch die Benutzer, die ihre Ausarbeitung noch nicht abgegeben haben, zur Beteiligung über Email aufgerufen und es wird eine neue Runde von Änderungen eingeleitet. Es ist klar, dass auch in diesem Fall die Daten von ihrem flüchtigen Speicherort weggelöscht werden können.

Im Abschnitt 8.3.1 werden die an der Reintegration beteiligten Konsistenzagenten aus einem rein theoretischen Blickwinkel heraus als ein MAS betrachtet. Diese theoretische Basis haben wir im Abschnitt 7.5 geschaffen. Im Abschnitt 8.3.2 verfolgen wir einen rein praktischen Ansatz. Es wird anschaulich beschrieben, wie die an der Reintegration beteiligten Agenten mit dem REVISE-System interagieren, welche Informationen dabei ausgetauscht werden und was die Konsistenzagenten nach dem Abschluss ihrer Interaktion mit dem REVISE-System machen.

8.3.1 Konsistenzagenten als MAS

Man kann die Konsistenzagenten, die an einer Reintegration beteiligt sind, als ein **Multi-agentensystem (MAS)** von **kooperierenden** und **argumentierenden** Agenten betrachten (siehe Abschnitt 7.5.3). Jeder der Agenten wird durch ein Fünftupel beschrieben:

$\langle \mathcal{P}, \mathcal{F}, Arg, Coop, Dom \rangle$.

Ich werde jetzt versuchen zu erklären, was jedes einzelne der fünf Elemente des Tupels für den konkreten Fall unserer Konsistenzagenten bedeutet. Jeder der Konsistenzagenten hat gegenüber seinen Ansprechpartnern Argumente, diese sind im logischen Programm \mathcal{P} beinhaltet. Die Flagge \mathcal{F} spielt hier keine Rolle, weil alle Agenten skeptisch sind, d.h. jeder von ihnen akzeptiert sowohl Attacken auf seine Prämissen als auch auf seine Folgerungen. Die Menge der Ansprechpartner eines jeden Agenten kann man in eine Menge von Agenten, mit denen er argumentiert (Arg), und in eine Menge von Agenten, mit denen er kooperiert ($Coop$), untergliedern. Kooperieren tun unsere Konsistenzagenten nicht. Ihre Argumente haben sie aus den Änderungsabsichten ihrer Benutzer abgeleitet, d.h. die Menge $Coop$ ist leer und spielt bei unserer weiteren Diskussion keine Rolle. Argumentieren tun sie aber schon. Für jeden der Konsistenzagenten beinhaltet die Menge Arg alle anderen Konsistenzagenten, die in denselben Konfliktfall verwickelt sind. Die Menge Dom für jeden der Konsistenzagenten hat als Elemente alle Prädikate, die in ihren eigenen logischen Programmen auftauchen.

Um das Ganze verständlicher zu machen, werde ich für jeden Konfliktfall, der bereits in den Abschnitten 7.2.1 und 7.2.2 vorgestellt wurde, die Argumente der Agenten, die in diese Konfliktfälle verwickelt sind, vorstellen.

Komponentenebene Buch, Kapitel:

Konfliktfälle **delete-insert** und **delete-update**:

Ein KA, der ein Buch löschen möchte, hat das folgende Argument:
delete(Buch)

Ein KA, der ein Kapitel löschen möchte, hat das folgende Argument:
Delete(Buch.Kapitel)

Ein KA, der einen Absatz hinzufügen möchte, hat die folgenden Argumente:
insert(Buch.Kapitel.Absatz)

$\perp \leftarrow \text{delete}(\text{Buch})$ (Konsistenzregel)
 $\perp \leftarrow \text{delete}(\text{Buch.Kapitel})$ (Konsistenzregel)

Ein KA, der einen Absatz ändern möchte, hat die folgenden Argumente:
update(Buch.Kapitel.Absatz)

$\perp \leftarrow \text{delete}(\text{Buch})$ (Konsistenzregel)
 $\perp \leftarrow \text{delete}(\text{Buch.Kapitel})$ (Konsistenzregel)

Ein KA, der ein Kapitel hinzufügen möchte, hat die folgenden Argumente:
insert(Buch.Kapitel)

$\perp \leftarrow \text{delete}(\text{Buch})$ (Konsistenzregel)

Konfliktfall insert-insert:

Ein KA, der ein neues Buch oder ein neues Kapitel einfügen möchte, hat die folgenden Argumente `insert(Buch)` bzw. `insert(Buch.Kapitel)`. In diesem Fall gibt es keine Argumente, die aus der Konsistenzregel entstehen, weil man sie nicht in WFSX schreiben kann.

Komponentenebene Absatz:**Konfliktfälle delete-insert, insert-insert, insert-update:**

Auch in diesen Fällen existieren aus dem oben genannten Grund keine Argumente, die aus der Konsistenzregel entstehen. Konsistenzagenten, die einen Absatz löschen, ändern oder neu hinzufügen möchten, haben die folgenden Argumente:

`delete(Buch.Kapitel.Absatz)`, `update(Buch.Kapitel.Absatz)`, `insert(Buch.Kapitel.Absatz)`

Konfliktfall delete-update:

Ein KA, der einen Absatz löschen möchte, hat nur ein Argument:

`delete(Buch.Kapitel.Absatz)`

Ein KA, der einen Absatz ändern möchte, hat die folgenden Argumente:

`update(Buch.Kapitel.Absatz)`

$\perp \leftarrow$ `delete(Buch.Kapitel.Absatz)` (Konsistenzregel)

Konfliktfall update-update:

Beide Konsistenzagenten haben das gleiche Argument: `update(Buch.Kapitel.Absatz)`

8.3.2 Interaktion der Konsistenzagenten mit dem REVISE-System

Wie man auf der Abbildung 8.2 sehen kann, wurde der existierenden Struktur der MoleDDE durch das KES neben den Konsistenzagenten noch das REVISE-System hinzugefügt. Unter REVISE-System ist hier der ganze auf Abbildung 7.2 dargestellte Konfliktlösungsmechanismus zu verstehen. Dieser Abschnitt sollte als eine Fortsetzung der Beschreibung des Reintegrationsprozesses aus Abschnitt 8.3 verstanden werden, wobei der Fokus hier auf der Schnittstelle zwischen Konsistenzagenten und REVISE-System liegt.

Sobald von einem Konsistenzagenten festgestellt wird, dass ein Reintegrationsverfahren nötig ist, startet er das Konsens-Modul. Die Konsistenzagenten senden die Fakten, die ihre Benutzer geschaffen haben, zusammen mit den Konsistenzregeln und der Information, von welchen Konsistenzagenten noch Argumente zu erwarten sind, an das Konsens-Modul, wie in Abbildung 8.3 zu sehen ist. Das Konsens-Modul trägt in die Wissensbasis gleich nach dem Kontakt mit dem ersten Konsistenzagenten eine Zeile in Form eines Kommentars für jeden der Konsistenzagenten, mit dem es erwartet Kontakt aufzunehmen, ein. Das hilft uns im weiteren Verlauf im Fall eines Ausfalles des Konsens-Moduls selbst. Falls sich alle Konsistenzagenten gemeldet haben, gibt es in der Wissensbasis keine Kommentare mehr mit den Namen der Benutzer, von denen Argumente erwartet werden. Das Konsens-Modul wartet, bis es die Argumente von allen Konsistenzagenten gesammelt hat und trägt sie in die Wissensbasis des Konfliktlösungsmechanismus ein. Falls ein oder mehrere Konsistenzagenten ausfallen, macht er mit den Konsistenzagenten weiter, die sich gemeldet haben. Er informiert sie auch darüber, welche Konsistenzagenten ausgefallen sind. Die Benutzer dieser Konsistenzagenten werden genauso behandelt, wie die

Benutzer, die nach der Mehrheitsbildung vom Reintegrationsprozess ausgeschlossen wurden (siehe Abschnitt 8.3).

Die Wissensbasis ist nichts weiter als eine Datei, die in unserem Fall zu Anfang leer ist. Die Argumente der einzelnen Konsistenzagenten werden vom Konsens-Modul in Form von Revisables eingetragen (siehe Abschnitt 7.4.1). Die Konsistenzregeln werden in Form von Integritätsbedingungen eingetragen (siehe vorigen Abschnitt).

Wie wir im Abschnitt 7.4.2 gesehen haben, nimmt das REVISE-System als Eingabe ein erweitertes logisches Programm. In unserem Fall wird, sobald auch der letzte Konsistenzagent seine Fakten gesendet hat, die Wissensbasis geladen. Als Ausgabe erzeugt es die Revision, also die Lösungsmenge. Falls kein Konflikt existiert, oder mit den vorgegebenen Revisables die existierenden Konflikte nicht behoben werden können, gibt es einfach eine leere Menge zurück. Es ist Aufgabe des Konsens-Moduls zu unterscheiden, wann es keinen Konflikt gibt und wann die existierenden Konflikte nicht behoben werden können. Wie wir bereits im vorigen Abschnitt gesehen haben, kann man leider nicht alle Konsistenzregeln in WFSX ausdrücken. Verstöße gegen diese Konsistenzregeln können vom REVISE-System weder erkannt noch gelöst werden. Anhand der Argumente der Konsistenzagenten kann das Konsens-Modul feststellen, ob und wenn ja, welche Konsistenzagenten in welchen Konfliktfall verwickelt sind. Konsistenzagenten, die in keinen Konfliktfall verwickelt sind, werden darüber informiert. Konfliktfälle, die von dem REVISE-System nicht erkannt werden können, werden vom Konsens-Modul erkannt, das den betreffenden Konsistenzagenten auch die nötigen Anweisungen gibt. Der weitere Verlauf wurde bereits im Abschnitt 8.3 beschrieben, hier möchte ich nur noch sagen, dass nach dem Abschluss der Reintegration der Inhalt der Wissensbasis gelöscht wird.

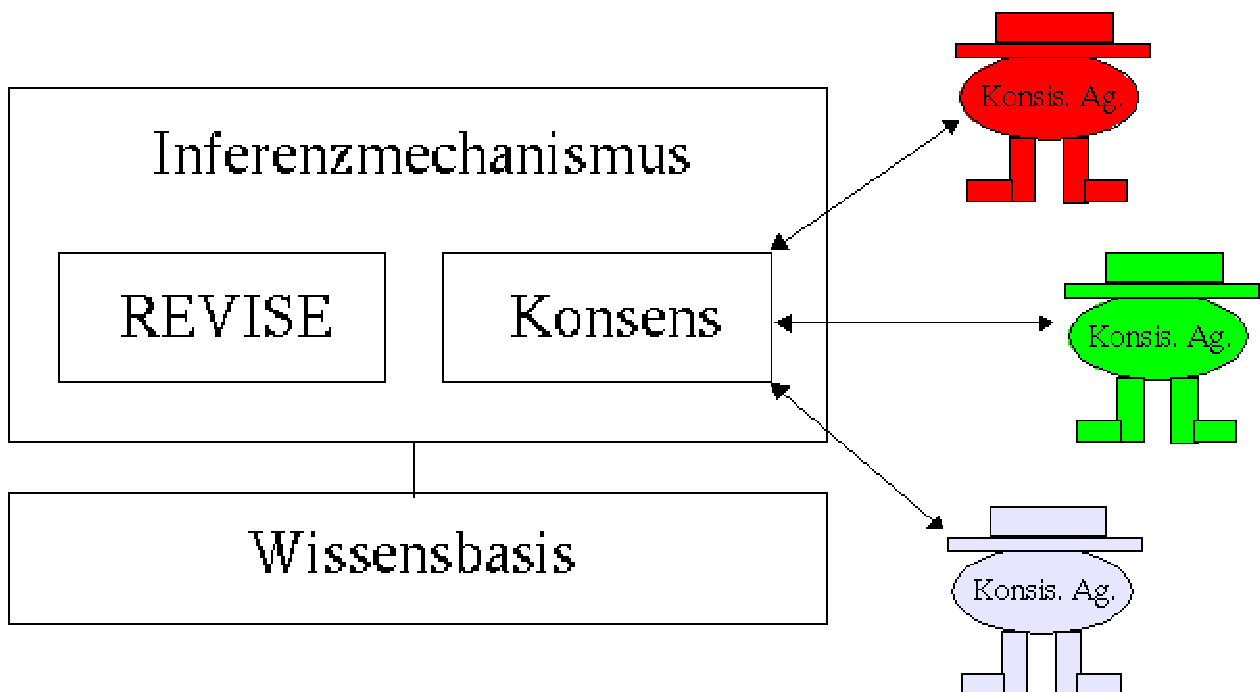


Abbildung 8.3: Interaktion zwischen Konsistenzagenten und REVISE-Systems

In Abbildung 8.3 kann man im Gegensatz zu Abbildung 7.2 die Schnittstelle zur Außenwelt nicht finden. Der Grund ist einfach der, dass die Schnittstelle hier als Teil des Konsens-Moduls betrachtet wird, oder andersherum kann man auch das Konsens-Modul als Teil der Schnittstelle zur Außenwelt betrachten.

9. Implementierung

Im Rahmen dieser Diplomarbeit hat keine Implementierung stattgefunden. Der Inhalt der folgenden Abschnitte sollte deshalb nicht als die Beschreibung eines existierenden Systems verstanden werden, sondern als eine Anleitung für die Implementierung der entsprechenden Komponenten.

9.1 Konsistenzagent

Der Konsistenzagent wurde bereits im Kapitel "Feinentwurf" ziemlich analytisch beschrieben. In diesem Abschnitt wird noch einmal die Aufgabe des Konsistenzagenten genauer betrachtet und es werden Einzelheiten beleuchtet, die im Feinentwurf keinen Platz hatten. In der Abbildung 9.1 kann man ein UML-Diagramm für den Konsistenzagenten und die Beziehung zu anderen Agentenklassen betrachten.

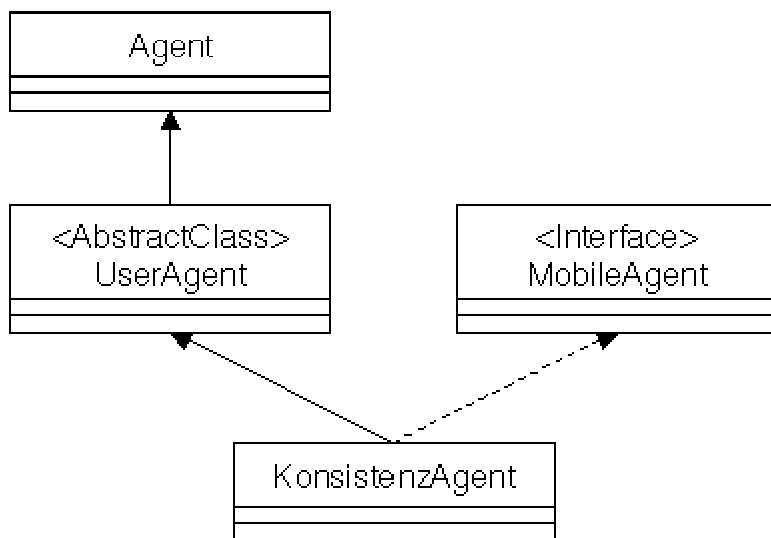


Abbildung 9.1: UML-Diagramm für den Konsistenzagenten

Wie bereits im Abschnitt 8.2 erwähnt, sollte man sich die Konsistenzagenten als eine Art Berater in Konsistenzfragen auf der Seite des DDE-Agenten vorstellen. Zu seinen Aufgaben gehören:

- Markierung der Objekte (Bücher, Kapitel), die von Benutzern entkoppelt bearbeitet werden sollen. Die Objekte werden von den entsprechenden Benutzern über die grafische Benutzungsoberfläche markiert. Die Informationen werden von den Benutzeragenten den Konsistenzagenten übertragen. Die Konsistenzagenten sind dafür zuständig, die betreffenden Objekte über DDE- und Dokumenten-Agenten zu markieren.
- Die Benutzerdaten, die der KA vom DDE-Agenten bekommt, haben ihre ursprüngliche Form, sind also Informationseinheiten. Der KA hat die Aufgabe diese Informationseinheiten "richtig" zu interpretieren, um mögliche Inkonsistenzen zu vermeiden. Interessiert ist der KA nur an Informationseinheiten, die entweder Dokumenten-

struktur oder Dokumenteninhalte ändern, also Informationseinheiten, die als Befehle `createBook`, `destroyBook`, `createSection`, `destroySection` und `writeSection` haben. Treten Informationseinheiten mit solchen Befehlen auf, hat der KA die Aufgabe, die entsprechenden Objekte daraufhin zu überprüfen, ob sie als geöffnet markiert sind. Sind die Objekte nicht markiert, leitet er die Informationseinheiten an den DDE-Agenten weiter. Sind die Objekte markiert, speichert er die Informationseinheiten in eine Datei und informiert den entsprechenden Benutzer über den DDE-Agent und die Editor-Schnittstelle darüber, dass ihre Daten nicht definitiv gespeichert wurden. Anschließend leitet er ein Reintegrationsverfahren ein.

- Während des Reintegrationsverfahrens spielt derjenige KA, der die Reintegration gestartet hat, auch eine Organisator-Rolle. Er informiert alle Konsistenzagenten, die an der Reintegration beteiligt sind, und er ist derjenige, der alle entkoppelt arbeitenden Benutzer per Email benachrichtigt, ihre Ausarbeitungen zu schicken. Er startet dann auch das Konsens-Modul (siehe nächster Abschnitt), um eine Argumentation der an der Reintegration beteiligten Konsistenzagenten zu ermöglichen.
- Jeder KA, der an einer Reintegration beteiligt ist, baut aus den oben erwähnten Informationseinheiten seine Argumente in WFSX (siehe Abschnitt 8.3.1). Für Strukturänderungen ist es leicht und kann anhand der Informationseinheiten geschehen. Für Inhaltsänderungen muss er den Inhalt des entsprechenden Kapitels über DDE- und Dokumenten-Agenten laden. Mittels einer Java-Portierung des GNU-Diff Tools kann er anschließend die beiden Inhalte vergleichen. Genauer gesagt vergleicht er einzelne Absätze der Kapitel. Er ist nicht an genauen Änderungen in jedem Absatz interessiert, sondern nur ob und wenn ja, welche geändert wurden
- Nach dem Abschluss der Reintegration, kann man drei Arten von Konsistenzagenten unterscheiden:
 - Konsistenzagenten, die gar nicht in einen Konflikt verwickelt waren, können einfach ihre vorübergehend gespeicherten Informationseinheiten an den DDE-Agenten weiterleiten und sie anschließend weglöschen. Die entsprechenden Benutzer brauchen gar nicht darüber informiert zu werden.
 - Konsistenzagenten, die in einen Konflikt verwickelt waren, der aber ohne Benutzerbeteiligung gelöst werden könnte. Das bedeutet, dass Änderungen der entsprechenden Benutzer rückgängig gemacht werden müssen, worüber die Benutzer informiert werden müssen. Anschließend können diese Informationseinheiten von ihrem Speicherplatz gelöscht werden.
 - Konsistenzagenten, die in einen Konflikt verwickelt waren, der nur mit Benutzerbeteiligung gelöst werden kann. Diese Konsistenzagenten informieren ihre Benutzer über die Art des Problems und darüber, welche anderen Benutzer noch daran beteiligt sind. Das Löschen der Informationseinheiten folgt auch in diesem Fall. Sie informieren auch die Benutzer, die ihre Replikat nicht schicken konnten.

Bevor ich diesen Abschnitt abschließen möchte, möchte ich noch einige Bemerkungen über die Möglichkeiten machen, die uns die vorhandene Dokumentenstruktur anbietet, Konflikte zu erkennen. Bereits im Abschnitt 4.2 wurde erwähnt, dass die Kapitelstruktur flach ist, worauf noch einmal im Abschnitt 7.1 aufmerksam gemacht wurde. Mit dieser flachen Struktur

ist es nicht möglich, Konsistenzprobleme, die als Ursache Strukturänderungen einzelner Kapitel haben, zu behandeln. Dazu sollte man eine strikte Namensgebung einführen, die bereits von der Editorumgebung überprüft wird. Eine solche Namensgebung könnte z.B. vorsehen, dass alle Kapitel und deren Unterkapitel nummeriert sind, dass die Anzahl der erlaubten Ebenen, die Nummerierung von jeder Ebene usw. vorgegeben ist. In der existierenden Mole-DDE existiert eine solche Dokumentenstruktur nicht, weshalb solche Konsistenzprobleme nicht behandelt werden können.

Ein weiterer Punkt, den man in Betracht ziehen sollte, sind die Absätze und deren Änderungen. Mit der existierenden Dokumentenstruktur ist es sehr schwer herauszufinden, welcher Absatz gelöscht und welcher eingefügt wurde. Falls die Anzahl der Absätze sowohl im Ursprung als auch im Replikat identisch ist, werden jeweils die Absätze an den entsprechenden Stellen verglichen. Was passiert aber z.B. wenn ein Absatz von einer Stelle in einem Abschnitt zu einer anderen Stelle innerhalb desselben Abschnitts verschoben wird? Ein Tool wie das Diff gibt uns kaum Möglichkeiten, solche Änderungen wahrzunehmen. Als Lösung kann man die Einführung einer weiteren Klasse vorschlagen, der Klasse `ddeAbsatz`. Diese neue Dokumentenstruktur muss auch von der Editorumgebung unterstützt werden.

9.2 Das Konsens-Modul

Bei dem Konsens-Modul handelt es sich um ein Java Programm, das von demjenigen Konsistenzagenten gestartet wird, der das Reintegrationsverfahren startet. Es läuft solange, bis die möglichen Konflikte zwischen den Konsistenzagenten entdeckt und die an den Konflikten beteiligten Agenten über die Lösung informiert werden.

Man sollte sich das Konsens-Modul als eine Art Vermittler zwischen den Konsistenzagenten, die an einer Reintegration beteiligt sind, vorstellen. Es sorgt für Konsens zwischen diesen Agenten, woher auch der Name kommt.

Das Konsens-Modul bildet die Schnittstelle des Konfliktlösungsmechanismus zu den Konsistenzagenten. Es ist der Empfänger sowohl der Argumente der Konsistenzagenten als auch der Konsistenzregeln. Seine Rolle beschränkt sich nicht nur auf das Eintragen dieser Daten in die Wissensbasis. Er überprüft die Argumente der Konsistenzagenten, um festzustellen, ob sie tatsächlich in einen Konflikt verwickelt sind. Das ist sehr einfach, wenn man bedenkt, dass die Argumente der Konsistenzagenten Änderungen am Dokument beschreiben. Welche Kombinationen von Änderungen Konfliktfälle sind, haben wir im Abschnitt 7.2 gesehen. Konsistenzagenten, die in keinen Konfliktfall verwickelt sind, werden vom Konsens-Modul anschließend informiert. In der Praxis heißt das, dass ihre Argumente unverändert bleiben.

Sind Konsistenzagenten in einen Konflikt verwickelt, muss man unterscheiden, ob für den Konfliktfall eine Konsistenzregel in WFSX existiert oder nicht. Falls nicht, ist das Konsens-Modul zuständig für die Konfliktlösung. Lösungen für Konfliktfälle, für die es keine Konsistenzregeln in WFSX gibt, sind also in dem Code des Konsens-Moduls implementiert. Für Konflikte, für die es Konsistenzregeln in WFSX gibt, trägt das Konsens-Modul deren Argumente und die Konsistenzregeln in die Wissensbasis ein. Anschließend startet es das REVISE-System und lädt die Wissensbasis. Die Antwort vom Revise-System wird dann vom Konsens-Modul an die Konsistenzagenten weitergeleitet.

Wie bereits im Abschnitt 8.3.2 erwähnt, trifft das Konsens-Modul sowohl für Ausfälle der Konsistenzagenten als auch für einen Ausfall des Moduls selbst die nötigen Maßnahmen.

9.3 Schnittstelle zwischen Java und SICStus-Prolog

Wie wir bereits im vorigen Abschnitt gesehen haben, ist für die Behandlung von Konflikten, für die Konsistenzregeln in WFSX existieren, das REVISE-System zuständig, ein System, das in Prolog implementiert ist und das unter SICStus-Prolog läuft. Das Konsens-Modul auf der anderen Seite, das das REVISE-System mit Informationen über Konflikte versorgt und das die Lösungsvorschläge an die Konsistenz-Agenten weiterleitet, ist ein Java-Objekt. Aus diesem Grund ist die Verbindung von Java- und Prolog-Programmen unabdingbar. Zu diesem Zweck dient uns die Jasper-Bibliothek, eine Bibliothek, die im Rahmen von SICStus-Prolog ab Version 3.7 existiert.

Das größte Problem in unserem Fall ist allerdings, dass Jasper erst ab JDK 1.2 lauffähig ist, sowohl für Sun Solaris 2.x (SPARC und x86) als auch für Linux (x86) und Windows (95/98/NT/2000). Das stellt ein Kompatibilitätsproblem dar, weil die existierende Mole-DDE mit JDK 1.1 implementiert ist.

Jasper ist eine bidirektionale Schnittstelle zwischen Java und SICStus. Die Schnittstelle besteht auf Seite von Java aus einem Java-Paket (`se.sics.jasper`), welches Klassen beinhaltet, die das SICStus Runtime-System darstellen. Der Prolog-Teil ist als ein Bibliotheksmodul (`library(jasper)`), das zur Erweiterung der Schnittstelle zu anderen Programmiersprachen entworfen wurde. Jasper kann abhängig davon, welches System als "Vater-Anwendung" agiert, in zwei verschiedenen Arten verwendet werden.

Wenn Java die "Vater Anwendung" ist, wird der SICStus Runtime-Kern in die JVM durch die Verwendung der `System.loadLibrary()`-Methode geladen (dies geschieht indirekt während der Initialisierung des SICStus-Objekts). In diesem Modus wird SICStus als Runtime-System geladen. Wenn SICStus die "Vater-Applikation" ist, wird Java durch Verwendung der Anfrage `use_module(library(jasper))` als eine fremde Resource geladen. Die Java-Engine wird durch `jasper_initialize/[1-2]` initialisiert.

Die Schnittstelle von Prolog zu Java interessiert uns im Rahmen dieser Diplomarbeit nicht und wird uns im weiteren nicht mehr beschäftigen. Dieser Teil über die Jasper-Schnittstelle ist unterteilt in zwei Abschnitte, wobei sich der erste mit der Installation von Jasper beschäftigt, der zweite mit den Kenntnissen, die man haben sollte, um Jasper sinnvoll einzusetzen.

9.3.1 Jasper zum Laufen bringen

Die Installation der Jasper-Bibliothek ist eigentlich, was die Verwendung dieser Software angeht, der Teil, der die meisten Probleme mit sich bringt. In diesem Abschnitt werden wir die wichtigsten Punkte betrachten, um die Jasper-Bibliothek unter einem Unix-Betriebssystem zum Laufen zu bringen.

Es gibt einige wichtige Punkte, die man beachten sollte, und die über Funktionieren oder Nicht-Funktionieren der Jasper-Bibliothek entscheiden. Der erste wichtige Punkt ist es,

den CLASSPATH richtig zu spezifizieren, damit Java die Jasper-Klassen finden kann. Dies geschieht, indem man den Pfad für die Datei `jasper.jar` richtig setzt. Der zweite ist die Festlegung, wo Java die native Bibliothek (`libjasper.so`) finden soll. Diese Bibliothek wird von der SICStus-Klasse in die JVM geladen, indem sie die Methode `System.loadLibrary("jasper")` aufruft. Diese Methode verwendet eine plattformabhängige Suchmethode, um die Jasper-Native-Bibliothek zu finden und nicht selten schlägt sie fehl. Dies kann vermieden werden, indem man in der Java-Eigenschaft `java.library.path` explizit die Lokation von `libjasper.so` folgendermaßen eingibt:

```
% java -Djava.library.path=/usr/local/lib [...]
```

Wenn Jasper in Runtime-Systemen verwendet wird, existieren noch zusätzliche Beschränkungen, deren Erwähnung ich hier aber nicht für sinnvoll halte¹.

Wenn alles richtig eingestellt wurde, sollte SICStus in den Adressraum der JVM geladen werden. Das einzige, was übrig bleibt, ist SICStus zu sagen, wo die Runtime-Bibliothek (d.h. `sprt.sav`) ist. Man kann dies entweder explizit machen, indem man während der Initialisierung des SICStus-Objektes ein zweites Argument mitgibt oder durch Spezifikation der Eigenschaft `sicstus.path`:

```
% java -Dsicstus.path=/usr/local/lib/sicstus-3.8
```

Wenn man keinen expliziten Pfad spezifiziert, wird SICStus von alleine nach der Runtime-Bibliothek suchen. Wenn alles richtig eingestellt ist, sollte man in der Lage sein, die Methode `main()` in der SICStus-Root-Klasse aufzurufen (welche einen kleinen Teil Test-Code enthält). Als Ergebnis bekommt man folgendes zu sehen:

```
% java -Djava.library.path="/usr/local/lib" \
-Dsicstus.path="/usr/local/lib/sicstus-3.8" \
-classpath "/usr/local/lib/sicstus-3.8/bin/jasper.jar" \
se.sics.jasper.SICStus
Trying to load SICStus.
If you see this message, you have successfully
initialized the SICStus Prolog engine.
```

9.3.2 Prolog von Java aus aufrufen

Wie bereits erwähnt kann man Prolog von Java aus aufrufen, indem man das Java-Paket Jasper verwendet. Das Paket enthält eine Reihe von Klassen, mit denen man Terme kreieren und manipulieren kann, Fragen stellen und mehrere Antworten verlangen kann. Am einfachsten kann man die Verwendung anhand eines Beispiels verstehen. In unserem Fall heißt das Prolog-Programm, das das REVISE-System lädt `boot`. Wie genau der Inhalt des Programms ist, interessiert uns nicht, was uns interessiert ist, dass der Befehl `read_file/1` die Wissensbasis lädt und dass der Befehl `solution/1` die Lösungen zurückgibt.

Ein Java-Programm, das dieses Prolog-Programm aufruft, könnte folgendermaßen aussehen:

```
import se.sics.jasper.*;
public class Simple
{
    public static void main(String argv[]) {
```

¹ Für zusätzliche Informationen siehe Manual von SICStus Prolog

```

SICStus sp;
SPTerm wissensbasis, loesung;
SPQuery query;
int i;

try {
    sp = new SICStus(argv,null);

    sp.restore("boot");

    wissensbasis = new SPTerm(sp, "Wissensbasis");
    loesung = new SPTerm(sp).putVariable();

    // Loesung:solution(loesung).
    query = sp.openQuery("Loesung", "solution",
        new SPTerm[] { loesung });

    try {
        while (query.nextSolution()) {
            System.out.println(loesung.toString());
        }
    } finally {
        query.close();
    }
}
catch ( Exception e ) {
    e.printStackTrace();
}
}
}

```

Dieses Beispiel funktioniert folgendermaßen:

1. Bevor beliebige Prädikate aufgerufen werden können, muss das SICStus-Runtime-System initialisiert werden. Dies geschieht, indem man ein Objekt der SICStus-Klasse erzeugt. Diese Klasse muss allerdings nur einmal für jeden Java-Prozess instanziiert werden. Mehrere SICStus-Objekte werden nicht unterstützt. In diesem Beispiel ist das zweite Argument von SICStus NULL, was eigentlich bedeutet, dass SICStus mit Hilfe seiner eigenen internen Methoden oder durch die Verwendung eines Pfades, welcher durch `-Dsicstus.path=[...]` spezifiziert wurde, nach der Datei `sprt.sav` suchen soll. Die meisten Methoden haben als erstes Argument das SICStus-Objekt.
2. Der nächste Schritt ist das Laden des Prolog-Codes. Dies geschieht durch die Methode `restore()`. Jetzt ist alles bereit, um Prolog-Anfragen zu stellen.
3. Um Anfragen zu stellen, muss man erst die Argumente für die Anfragen kreieren. Die Argumente werden in einem Array gespeichert, welches der entsprechenden Methode übergeben wird, um die Anfrage zu stellen. Die Argumente bestehen aus Objekten der Klasse `SPTerm`. Wenn man z.B. ein Atom und eine Variable für die Anfrage

```
| ?- solution( Loesung ).
```

braucht, dann verwendet man den folgenden Java-Code:

```
wissensbasis = new SPTerm(sp, "Wissensbasis");  
loesung = new SPTerm(sp).putVariable();
```

4. Die Anfrage selbst kann man auf drei verschiedenen Arten stellen. Entweder interessiert man sich für eine Lösung (`SICStus.query, ...`) oder für Seiteneffekte (`SICStus.queryCutFail, ...`) oder, wie es hier der Fall ist, für mehrere Lösungen durch Backtracking (`SICStus.openQuery, ...`). Die Methode `openQuery` gibt eine Referenz zu einer Anfrage zurück, ein Objekt der Klasse `SPQuery`. Um Lösungen zu erhalten, wird die Methode `nextSolution` ohne Argumente aufgerufen. Sie gibt als Antwort immer `TRUE` zurück, solange noch weitere Antworten existieren. Das obere Beispiel wird solange Werte der Variablen `Loesung` ausgeben, bis keine weiteren Lösungen mehr existieren. Man sollte beachten, dass die Anfrage abgeschlossen werden muss, auch wenn `nextSolution` angezeigt hat, dass es keine weiteren Lösungen gibt.

Weitere wichtige Punkte für den Einsatz der Schnittstelle `Jasper` sind die Typumwandlung von Prolog-Argumenten in Java-Typen und umgekehrt, die Behandlung von Ausnahmefällen (`exception handling`) und die Referenz von Klassen des `Jasper`-Pakets. Für detailliertere Informationen über die `Jasper`-Bibliothek sollte man die Dokumentation von `SICStus-Prolog` betrachten.

10. Testszzenarien

Ziel von Testszzenarien für Software-Produkte ist, genauso wie für jedes andere Produkt, nicht der Beweis, dass der zu testende Gegenstand richtig funktioniert, sondern die Entdeckung von möglichen Schwachpunkten, Fehlern und Abweichungen von der Spezifikation [LuJo97]. Deswegen sollte man während eines Tests das Produkt unter möglichst hoher Belastung stellen, allerdings nur im Rahmen der von dem Produkt zu erwartenden Funktionsbedingungen und es sollten möglichst alle seine Eigenschaften überprüft werden.

In unserem Fall handelt es sich allerdings um eine prototypische Realisierung der Konzepte und nicht um ein richtiges Software-Produkt. Aus diesem Grund werden auch nicht alle Eigenschaften, die ein Software-Produkt aufweist, wie z.B. Verfügbarkeit, Robustheit, Portabilität (obwohl das System als Java-Implementierung über eine hohe Portabilität verfügt), Benutzerfreundlichkeit usw. getestet. Es wird hauptsächlich die Funktionalität der Implementierung geprüft. Dabei kann man wiederum zwischen zwei Gesichtspunkten unterscheiden. Der erste Gesichtspunkt betrifft die Übereinstimmung der Implementierung mit der Spezifikation. Falls die Übereinstimmung gegeben ist, kann man auch bemessen, wie gut das System das von ihm geforderte erfüllt. Das ist der zweite Gesichtspunkt, der außer der Berücksichtigung verschiedener Eingabeparameter auch eine Metrik zur Bewertung der Ergebnisse einführt. In unserem Fall interessiert uns eine solche Metrik nicht. Es existiert aber bereits ein anderes System [ScPe99], das eine sehr ähnliche Funktionalität aufweist. Dieses System kann als Maßstab dafür dienen, wie gut oder wie schlecht unser System ist.

In den Testszzenarien, die ich im Weiteren vorschlage, versuche ich, sowohl beide Gesichtspunkte als auch möglichst viele Parameter, die bei der Bearbeitung eines Dokuments eine Rolle spielen könnten, zu berücksichtigen. Weil ein oder mehrere Parameter fast in jedem der folgenden Testszzenarien betrachtet werden sollten, erachte ich es als sinnvoll, zunächst die Parameter zu präsentieren:

- 1) Die Anzahl der Bearbeiter ist ein wichtiger Parameter für jedes TestszENARIO. Im Abschnitt 5.1.1 haben wir bereits die Annahme gemacht, dass Teams selten aus mehr als zehn Mitgliedern bestehen. Hier drücken wir diese Zahl noch mal nach unten und machen die Annahme, dass höchstens bis zu fünf Personen mit unserem System arbeiten werden. Darüber hinaus haben wir bis jetzt immer zwischen gekoppelter und entkoppelter (für mobile Benutzer) Arbeitsweise unterschieden. Die Kombination der Anzahl der Benutzer und ihrer Arbeitsweise erscheint mir als der wichtigste Parameter für alle Testszzenarien.
- 2) Der zweite Parameter, der betrachtet werden sollte, ist die Größe der Dokumente. Obwohl für Dokumente eine Angabe darüber, was die reguläre Größe sein könnte, schwer zu machen ist, machen wir für unser KES die Annahme, dass sie meistens eine Größe zwischen einigen und einigen Hundert kB haben könnten. Diese Annahme ist daher gerechtfertigt, dass die Mole-DDE in der jetzigen Form keine multimedialen Dokumente unterstützt. Größere Dokumente spielen in unseren Testszzenarien eine besondere Rolle, wenn es um inhaltliche Änderungen geht. Und somit kommen wir zu unserem nächsten Parameter, dem Umfang der Änderungen.

- 3) Unter Änderungen sollte man hier wieder sowohl Struktur- als auch Inhaltsänderungen der Dokumente betrachten, wie sie im Abschnitt 7.1 definiert wurden. Aber auch jegliche Kombination dieser beider Änderungsarten sollte als möglich erachtet werden.
- 4) Konsistenzregeln und Konfliktfälle können auch als Parameter betrachtet werden. In den Testszzenarien sollte man versuchen, jeden einzelnen Konfliktfall und möglichst viele Kombinationen von ihnen abzudecken.

Nun ist es Zeit, nach der Präsentation der Parameter auch zu den Testszzenarien zu kommen. Meiner Meinung nach wären die folgenden Testszzenarien sinnvoll:

- Bearbeitung eines Dokuments ohne Konfliktfälle, sowohl für einen als auch für mehrere Benutzer. Die anderen Parameter, wie Größe der Dateien oder Änderungen, kann man variieren, muss man aber nicht. Dieses Szenario soll einfach die Frage beantworten, ob sich das System genauso wie vorher verhält, bevor also das KES implementiert wurde.
- Ein TestszENARIO mit Variation der Konfliktfälle, der Änderungen an einem Dokument, der Größe des Dokuments und der Anzahl der Benutzer und ihrer Arbeitsweisen. Dieses TestszENARIO soll die Funktionalität des KES unter normalen Bedingungen testen. Dabei werden möglichst viele Parameter variiert, um möglichst viele Fälle abzudecken.
- Ein TestszENARIO, um das KES unter Randbedingungen zu testen. Solche Randbedingungen sind z.B. das Überschreiten der Zeitgrenze von 24 Stunden nach der ersten Ausarbeitung oder ein Server-Absturz. Ziel ist es, herauszufinden, wie sich das System in diesen Fällen verhält. Läuft tatsächlich nach dem Ablauf der 24 Stunden alles so wie geplant, oder passiert etwas anderes als das Erwünschte? Was geschieht während eines Server-Absturzes oder während der Server wieder hochgefahren wird? Bekommen die Benutzer, die ihre Daten übertragen wollen, die richtigen Fehlermeldungen, oder leben sie mit der Illusion, dass alles gut geklappt hat, obwohl nichts funktioniert hat? Ist eine Fortsetzung der Reintegration nach dem Hochfahren des Servers möglich? Wenn nein, warum nicht, was kann man hier besser machen?

In den obigen Szenarien wurde das KES stets als ein Ganzes betrachtet, was aber nicht immer sinnvoll ist. In Fällen z.B., in denen die Funktion des Systems von der erwünschten abweicht, ist es durchaus sinnvoll, es in zwei Teile zu unterteilen. Der eine Teil wären dann die Agenten und der andere der Konfliktlösungsmechanismus. Eine solche Trennung schafft einen besseren Überblick darüber, wo die Probleme liegen könnten, ist aber ungeeignet für Schlussfolgerungen über das gesamte System.

11. Zusammenfassung und Bewertung

In dieser Diplomarbeit wurde das Potential der mobilen Agenten in Bezug auf Konsistenz-erhaltung untersucht. Ziel war herauszufinden, welche Möglichkeiten mobile Agenten, die miteinander verhandeln können, bieten, inkonsistente Zustände eines gemeinsam zu bearbeitenden Dokuments zu erkennen und zu vermeiden. Zu diesem Zweck wurden für die existierende Mole-DDE geeignete Konzepte zur Konsistenzverhandlung zwischen mobilen Agenten entworfen.

Die Konsistenz wurde mit Hilfe von Konsistenzregeln definiert, die in WFSX geschrieben wurden. Das REVISE-System, ein Konfliktlösungsmechanismus für WFSX, gibt den mobilen Agenten der Mole-DDE die Möglichkeit, miteinander zu verhandeln. Diese verhandlungsfähigen Agenten können tatsächlich durch Zusammenarbeit inkonsistente Zustände entdecken und beheben, und sie erlauben zusätzlich die Anpassbarkeit der Konsistenzregeln während die DDE läuft. Darüber hinaus gestalten sie die Erweiterung des Konsistenz-erhaltungssystems um zusätzliche Konsistenzregeln relativ unproblematisch.

Man kann allgemein betrachtet die Intelligenz, die für die Konsistenz-erhaltung nötig ist, in zwei Arten untergliedern:

- Intelligenz, um inkonsistente Zustände zu entdecken und
- Intelligenz, um Inkonsistenzen zu beheben

Für beide Arten der Intelligenz erlaubt unser Konzept des Konfliktlösungsmechanismus entweder, sie in den Code einzubauen oder sie mit Hilfe eines intelligenten Systems wie dem REVISE-System zu implementieren. Intelligenz, die von einem solchen System wie dem REVISE-System realisiert wird, kann als eine höhere Art der Intelligenz eingestuft werden, weil sie viel leichter zu implementieren (man definiert die Regeln in WFSX) und zu ändern ist (man ändert diese Regeln). Leider lässt sich aber nicht jede Konsistenzregel in WFSX schreiben und selbst wenn so etwas möglich wäre, wäre trotzdem eine gewisse Portion von Intelligenz im Code nötig.

Diese fest im Code gebundene Intelligenz ist der Grund, der die Vorstellung eines KES, bei dem man einfach zusätzlich neue Konsistenzregeln in WFSX eingeben oder bereits existierende einfach modifizieren kann, zu einer Illusion macht. Eine Änderung des Codes ist leider in vielen Fällen unvermeidbar.

In dem Entwurf, der in dieser Arbeit präsentiert wurde, ist sämtliche Intelligenz des KES von nur zwei, vor allen Dingen neu hinzugekommenen Modulen realisiert, dem Konsistenzagenten und dem Konfliktlösungsmechanismus. Das erlaubt einen hohen Grad an Wiederverwendbarkeit, Wartbarkeit und Erweiterbarkeit des KES. Die Änderungen in der Struktur der existierenden Mole-DDE waren also gering.

Gegenüber dem CORBA-Ansatz für Konsistenz-erhaltung, der in [ScPe99] präsentiert wurde, bietet der in dieser Arbeit präsentierte Ansatz der mobilen Agenten zweifellos weitere, darüber hinausgehende Möglichkeiten.

12. Ausblick

Die Möglichkeit zur Erweiterung, die unser Entwurf bietet, wurde bereits im vorigen Kapitel betont. Hier werden Gedanken präsentiert, welche Erweiterungen des Entwurfs möglich und meiner Meinung nach auch sinnvoll sind.

Außer einigen replikatübergreifenden Konsistenzregeln, die hier behandelt wurden, kann man auch andere Konsistenzregeln definieren. Solche Konsistenzregeln sind meist an die Art des Dokuments gebunden. Eine Erweiterung der existierenden mole-DDE um verschiedene Dokumentenarten, wie z.B. Veröffentlichungen, Projektanträge, Software-Dokumentation und ein entsprechendes Hinzufügen von Konsistenzregeln, die das Umgehen mit solchen Dokumentenarten erleichtern, sind durchaus vorstellbar. Besonders die Unterstützung von XML-Dokumenten und damit verbunden von beliebigen auch multimedialen Dokumenten, erscheint in meinen Augen als die größte Herausforderung. Darüber hinaus kann man sich auch dokumentenübergreifende Konsistenzregeln, die zuständig für die Konsistenzerhaltung der Informationen in verschiedenen Dokumenten sind, vorstellen.

Bereits am Anfang dieser Arbeit wurde darauf aufmerksam gemacht, dass verschiedene Rollen innerhalb eines Teams nicht berücksichtigt werden. Eine Reintegration von Replikaten könnte auch diesen Parameter in Betracht ziehen, sodass die Änderungen von Benutzern mit verschiedenen Rollen unterschiedliche Gewichtung haben könnten. Einige der in dieser Arbeit betrachteten Konfliktfälle wären dann keine mehr, allerdings könnten dadurch neue Konfliktfälle entstehen. Verschiedene Rollen eröffnen durchaus neue Möglichkeiten bezüglich der Behandlung von Inkonsistenzen, führen aber auch zu einer höheren Komplexität der Konsistenzbehandlung.

Für die Wartezeit innerhalb der alle Benutzer ihre Ausarbeitungen schicken müssen, wurde in unserem Reintegrationsszenario ein Zeitraum von 24 Stunden festgelegt. Dieser Zeitraum könnte auch ein Verhandlungsgegenstand unter den Konsistenzagenten sein.

Am Anfang war geplant, die Schnittstelle zwischen der Editorumgebung und dem DDE-System, durch einen Konsistenzagenten, der von der Editorumgebung gestartet wird und zum DDE-System wandert, zu ersetzen. Diese Idee habe ich im Laufe der Arbeit aufgegeben, weil dadurch eine Unterstützung von Thin-Clients unmöglich wird. Die Schnittstelle ist trotzdem nicht die ideale Lösung. Besonders bei der Übertragung von größeren Dokumenten erweist sie sich als Flaschenhals. Das Weglassen des Teils des Wrappers, der in Java implementiert ist, d.h den ganzen Wrapper in Emacs-Lisp implementieren, würde das System vereinfachen und auch schneller machen.

Das ganze System, also Mole-DDE und KES, sollte ein verteiltes System sein. Die Realität sieht anders aus. Zwar dürfen die Benutzeragenten auf verschiedenen Rechnern laufen, der Kern des DDE-Systems aber, also DDE-, Dokumenten-Agent und Dokumentenstruktur, existiert nur einmal. Dieser Gegebenheit hat sich unser KES angepasst. Es existieren zwar mehrere Konsistenzagenten, treffen tun sie sich aber nur auf einem Rechner. Eine Verbesserung der Lastbalancierung und vor allem Ausfallsicherheit ist durchaus vorstellbar.

Literaturverzeichnis

- [BeMi99] Berger Michael. Generic support of autonomous work and reintegration in the context of computer supported team work (in german). PhD thesis, Fakultät Informatik, Technische Universität Dresden, 1999.
- [BeScVö96] M. Berger, A. Schill, G. Völksen. Supporting Autonomous Work and Reintegration in Collaborative systems. In Coordination Technology for Collaborative Applications-Organizations, Processes and Agents. W. Conen, G. Neumann (Editors). New York, Springer-Verlag, 1998. ISBN 3-540-64170-X, pp. 177-198.
- [BuCo97] Burger Cora. Groupware-Kooperationsunterstützung für verteilte Anwendungen. dpunkt, Heidelberg, 1997.
- [BuCo00] Burger Cora. A comparison off different platforms for a collaborative distributed document environment. MAMA 2000.
- [BüSc88] Kl. Büning, Schmitgen. PROLOG, Teubner-Verlag Stuttgart, 1988.
- [CoCo94] I. M. Copi and C. Cohen. Introduction to Logic. Prentice Hall, 1994.
- [DaPeSc97] C.V. Damasio, L.M. Pereira, M. Schröder. REVISE: Logic Programming and Diagnosis. Proceedings of LPNMR97, Springer Verlag.
- [EbSa97] M. R. Ebling and M. Satyanarayanan. Translucent Cache Management for Mobile Computing.
- [HoJo93] Howard H. John. Using Reconciliation to Share Files Between Occasionally Connected Computers. WWOS-IV, Fourth Workshop on Workstation Operating Systems, Napa, California, October 14-15 1993: pages 56-60.
- [HuSi98] M. N. Huhns and M.P. Singh, 1998. "Cognitive agents," IEEE Internet Computing, volume 2, number 6, pp. 87-89.
- [ImBa93] T. Imielinski and B.R. Badrinath. Data Management for Mobile Computing. SIGMOD RECORD, Vol. 22, No. 1, March 1993.
- [KrSyEv98] S. Kraus, K. Sycara and A. Evenchik. Reaching agreements through argumentation: a logical model and implementation. Artificial Intelligence, 1998. To appear.
- [LePa97] Levi Paul. Skript zur Vorlesung "Grundlagen der verteilten künstlichen Intelligenz und der Bildverarbeitung". Universität Stuttgart, 1997.
- [LuJo97] Ludewig Jochen. Skript zur Vorlesung "Grundlagen des Software Engineering". Stuttgart, 1997.

Literaturverzeichnis

- [NoSa95] B. D. Noble and M. Satyanarayanan. A Research Status Report for Mobile Data Access.
- [PaGuHe97] T. Page, R. Guy, J. Heidemann, D. Ratner, P. Reiher, A. Goel, G. Kuenning and G. J. Popek. Perspectives on Optimistically Replicated Peer-to-Peer Filing. *Software-Practice and Experience*, Vol 28(2), pages 155-180, Dec. 1997 / Feb. 1998.
- [PaSiJe98] S. Parsons, C. Sierra and N. Jennings. Agents that reason and negotiate by arguing. *J. of Logic and Computation*, 8(3): 261-292, 1998.
- [PeAl92] L. M. Pereira and J.J. Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann (Ed.), *European Conference on Artificial Intelligence*, pages 102-106. John Wiley & Sons, 1992.
- [RaGe95] A. Rao and M. Georgeff, 1995. "BDI agents: From theory to practice,". In: *ICMAS-95: First International Conference on Multi-Agent Systems: July 12-14, 1995, San Francisco, California: Proceedings*. Menlo Park, Calif.: AAAI Press; Cambridge, Mass.: MIT Press, pp. 312-319.
- [RaPoRe96] D. Ratner, G.J. Popek, P. Reiher. The Ward Modell: A Scalable Replication Architecture for Mobility.
- [RaRePo97] D. Ratner, P. Reiher, and G. J. Popek. Replication Requirements in Mobile Environments. *Proceedings of the Dial M for Mobility Workshop*, Oct. 1997.
- [RaRePo97b] D. Ratner, P. Reiher, and G. J. Popek. Dynamic Version Vector Maintenance. *University of California Tech Report CSD-970022*.
- [ReHeRa94] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner and G. J. Popek. Resolving File Conflicts in the Ficus File System. In *USENIX Conference Proceedings*, June 1994, pages 183-195.
- [RePoGu96] P. Reiher, G. J. Popek, M. Gunter, J. Salomone, D. Ratner. Peer-to-Peer Reconciliation for Mobile Computers. *European Conference on Object Oriented Programming '96 Second Workshop on Mobility and Replication*, June 1996.
- [ScMi99] Schröder, Michael. An efficient argumentation framework for negotiating autonomous agents. In *Proceedings of the Workshop on Modelling Autonomous Agents in a Multi-Agent World MAAMAW99*. Valencia Spain, Springer-Verlag, July 1999.
- [ScPe99] Schurr Peter. Erweiterung von DDE (Distributed Document Environment) um Konsistenzbehandlung in mobiler Umgebung. *Diplomarbeit, Universität Stuttgart*, November 1999.
- [Sör96] M. G. Sörensen. A Model for Multi-Level Consistency.

Literaturverzeichnis

- [StRa99] Steinmetz Ralf. Multimedia Technologie, Grundlagen, Komponenten und Systeme. Springer Verlag 1999.
- [SuEI98] C. Sun, C. Ellis. Operational Transformation in Real-Time Group Editors: Issues, Algorithms and Achievements. CSCW 98 Seattle Washington USA pages 59-68, ACM 1998.
- [TaCI98] P. Tarr and L.A. Clarke. Consistency Management for Complex Applications. Proceedings 20th International Conference on Software Engineering (ICSE '98), May 1998.
- [WeLu00] Weberruß Lukas. Erweiterung von MoleOffice um gemeinsame Dokumentenbearbeitung. Diplomarbeit, Universität Stuttgart, April 2000.
- [WoJe94] M.J. Wooldridge and N.R. Jennings (Editors), 1995. Intelligent Agents: ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Amsterdam, the Netherlands, August 8-9, 1994: Proceedings. New York:Springer.
- [WoMi99] M. J. Wooldridge. Intelligent Agents, MIT Press, Cambridge 1999.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und
nur die angegebenen Quellen benutzt zu haben

(Georgios Tzizlis)