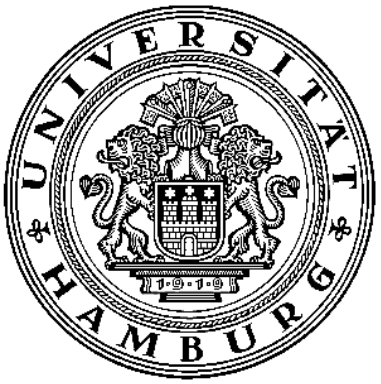

Change Management für Metadaten in Digitalen Bibliotheken

Diplomarbeit

Christoph Spreen
Fachbereich Informatik
Universität Hamburg

Betreuung:

Prof. Dr. Joachim W. Schmidt
Arbeitsbereich Softwaresysteme
Technische Universität Hamburg-Harburg

Prof. Dr. Christopher Habel
Arbeitsbereich Wissens- und
Sprachverarbeitung
Universität Hamburg

24. Februar 2000

Zusammenfassung

Die von Digitalen Bibliotheken zur Verfügung gestellten Informationsressourcen unterliegen Änderungen, von denen auch deren Metadaten betroffen sind. Die Handhabung dieser Veränderungen erfordert den Einsatz eines Systems zum Change Management, welches das Stattfinden von Änderungen erkennt, über relevante Änderungen informiert und entsprechende Reaktionen erlaubt.

Ausgangspunkt dieser Arbeit ist das Change Management für semistrukturierte Daten, wie man sie für die Darstellung von Metadaten verwenden kann. Sie lassen sich durch beschriftete Bäume repräsentieren.

In der Arbeit wird ein Verfahren entwickelt, welches zwei beschriftete Bäume miteinander vergleicht und deren Differenzen in Form einer Folge von Änderungsoperationen ermittelt. Derartigen *edit-scripts* liegt ein Operationen-Repertoire zugrunde, welches das *Ändern, Einfügen, Entfernen, Verschieben, Kopieren* und *Einschmelzen* von Teilbäumen umfaßt, wodurch die Änderungsfolgen eine hohe Aussagekraft erhalten.

Das entwickelte Verfahren wird von einem konfigurierbaren Dienst implementiert, dessen Informationsaustausch in Bezug auf die zu vergleichenden Metadaten und die erzeugten *edit-scripts* auf der Grundlage von XML (*extensible markup language*) erfolgt, wodurch weite Anwendungsbereiche für den Dienst und eine Integration des Dienstes in andere Anwendungen ermöglicht werden.

Danksagung

Das Gelingen dieser Arbeit wäre ohne die Hilfe vieler Personen nicht möglich gewesen. Ihnen möchte ich an dieser Stelle herzlich danken.

Mein spezieller Dank gilt Herrn Prof. Dr. Joachim W. Schmidt für die Unterstützung bei der Durchführung der Arbeit und die Bereitstellung der dafür verwendeten Hilfsmittel sowie Herrn Prof. Dr. Christopher Habel für die zusätzliche Betreuung dieser Arbeit.

Bei Frau Claudia Niederée und Frau Ulrike Steffens möchte ich mich für die vielen Anregungen, die positive Kritik und vor allem dafür bedanken, daß sie immer Zeit für mich und meine Arbeit gefunden haben.

Den Mitarbeiten und Studenten des Arbeitsbereichs Softwaresysteme danke ich für das ausgesprochen angenehme Arbeitsklima, dem ich es zu verdanken habe, daß diese Arbeit auch viel Freude gemacht hat.

Für viele Anregungen aus dem Bereich der Graphentheorie und theoretischen Informatik möchte ich mich bei meinem Schwager Dr. Reinhard Franzkeit und Herrn Berndt Farwer bedanken.

Den einen oder anderen Fehler der deutschen Rechtschreibung konnten meine Mutter Gisela Spreen und ein Praktikant des Arbeitsbereichs Softwaresysteme Felix Jaacks ausfindig machen, wofür ich beiden herzlich danke.

Ein besonderer Dank gilt meiner Familie: Für die Möglichkeit, dieses Studium durchzuführen, danke ich meinen Eltern sehr. Es ist sehr schade, daß sich mein Vater nicht mehr über diese Arbeit freuen kann. Vor allem aber möchte ich meiner Freundin Natascha für ihr Vertrauen, ihre Liebe, aber auch für ihre Nachsicht der vergangenen zwei Jahre danken.

Hamburg, 24. Februar 2000 Christoph Spreen

Der in der Informatik verwendete Fachterminus entstammt größtenteils dem Englischen, weswegen es nicht leicht ist, einheitlich deutschsprachige Arbeiten in Informatik zu erstellen. Da es nicht immer möglich ist, anstelle gebräuchlicher englischer Fachausdrücke deutsche zu verwenden, erscheinen englische Worte auch im fließenden Text. Sie werden wie wichtige Begriffe und Bezeichnungen von Produkten durch *kursive Schriftweise* hervorgehoben. Die Namen von Personen, Firmen und Organisationen werden in KAPITÄLCHEN gesetzt, während Fragmente von Programmen, wie etwa Methodennamen in `Schreibmaschinenschrift` erscheinen.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 2
- 1.2 Aufgabenstellung . . . 2
  - 1.2.1 Ziel der Arbeit . . . 3
  - 1.2.2 Anwendungsgebiete des Dienstes . . . 3
- 1.3 Struktur der Arbeit . . . 4

2 Charakteristika des Change Managements **5**
- 2.1 Definition des Change Managements . . . 5
- 2.2 Anforderungen an ein Change Management System . . . 6
- 2.3 Kriterienkatalog . . . 8

3 Klassifizierung existierender Werkzeuge und Dienste **10**
- 3.1 Untersuchte Medien . . . 10
- 3.2 Grad der Berücksichtigung der jeweiligen Dokumentenstruktur . . . 12
- 3.3 Differenziertheit in Bezug auf die Auswertung . . . 12
- 3.4 Art des Dienstes . . . 13

4 Existierende Algorithmen zum Change Management **15**
- 4.1 Methoden zur Repräsentation von Differenzen . . . 15
- 4.2 Vergleich von Sequenzen . . . 18
- 4.3 Vergleich von Bäumen . . . 20
  - 4.3.1 Vergleich geordneter Bäume . . . 21
  - 4.3.2 Vergleich ungeordneter Bäume . . . 25
- 4.4 Überblick der Verfahren . . . 35

5 Verwendete Entwicklungsumgebung und Sprachen **38**
- 5.1 JAVA . . . 38
- 5.2 XML . . . 39
- 5.3 UML . . . 39
- 5.4 Entwurfsmuster . . . 40

6 Entwicklung Generischer Dienste für das Change Management **42**
- 6.1 Ablauf des Verfahrens . . . 42
- 6.2 Modell und Annahmen des realisierten Ansatzes . . . 44
- 6.3 Das Operationen-Repertoire . . . 44
- 6.4 Repräsentation relevanter Information . . . 49
  - 6.4.1 Repräsentation der Änderungen . . . 49
  - 6.4.2 Repräsentation der Metadaten . . . 51
- 6.5 Verwendete Algorithmen . . . 52
  - 6.5.1 Erzeugung des induzierten Graphen und Bildung der Kantengewichte . . . 52
  - 6.5.2 Beschriftung der Kanten . . . 61
  - 6.5.3 Ermittlung des *edit-scripts* . . . 66

6.5.4 Betrachtungen zur Zeitabschätzung 73

7 Ausgewählte Aspekte der Realisierung 75
7.1 Grobarchitektur 75
7.2 Ereignisgesteuerte Verarbeitung von XML-Dokumenten 76
7.3 Repräsentation und Bearbeitung bipartiter Graphen 78
7.4 Austauschbare Algorithmen zum Baumvergleich 80
7.5 Generierung des *edit-scripts* 81
7.6 Hindernisse und Richtlinien 82
7.7 Visualisierung 83
7.8 Konfiguration des Dienstes 88

8 Zusammenfassung und Ausblick 90
8.1 Bewertung des entwickelten Systems 91
8.2 Ausblick 92
8.2.1 Weiterführende Arbeiten 92
8.2.2 Erweiterungen 92

Anhang 94

A Planarisierungsproblem in bipartiten Graphen 94

B DTD des *edit-scripts* und *edit-script*-Beispiele in XML 96
B.1 DTD des *edit-scripts* 96
B.2 Beispiele für *edit-scripts* in XML 97
B.2.1 *update*-Operation 97
B.2.2 *insert*-Operation 98
B.2.3 *delete*-Operation 98
B.2.4 *move*-Operation 98
B.2.5 *copy*-Operation 98
B.2.6 *glue*-Operation 99

C Die CD-Rom zur Arbeit 100

D Glossar 103

Literaturverzeichnis 108

Kapitel 1

Einleitung

Durch die immer schneller voranschreitende Entwicklung der Informationstechnologien im allgemeinen und die wachsende Verbreitung des Internet und des *World Wide Web* in den vergangenen Jahren sind zwei Tendenzen zu beobachten. Zum einen bieten die derzeit erreichten technischen Voraussetzungen die Möglichkeit des nahezu unbegrenzten Austauschs digitaler Informationen jeder Art. Zum anderen kann aber gerade diese Fülle erreichbarer Informationen eine Recherche sehr erschweren. Nicht mehr die *prinzipielle* Verfügbarkeit einer Informationsresource ist das Problem, sondern die *tatsächliche* Verfügbarkeit zur richtigen Zeit am richtigen Ort und im richtigen Format. Aus den genannten Gründen besteht eine hohe Nachfrage nach Systemen, welche die gezielte Informationsbeschaffung erleichtern. Die Aufgabe dieser Systeme ist es, dem Anwender die Informationsrecherche dahingehend zu vereinfachen, daß er aus der Fülle der vorhandenen Information diejenige erhält, die für ihn relevant ist. In dieser Situation kommt den Digitalen Bibliotheken eine wichtige Rolle als Vermittler zwischen Informationskonsumenten und Informationsanbietern zu [Bar98].

Das primäre Ziel einer Digitalen Bibliothek liegt in der Verbesserung der Ausbeutung vernetzter Informationen, wobei das Hauptaugenmerk auf die Erfüllung individueller Bedürfnisse der Informationskonsumenten und seiner Aufgaben gerichtet ist. Damit schließt eine Digitale Bibliothek die Lücke zwischen einer nahezu unendlich großen Menge von Informationen auf der einen und den speziellen Bedürfnissen ihrer Anwender und deren Aufgaben auf der anderen Seite [SSNM97].

Den wesentlichsten Teil einer Digitalen Bibliothek stellen Kollektionen meist verteilter Informationsressourcen dar, die auf die Bedürfnisse bestimmter Gruppen von Informationskonsumenten abgestimmt sind. Diese Kollektionen entsprechen damit in gewisser Weise klassischen Medien in traditionellen Bibliotheken, wie etwa Büchern oder Zeitschriften. Informationen zur Strukturierung, Beschreibung und Annotation der Kollektionen, die in klassischen Medien Katalogen, Annotationen und Kommentaren entsprechen, bilden als *Metadaten* einen weiteren Teil Digitaler Bibliotheken. Einen dritten Teil stellen schließlich Dienste, wie etwa solche zur Auswahl, Kombination, Strukturierung und Annotation dar, die als *Mehrwertdienste* die Handhabung der Inhalte der Digitalen Bibliothek erleichtern.

Durch die Spezialisierung für bestimmte Benutzergruppen und das Angebot von Mehrwertinformationen leisten Digitale Bibliotheken einen wichtigen Beitrag zur effektiven und effizienten Handhabung von Informationsressourcen.

Beispiele Digitaler Bibliotheken bilden etwa das System *MeDoc*, dessen Ziel der Informationsaustausch und die Literaturversorgung in der Wissenschaft darstellt [Bar98], die Digitale Bibliothek der UC-Berkeley [Ten96] und die *Warburg Electronic Library*, einem interdisziplinären Kooperationsprojekt des Kunstgeschichtlichen Seminars der Universität Hamburg und des Arbeitsbereichs Softwaresysteme der Technischen Universität Hamburg-Harburg, in dessen Rahmen der Bildindex

zur politischen Ikonographie in elektronischer Form nutzbar gemacht und um zusätzliche Dienste erweitert wird [Sch97, NHM+96].

Die von Digitalen Bibliotheken angebotenen Informationsressourcen sind nicht als statisch zu betrachten, sondern unterliegen Änderungen, von denen auch die Metadaten betroffen sind. Dieses legt es nahe, im Rahmen der von der Digitalen Bibliothek eingesetzten Dienste, ein *Change Management* zu betreiben, welches das Stattfinden von Veränderungen der Informationsressourcen erkennt und auf Grundlage erkannter Änderungen entsprechende Reaktionen auslöst.

1.1 Motivation

Digitale Bibliotheken bieten ihren Anwendern den Zugang zu einer großen Anzahl von Informationsressourcen, die nicht notwendigerweise dort lokalisiert sind, wo sich die Bibliothek befindet. Sie werden in der Regel sowohl von der Digitalen Bibliothek als auch jeweils voneinander getrennt gehalten. Die durch die Bibliothek zugänglich gemachten Informationsressourcen können beispielsweise Bücher, Artikel, Bilder, Videosequenzen, Programmbibliotheken oder andere Arten digitalisierbarer Information darstellen, deren Handhabung von oben erwähnten Diensten unterstützt wird. Die Metainformationen der von einer Digitalen Bibliothek zur Verfügung gestellten Informationen lassen sich als aus drei Teilen bestehend auffassen:

Die partiellen Informationen enthalten Informationen, die sich automatisch aus den Informationsressourcen generieren lassen. Dieses ist bei Artikeln beispielsweise ein Inhaltsverzeichnis, die Dokumentengröße oder der Dokumententyp.

Als Mehrwert-Informationen hingegen werden diejenigen Informationen bezeichnet, die sich auf die Informationsresource beziehen, sich aber nicht automatisch aus dieser ableiten lassen. Hierzu gehören - wieder am Beispiel eines Artikels - die Zusammenfassung oder persönliche Anmerkungen der Anwender und Klassifikationsinformationen, wie Schlagwörter.

Der Verweis auf die Quelle der Informationsresource sorgt dafür, daß die Informationsressourcen lokalisiert werden können. Dieses ist beispielsweise eine URL (*Uniform Resource Locator*).

Den Metainformationen kommt bei der Handhabung der Informationsressourcen eine wichtige Rolle zu. Insbesondere ist der Informationskonsument mit Hilfe solcher Informationen in der Lage zu entscheiden, ob er sich für ein angebotenes Dokument interessiert, ohne es vorher erworben zu haben.

Von Digitalen Bibliotheken angebotene Informationsressourcen unterliegen Veränderungen, über deren Art sich die Kunden der Bibliothek informieren können sollten. So werden für Kundenprogrammierer entwickelte Programmbibliotheken immer wieder erweitert und strukturell verändert. Ein sich für diese Informationsressourcen interessierender Kunde sollte in der Lage sein, sich über Art und Umfang dieser Änderungen zu informieren, um auf dieser Basis seine eigenen Informationen aktualisieren und/oder erneute Kaufentscheidungen abwägen zu können.

1.2 Aufgabenstellung

Ausgangspunkt dieser Arbeit ist das Change Management für Metadaten, das einen wichtigen Mehrwertdienst für Digitale Bibliotheken darstellt. Bei Änderungen der über die Digitale Bibliothek angebotenen Informationsressourcen kann es dazu kommen, daß die zu den Ressourcen verwalteten Metadaten nicht mehr aktuell sind und entsprechend modifiziert werden müssen. Ein

Dienst zum Change Management erkennt und präsentiert die aus der Modifikation der Informationsressourcen resultierenden Änderungen der Metadaten und unterstützt so deren konsistente Aktualisierung.

1.2.1 Ziel der Arbeit

Die Zielsetzung der Arbeit besteht in der Entwicklung eines generischen Dienstes zum Vergleich semistrukturierter Daten, der den Kern eines Dienstes zum Change Management bildet. Der schriftliche Teil der Arbeit enthält eine Beschreibung und Diskussion sowohl verschiedener Dienste, die dieses Problem in unterschiedlicher Ausprägung lösen, als auch verschiedener Algorithmen, die sich mit dem Vergleich semistrukturierter Daten unter verschiedenen Aspekten beschäftigen. Aus dieser Diskussion gehen Anforderungen und Ideen für den Algorithmus hervor, der für den zu entwickelnden Dienst verwendet wird.

Als konkretes Ergebnis entsteht ein Dienst, der zwei XML-Dokumente unter Berücksichtigung ihrer Struktur so miteinander vergleicht, daß als Resultat ein Änderungsprotokoll (ein sogenannter *edit-script*) entsteht. Dieses gibt Auskunft darüber, mit Hilfe welcher Folgen von Änderungsoperationen das eine Dokument aus dem anderen hervorgehen kann. Wichtige Teilprobleme sind dabei die Erkennung von Änderungen, deren adäquate Repräsentation und die übersichtliche Präsentation.

Aufgrund der vielfältigen Einsatzgebiete von XML und seiner wachsenden Bedeutung im Bereich der Daten- und Dokumentenbeschreibung und des Informationsaustauschs wird XML als Form für die Repräsentation der zu vergleichenden Dokumente und der Änderungen gewählt.

Zu zwei Dokumenten kann es unterschiedliche *edit-scripts* geben, mit deren Hilfe man das eine in das andere Dokument überführen kann. Ziel des Dienstes ist es, einen möglichst kompakten und zugleich aussagekräftigen *edit-script* zu generieren. Die Bezeichnung *aussagekräftig* bezieht sich in diesem Zusammenhang sowohl auf die Art der im *edit-script* verwendeten Operationen, als auch auf deren Reihenfolge. Beziehen sich mehrere Operationen auf das selbe oder dicht beieinander liegende Elemente, ist es sinnvoll, diese im *edit-script* dicht beieinander liegend vorzufinden. Neben den in den meisten bisher bekannten Diensten gebräuchlichen Operationen *Einfügen, Löschen* und *Ändern* werden auch Operationen wie *Kopieren* und *Verschieben* genutzt, wobei sich die Operationen jeweils auf Knoten einschließlich ihrer Nachfolgerknoten beziehen.

Besonderes Gewicht wird bei dem zu erstellenden Dienst auf einen generischen Ansatz und auf klare Schnittstellen gelegt, die den Einsatz des Dienstes in unterschiedlichen Kontexten (siehe 1.2.2) erleichtern.

1.2.2 Anwendungsgebiete des Dienstes

Aufgrund der Eignung und vielfältigen Verwendung von XML ergeben sich zahlreiche Anwendungen für den im Rahmen dieser Arbeit entwickelten Dienst. Zu den wichtigsten zählen:

- Erkennung zusammengehöriger Versionen: Von einer Digitalen Bibliothek aufzunehmende Informationsressourcen lassen sich mit Hilfe dieses Dienstes daraufhin untersuchen, ob es sich um andere Versionen bereits vorhandener Informationsressourcen handelt.
- Erkennung von Doubletten: der Dienst läßt sich dazu einsetzen, um von einer Digitalen Bibliothek aufzunehmende Informationsressourcen daraufhin zu überprüfen, ob sie bereits aufgenommenen so ähnlich sind, daß es sich vermutlich um Doubletten handelt.
- Verwendung für die Versionsverwaltung: Mit Hilfe des Dienstes ist es möglich, die Änderungen sowohl in auseinander hervorgegangenen als auch sich parallel entwickelnden Versionen einer Informationsresource nachvollziehbar zu machen.

- Der Dienst läßt sich als Grundlage für ein *merging tool* einsetzen: die Differenzen zweier Dokumente werden als Folge von Änderungsoperationen repräsentiert, deren einzelne Operationen selektiv angenommen oder zurückgewiesen werden können.

1.3 Struktur der Arbeit

Die vorliegende Arbeit ist wie folgt gegliedert: Nach dieser Einleitung werden im Kapitel 2 einige grundlegende Eigenschaften des Change Management und dessen im Rahmen dieser Arbeit betrachteten Umfelds genauer beschrieben. Die Definition wichtiger Anforderungen an ein Change Management System und des darauf aufbauenden Kriterienkatalogs bildet die Grundlage einer Klassifizierung existierender Dienste in Kapitel 3. Dabei wird eine breite Palette unterschiedlicher Dienste zum Erkennen und zur Präsentation von Änderungen betrachtet.

In Kapitel 4 werden ausgewählte Algorithmen zum Change Management untersucht. Nach einer abstrakten Sichtweise auf den Vergleich von Informationsbeständen werden drei Problemklassen identifiziert und in Verbindung mit einigen Verfahren zu deren Lösung vorgestellt.

Eine Beschreibung der für die Implementierung verwendeten Entwicklungsumgebung und Sprachen erfolgt in Kapitel 5.

Auf der Grundlage der in den Kapiteln 2 bis 4 erarbeiteten Voraussetzungen wird in Kapitel 6 ein eigenes Verfahren vorgestellt. Dieses realisiert ein strukturorientiertes Verfahren, um zwei beschriftete Bäume miteinander zu vergleichen und das Ergebnis dieses Vergleichs in Form eines aussagekräftigen *edit-scripts* zur Verfügung zu stellen.

Die Beschreibung einiger ausgewählter Aspekte des vorgestellten Verfahrens erfolgt in Kapitel 7. Es umfaßt neben den wichtigsten Aspekten der Implementierung die Darstellung der Visualisierungskomponenten des Dienstes, welche eine Nachvollziehbarkeit des Verfahrens ermöglicht.

Die vorliegende Arbeit schließt mit einer Zusammenfassung und dem Ausblick auf weiterführende Arbeiten, sowie auf mögliche Erweiterungen des Verfahrens und der prototypischen Implementierung in Kapitel 8.

Kapitel 2

Charakteristika des Change Managements

Bevor im Kapitel 3 eine Klassifizierung bestehender Dienste für das Change Management vorgenommen wird, beschäftigt sich dieses Kapitel mit grundlegenden Aspekten des Change Management und den Anforderungen an Dienste für das Change Management. Zunächst erfolgt im Abschnitt 2.1 eine Definition des Change Management und des hier betrachteten Umfelds. Die Anforderungen sowie ein darauf aufbauender Kriterienkatalog werden schließlich in den Abschnitten 2.2 und 2.3 dargestellt.

2.1 Definition des Change Managements

Unter dem Begriff *Change Management* versteht man jede Form der Handhabung von Veränderungen. Im Bereich der Betriebswirtschaft wird darunter die Veränderung von Geschäftsprozessen in Unternehmen verstanden. Solche Änderungen betreffen meist weite Teile des Unternehmens, wobei organisatorische Fragestellungen im Vordergrund stehen. Eng damit verbunden ist in vielen Change Management - Projekten die Einführung neuer IT-Komponenten. In nahezu allen Bereichen der Wirtschaftswissenschaften spielt das Change Management eine wichtige Rolle [Ber98].

Diese Arbeit beschäftigt sich mit dem Change Management im Zusammenhang mit der Handhabung von Änderungen in Informationsbeständen. Dieses umfaßt das Erkennen und eine eventuelle Reaktion auf das Stattfinden von Änderungen. Von besonderem Interesse ist für diese Arbeit das Aufbereiten von Änderungen in Informationsbeständen. Dabei sind *Informationsbestände* Ausprägungen unterschiedlicher *Informationsarten*, die sich sowohl in ihrer Struktur (Syntax), als auch in ihrer Bedeutung (Semantik) unterscheiden können. So lassen sich beispielsweise Sequenzen von Ausprägungen bestimmter Elementarten als eine Informationsart bezeichnen, von der sich Sequenzen von Ausprägungen anderer Elementarten unterscheiden. Wieder andere Informationsarten würden durch Mengen oder Bäume von Elementen gebildet. Zwei Informationsarten mit gleicher Syntax können als unterschiedlich erachtet werden, wenn die ihnen zugrundeliegenden Bedeutungen voneinander abweichen. Beispielsweise könnten sowohl Folgen von Namen als auch Folgen von Telefonnummern als Folgen von Zeichenketten organisiert sein, demnach die gleiche Struktur aufweisen. Dennoch sind diese Informationsarten unterschiedlich, da sie voneinander abweichende Bedeutungen haben. Beim Change Management müssen sowohl Änderungen in der konkreten Instanz als auch in der Informationsart berücksichtigt werden. Ein Vergleich konkreter Instanzen zur Ermittlung von Differenzen setzt voraus, daß die zugrundeliegenden Informationsarten hinreichend ähnlich sind.

Für ein Interesse für die Unterschiede zwischen zwei Informationsbeständen gibt es drei Szenarien (Abbildung 2.1):

a) Ein Informationsbestand A_0 wird im Laufe der Zeit modifiziert, so daß ein Informationsbestand A_n entsteht. Ein Vergleich dieser beiden Versionen soll Aufschluß über einzelne vorgenommene Modifikationen geben.

b) Ein Informationsbestand A_0 kann gewissermaßen gleichzeitig in unterschiedlicher Weise modifiziert werden, so daß zusätzlich die Informationsbestände A_m und A_n existieren. Ein Vergleich dieser beiden letztgenannten Versionen zeigt Differenzen, die unter anderem als Grundlage für ein Zusammenführen (*merging*) der Bestände verwendet werden können.

c) Zwei Informationsbestände A und B können miteinander verglichen werden, ohne die Annahme, daß sie aus einem gemeinsamen Informationsbestand hervorgegangen sind. Auch hier kann das Ziel wieder ein reiner Vergleich der Bestandteile oder auch ihr Zusammenführen zu einem Bestand sein.

Allen drei Szenarien gemeinsam ist die Tatsache, daß Differenzen zwischen Informationsbeständen aufgezeigt werden, ohne Kenntnis der Historie haben zu müssen. Das Fehlen dieser Historie kann mehrere Gründe haben: es fehlten die technischen Voraussetzungen zu ihrer Erzeugung, die für die Modifikationen der Informationsbestände verantwortlichen Personen hatten kein Interesse, die Historie zu veröffentlichen oder die Informationsbestände sind vollständig unabhängig voneinander entstanden.

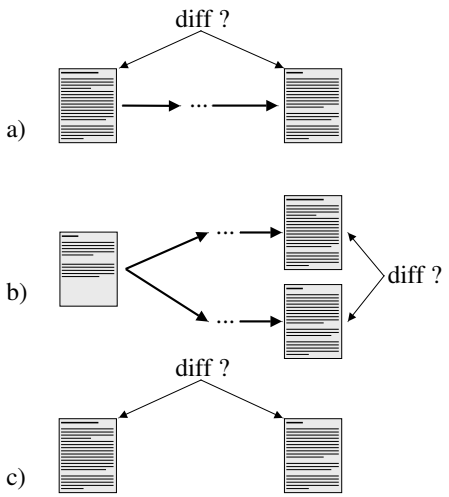

Abbildung 2.1: Drei Szenarien für die Existenz unterschiedlicher Informationsbestände

2.2 Anforderungen an ein Change Management System

Nach der in Abschnitt 2.1 gegebenen Definition des Change Management, werden in diesem Abschnitt die Anforderungen an einen Dienst für das Change Management angegeben. Aufbauend auf diesen Anforderungen ergibt sich ein entsprechender Kriterienkatalog.

Die Handhabung von Änderungen in Informationsbeständen umfaßt das Erkennen von Änderungen, deren adäquate Präsentation und einen Prozeß für den Umgang mit den Änderungen.

Urheber der Informationsressourcen (die Erzeuger von Informationsressourcen und diejenigen, die Informationsressourcen verändern, werden hier als *Urheber* bezeichnet) und Informationskonsumenten sind in der Regel unterschiedliche Personen, beziehungsweise Organisationen. Somit besteht eine räumliche und organisatorische Lücke zwischen beiden, die von einem Dienst für das Change Management überwunden werden muß, wenn der Informationskonsument ein Interesse an den Informationsressourcen und deren Veränderungen hat.

Das Erkennen von Änderungen kann sich auf das Erkennen der Tatsache, *daß* Informationsbestände Unterschiede aufweisen, beschränken oder aber darüber hinausgehend genauere Analysen darüber durchführen, welche Unterschiede im einzelnen bestehen. Dabei wird dem Grad, in dem auf die Struktur der zu analysierenden Informationsbestände eingegangen wird, eine große Bedeutung beigemessen. Sowohl diese Berücksichtigung der Strukturen der miteinander zu vergleichenden Informationsbestände, als auch die Repräsentation der erkannten Unterschiede beeinflussen die Aussagekraft der Resultate erheblich.

Das Änderungsverhalten von Informationsbeständen ist sehr unterschiedlich. Es variiert hinsichtlich der zeitlichen Häufigkeit, dem Umfang und der Art stattgefundener Änderungen. Diesen Unterschieden auf der einen Seite und den Bedürfnissen der Anwender des Change Management Systems auf der anderen Seite ist daher Rechnung zu tragen.

Ebenfalls zu berücksichtigen ist die Art und der Umfang der zur Verfügung gestellten Informationen über die Änderungen, wie beispielsweise ausführliche Protokolle. Diese können eine genaue Analyse der Informationsbestände überflüssig machen.

Für die Kenntnisnahme des Informationskonsumenten vom Stattfinden der Änderungen lassen sich drei zu berücksichtigende Mechanismen unterscheiden:

- Bei einer *Subskription* werden die Informationskonsumenten von dem Urheber benachrichtigt, sofern sie ihr generelles Interesse an den Informationsressourcen bei dem Urheber (durch eine *Einschreibung*) bekundet haben. Der Urheber verwaltet eine Kollektion mit den eingeschriebenen interessierten Informationskonsumenten und benachrichtigt diese nach Durchführung von Veränderungen. Ein solcher Dienst wird verfeinert durch die Berücksichtigung individueller Interessen und Präferenzen der Informationskonsumenten. Dieser Mechanismus setzt voraus, daß der Urheber die Bereitschaft und Möglichkeit besitzt, eingeschriebene Informationskonsumenten zu benachrichtigen.

- Durch eigenständiges Überprüfen der Informationsressourcen durch den Informationskonsumenten ist dieser in der Lage, Veränderungen an diesen zu erkennen. Beispielsweise mit Hilfe sogenannter *Robots* [Kos94a, Kos95] werden die Informationsbestände des Urhebers besucht und auf Änderungen hin überprüft. Dieser Mechanismus setzt die Erreichbarkeit der Informationsbestände für den Informationskonsumenten und damit eine gewisse Bereitschaft des Urhebers voraus [Kos94b]. Darüber hinaus birgt dieser Mechanismus den Nachteil, daß das Überprüfen der Informationsbestände auf Veränderungen mehrfach (nämlich von jedem interessierten Informationskonsumenten) vorgenommen wird, wodurch es zu starken Belastungen der Systeme kommen kann, die die Informationsressourcen halten [HSW96]. Bei diesem Mechanismus sind die individuellen Bedürfnisse der Informationskonsumenten von diesen selbst zu berücksichtigen.

- Der dritte Mechanismus involviert eine dritte Partei, die die Kooperation zwischen Urhebern und Informationskonsumenten vornimmt. Sie verwaltet eine Kollektion der interessierten Informationskonsumenten und übernimmt deren möglicherweise individuelle Benachrichtigung bei erkannten Veränderungen der Informationsressourcen. Dadurch wird der Urheber davon entlastet, eine möglicherweise sehr große Anzahl von Informationskonsumenten zu verwalten und bei Änderungen zu benachrichtigen. Zum anderen braucht der Informationskonsument

nicht beständig die Informationsbestände auf Änderungen zu überprüfen. Diese dritte Partei übernimmt die Berücksichtigung individueller Bedürfnisse der Informationskonsumenten gegebenenfalls durch den Einsatz spezieller Werkzeuge und Dienste.

Zumindest einer dieser drei Mechanismen muß von einem System für das Change Management zur Verfügung gestellt werden, damit der Informationskonsument von Veränderungen der Informationsressourcen erfährt.

Um Veränderungen - oder allgemeiner Unterschiede - in Informationsbeständen erkennen und gegebenenfalls untersuchen zu können, muß die Möglichkeit bestehen, in hinreichendem Maße auf diese Informationsbestände zugreifen zu können. Dieses betrifft sowohl die Überbrückung möglicherweise beträchtlicher Entfernungen zu den Informationsbeständen, den Besitz notwendiger Zugriffsberechtigungen, sowie die Kenntnis der Formate der zu untersuchenden Informationsbestände. Sich bei letzteren auf einzelne, nur für spezielle Anwendungen entwickelte Formate zu beschränken, ist sicherlich ebenso ungeeignet wie die Verwendung von Formaten, die zwar universell einsetzbar, weitgehend bekannt sind, aber eine Berücksichtigung der Struktur der Informationsbestände nicht im geforderten Maße ermöglichen.

2.3 Kriterienkatalog

Aufbauend auf den in 2.2 beschriebenen Anforderungen an ein Change Management System lassen sich eine Reihe von Kriterien zur Klassifizierung und Bewertung von Diensten für das Change Management angeben. Anhand einer Auswahl dieser Kriterien wird die in Kapitel 3 beschriebene Klassifizierung bestehender Dienste vorgenommen.

Generizität: Dieses Kriterium beschränkt sich nicht nur auf Dienste für das Change Management, sondern ist generell eine erstrebenswerte Eigenschaft für IT-Komponenten. Die *Generizität* ist ein Maß für die allgemeine Verwendbarkeit für ähnliche Aufgaben in unterschiedlichen Umgebungen. Durch die Wahl schmaler und allgemeingehaltener Schnittstellen, Parametrisierbarkeit und Konfigurierbarkeit erhöht sich die Generizität eines Dienstes, der sich dadurch oft für mehr als nur für die zur Zeit ihrer Entwicklung vorgesehenen Zwecke einsetzen läßt, also einen hohen Grad der Wiederverwendbarkeit aufweist [PB93]. Ein generischer Dienst für das Change Management ist zum Beispiel nicht auf spezielle Informationsressourcen mit deren Formaten und Benutzerschnittstellen festgelegt. Dies kann etwa durch die Verwendung weit verbreiteter Formate für den Austausch von Informationen gewährleistet werden.

Variable Strategien: Dienste für das Change Management bestehen aus unterschiedlichen Komponenten mit jeweils eigenen Aufgabenbereichen, wie etwa das Erkennen von Änderungen oder die Benachrichtigung einer Gruppe von Informationskonsumenten. In der Regel gibt es für diese Aufgabenbereiche unterschiedliche Strategien für ihre Lösung. Bei Betrachtung der oben angesprochenen Mechanismen der Kenntnisnahme des Informationskonsumenten vom Stattfinden der Änderungen gibt es beispielsweise für die beiden letztgenannten Mechanismen Strategien, die individuelle Bedürfnisse der Informationskonsumenten berücksichtigen, oder die Informationsbedürfnisse der Konsumenten einheitlich behandeln. Ebenso sind unterschiedliche Strategien für das Erkennen von Differenzen einsetzbar. Für die vielseitige Einsetzbarkeit des Dienstes ist es wichtig, für Aufgaben unterschiedliche Strategien anzubieten. Dieses Kriterium betrifft, wie das oben genannte der Generizität, nicht nur Dienste für das Change Management.

Performance: Die *Performance* eines Dienstes beschreibt das Verhältnis seiner *Wirksamkeit* (die Güte der von ihm erzeugten Lösungen in Bezug auf die Aufgabenstellung) zu der Menge der von ihm verbrauchten Ressourcen. Unter *Ressourcen* fallen hierbei beispielsweise die eingesetzte Rechenleistung, Rechenzeit sowie der benötigte Speicherplatz. Bei dem Entwurf

und der Entwicklung von IT-Komponenten gilt es, eine hohe Wirksamkeit bei einem möglichst geringen Ressourcenverbrauch zu erzielen. Diese beiden Teilkriterien stehen einander häufig entgegen. So ist die Lösung der Probleme einer mit *NP-vollständig* bezeichneten Problemklasse (wie etwa das sogenannte *Erfüllbarkeitsproblem* [HU79]) nicht mit polynomiellen Zeitaufwand lösbar, so daß man sich in derartigen Fällen häufig sogenannter *Heuristiken* bedient, die näherungsweise Lösungen der Probleme bei akzeptablem Einsatz von Ressourcen ermöglichen. Gute Systeme zeichnen sich häufig durch einen geeigneten Kompromiß zwischen Wirksamkeit und Ressourcenverbrauch aus. Auch dieses Kriterium der *Performance* betrifft nicht nur Dienste des Change Management.

Interaktion mit der Umgebung: Neben dem reinen Erkennen von Änderungen umfaßt das Aufgabengebiet eines Systems für das Change Management auch die Interaktion mit seiner Umgebung. Zum einen muß das System in der Lage sein, auf Informationsbestände zuzugreifen und sie auf möglicherweise stattgefundene Änderungen hin zu untersuchen. Zum anderen sind die Informationskonsumenten über die vom System erkannten Änderungen (oder eventuell auch über das Ausbleiben von Änderungen nach Ablauf einer gegebenen Zeitspanne) etwa über eines der oben genannten Mechanismen zu informieren. Dieses Kriterium, das speziell für Systeme für das Change Management ist, beschreibt das Ausmaß, in dem das System in Interaktion mit seiner Umgebung steht.

Berücksichtigung der Dokumentenstruktur: Den miteinander zu vergleichenden Informationsbeständen liegt jeweils eine bestimmte Struktur zugrunde, die bei dem Vergleich in unterschiedlichem Maße berücksichtigt wird. Hierarchische Strukturen beispielsweise lassen sich zwar sequentiell durchlaufen. Ein Vergleich hierarchisch strukturierter Informationsbestände unter Vernachlässigung dieser Struktur führt jedoch zu unschärferen Ergebnissen, da die hierarchische Einordnung verlorengeht.

Betrachtet der Dienst die Struktur der Dokumente sehr genau, lassen sich Strukturbestandteile in Bezug auf die Relevanz bei Veränderungen individuell unterschiedlich betrachten. So ist es beispielsweise möglich, daß ein Dienst Veränderungen bestimmter Dokumentenbestandteile in Abhängigkeit ihrer Struktur und ihrer Stellung in der Gesamtstruktur unterschiedlich behandelt. Somit stellt der Grad der Berücksichtigung der Dokumentenstruktur ein wichtiges Kriterium für Systeme zum Change Management dar.

Differenziertheit der Vergleichsergebnisse: Der Grad der Differenziertheit der Ergebnisse eines Vergleichs von Informationsbeständen stellt ein weiteres Kriterium dar. Ein Vergleich, dessen Ergebnis alleine lautet, *daß* Änderungen stattgefunden haben, ist nicht so aussagekräftig, wie ein Ergebnis, das genaue Angaben darüber macht, welche Dokumententeile eingefügt, entfernt, geändert, verschoben oder kopiert wurden. Das Ziel eines Dienstes für das Change Management sollte es sein, die erkannten Differenzen so aussagekräftig zu repräsentieren, wie dies für eine sinnvolle Reaktion auf die Änderungen erforderlich ist. Das Melden der Tatsache, *daß* Änderungen stattgefunden haben, reicht dafür in der Regel nicht aus.

Kompaktheit der Repräsentation erkannter Differenzen: Das letzte hier behandelte Kriterium bezieht sich auf die Kompaktheit erkannter Änderungen. Es steht in enger Beziehung zu dem der Differenziertheit. Wie später gezeigt wird, lassen sich die Unterschiede zwischen Informationsbeständen auf unterschiedliche Weise beschreiben. Bei der Verwendung sogenannter *edit-scripts*, die die Differenzen in Form von Sequenzen von Änderungsoperationen darstellen, gibt es mehrere *edit-scripts* zur Beschreibung der Differenz zwischen zwei Informationsbeständen. Sie unterscheiden sich in ihrer Länge und in ihrer Zusammensetzung voneinander. Sehr differenzierte, aber sehr umfangreiche Ergebnisse, die beispielsweise das Verändern nahezu aller Worte eines Satzes in einem Text repräsentieren, sind zwar sehr aussagekräftig, jedoch für einen menschlichen Nutzer des Systems nicht so eingängig, wie das Ergebnis eines Systems, das das Entfernen des alten und Einfügen des neuen Satzes meldet [NCK+92]. Es muß das Ziel eines Dienstes für das Change Management sein, möglichst kompakte Ergebnisse zu liefern, die zudem keine überflüssigen Operationen enthalten.

Kapitel 3

Klassifizierung existierender Werkzeuge und Dienste

In Kapitel 2 wurden Kriterien zur Bewertung eines Change Management Systems für Informationsressourcen erarbeitet. Bevor in Kapitel 4 näher auf konkrete Algorithmen zum Erkennen von Differenzen eingegangen wird, soll an dieser Stelle eine Klassifizierung bestehender Dienste in Zusammenhang mit dem Erkennen von Änderungen in Informationsressourcen vorgenommen werden. Diese Klassifizierung erfolgt nach einer Auswahl der am Ende des vorigen Kapitels aufgeführten Kriterien:

Untersuchte Medien: Unter Berücksichtigung des Kriteriums der Generizität läßt sich eine Einteilung der Dienste in Bezug auf die von ihnen untersuchten Medien vornehmen.

Grad der Berücksichtigung der jeweiligen Dokumentenstruktur: In Abhängigkeit der Berücksichtigung der Struktur der untersuchten Dokumente lassen sich die verschiedenen Dienste in unterschiedliche Gruppen einteilen.

Differenziertheit in Bezug auf die Auswertung: Nach dem Grad der Differenziertheit der Vergleichsergebnisse werden die Dienste in unterschiedliche Gruppen eingeteilt. Diese Einteilung bezieht sich auf das am Ende des vorigen Kapitel beschriebene Kriterium der *Differenziertheit der Vergleichsergebnisse*.

Art des Dienstes: Schließlich wird eine Einteilung der miteinander verglichenen Dienste nach der Art des Dienstes vorgenommen.

Im Folgenden werden die genannten Kriterien genauer beschrieben.

3.1 Untersuchte Medien

Change Management Systeme lassen sich unter anderem danach klassifizieren, welcher Art die von ihnen untersuchten Informationsbestände sind. So lassen sich Dienste zum Vergleich von Text-Dokumenten von solchen unterscheiden, die HTML- oder XML-Dokumente miteinander vergleichen:

Vergleich von Text-Dokumenten: Dienste dieser Gruppe arbeiten im allgemeinen zeilenorientiert. Ein Repräsentant dieser Gruppe ist der aus *UNIX*-Umgebungen bekannte Dienst *gnuDiff* [GNU84]. Es verwendet ein in [Mye86] beschriebenes Verfahren, welches die Differenz zweier Sequenzen mit Hilfe sogenannter längster gemeinsamer Untersequenzen (*longest common subsequences* - LCS) ermittelt. Im Abschnitt 4.2 erfolgt eine etwas ausführlichere Beschreibung dieses Themas.

Vergleich von LaTeX-Dokumenten: Dem Vergleich von LaTeX-Dokumenten dient das in [CRGMW96] beschriebene Verfahren. Der Dienst erhält zwei Versionen eines LaTeX-Dokuments als Eingabe und erzeugt als Ausgabe eine Version, die die Änderungen speziell auszeichnet. Eine nähere Beschreibung des diesem Dienst zugrundeliegenden Algorithmus findet sich in Abschnitt 4.3.1.

Vergleich von HTML-Dokumenten: Dieser Gruppe fallen die meisten betrachteten Dienste zu, was sich unter anderem mit der starken Verbreitung von HTML begründen läßt. Stellvertretend seien hier einige konkrete Dienste erwähnt:

- Der Dienst *Webguide* [DBCK96] dient dazu, den Anwendern Änderungen von HTML-Seiten so darzulegen, daß die aufgefundenen Differenzen in einer neuen HTML-Seite zusammengestellt werden. Dabei lassen sich zusätzlich anwenderspezifische Kriterien verwenden, wie beispielsweise Angaben über den Zeitpunkt zu dem der Anwender die ihn interessierenden HTML-Seiten zum letzten Mal gesehen hat, die Frequenz, mit der der Dienst die HTML-Seiten auf Änderungen hin untersuchen soll oder der Mechanismus der Benachrichtigung des Anwenders, wenn Veränderungen erkannt wurden.

- Das System *NetAttaché Pro* [Dev] dient (unter anderem) dazu, Veränderungen in HTML-Seiten zu erkennen. Im Kern ist *NetAttaché Pro* ein sogenannter *Web Spider*, der Web-Seiten, inklusive der durch Verweise *links* von diesen Seiten aus erreichbaren Seiten (bis zu einer bestimmten Verweistiefe), lädt und lokal speichert, um sie zu einem späteren Zeitpunkt von dort aus nutzen zu können (um beispielsweise Verbindungskosten zu sparen). Darüberhinaus werden sogenannte Filter angeboten, die zum Beispiel dazu dienen, Änderungen in vormalig besuchten Web-Seiten hervorzuheben.

- Der Dienst *Mind it* [Tec] benachrichtigt Anwender bei Änderungen von HTML-Seiten durch Versenden einer *e-mail*. Dieser Mechanismus wird *Subskription* genannt: Informationskonsumenten tragen sich bei dem Urheber der Seiten oder wie in diesem Fall bei einer dritten Partei, die die Kooperation zwischen Urhebern und Informationskonsumenten übernimmt ein und werden dort als *interessierte Informationskonsumenten* verwaltet. Bei *Mind it* lassen sich einzelne HTML-Seiten, Teile dieser Seiten, Bilder, Verweise und bestimmte Schlüsselworte beobachten.

- Der Dienst *WebPluck* [Hil98] durchsucht eine vom Anwender angegebene Liste von *Uniform Resource Locators (URL)* und erstellt aus den Inhalten dieser Seiten ein neues HTML-Dokument gewissermaßen als persönliche Zeitung, deren einzelne Artikel von verschiedenen *URLs* stammen. Dabei werden jeweils nur die aktuellen Inhalte berücksichtigt. Änderungen oder Unterschiede zu vergangenen Versionen werden nicht hervorgehoben.

Vergleich von XML-Dokumenten: Das *XML Diff and Merge Tool* von IBM-AlphaWorks [IBM99] ist ein vollständig in *JAVA* entwickelter Dienst, der es dem Anwender ermöglicht, Unterschiede zwischen zwei XML-Dokumenten zu erkennen und aus diesen beiden ein drittes Dokument zu erzeugen. Nach Angabe eines Basis- und eines Zieldokuments hebt der Dienst jede Änderung im Zieldokument bezüglich des Basisdokuments graphisch mit Hilfe von Symbolen und Farben hervor und ermöglicht es dem Anwender, die Veränderungen zu inspizieren und für jede einzelne zu entscheiden, ob sie übernommen werden soll. Dieser Dienst bedient sich der ebenfalls von IBM-AlphaWorks entwickelten mit *XMLTreeDiff* [IBM98b] bezeichneten *JAVA beans*. Diese Klassen dienen dazu, Unterschiede in XML-Dokumenten zu ermitteln, wobei das Verändern, Einfügen und Entfernen von XML-Elementen erkannt wird (insbesondere also nicht das Verschieben und Kopieren von Knoten). Bei dem Vergleich der XML-Dokumente werden diese als beschriftete Bäume betrachtet (wie einige der in Abschnitt

4.3 beschriebenen Verfahren). Ferner werden eindeutige Identifizierungen der Knoten vorausgesetzt (durch Verwendung von Attributen des Typs `ID` in den XML-Dokumenten, bzw. durch Betrachtung des Inhalts des Elements (Name und Attribute) als Identifikation).

3.2 Grad der Berücksichtigung der jeweiligen Dokumentenstruktur

Dienste, die Informationsbestände miteinander vergleichen sollen, berücksichtigen die Struktur der Informationsbestände in unterschiedlichem Grade. Einer Klassifizierung nach diesem Kriterium liegt der Gedanke zugrunde, daß sich Informationsbestände einer Art aus Elementen bestimmter Arten in bestimmter Weise zusammensetzen. So werden die Elemente, aus denen sich ein Text-Dokument zusammensetzt, durch Zeilen, also Zeichenketten repräsentiert, die sich wieder aus einzelnen Worten zusammensetzen. Die Elemente hierarchischer Strukturen werden durch Knoten in Bäumen repräsentiert. Im Folgenden werden vier Grade der Berücksichtigung der Dokumentenstruktur betrachtet:

Zeichenorientiert: Zeichenorientiert arbeitende Dienste betrachten die Informationsbestände als Folgen von Zeichen und vergleichen, wie beispielsweise das zum Betriebssystem *MS-DOS* [MS0] gehörende Programm `comp.exe`, Dokumente zeichenweise. Sie sind damit in der Lage alle erdenklichen Dokumente zu vergleichen, da jedes Dokument letztlich als Folge von Zeichen in einer Datei speicherbar ist. Weitergehende Strukturinformation wird ignoriert.

Zeilenorientiert: Dienste, wie das Programm *gnuDiff* [GNU84], setzen bei dem Vergleich zweier Dokumente voraus, daß es sich um Text-Dokumente handelt, die sich als Folgen von Zeichenketten beschreiben lassen. Die Dokumente werden zeilenweise verglichen. Damit sind diese Dienste nicht so universell einsetzbar wie zeichenorientierte, ermitteln jedoch für Text-Dokumente, wie z.B. Programmcode, aussagekräftigere Ergebnisse.

Baumorientiert: Dienste, wie die in [CRGMW96] und [CGM97] beschriebenen, sowie der Dienst *XML Diff and Merge Tool* betrachten die zu vergleichenden Informationsbestände als Bäume, die sich aus Baum-Knoten zusammensetzen. Ebenso läßt sich der Dienst *WebPluck* [Hil98] dieser Gruppe zuteilen, da hier, ähnlich wie bei dem *XML Diff and Merge Tool*, keine Verweise zu weiteren Dokumenten betrachtet werden.

Graphenorientiert: Prinzipiell sind all jene Dienste, die HTML-Dokumente mit ihren Verweisen zu weiteren Dokumenten bis zu einer vorzugebenen Tiefe miteinander vergleichen, als graphenorientiert zu betrachten, da sich derartige Strukturen nicht durch Bäume beschreiben lassen[1]. In diese Gruppe fallen demnach *Mind it* [Tec], *NetAttaché Pro* [Dev] und der Dienst *Webguide* [DBCK96].

3.3 Differenziertheit in Bezug auf die Auswertung

Der Grad der Differenziertheit der ermittelten Änderungen stellt ein weiteres Kriterium für eine Klassifizierung von Change Management Systemen dar.

Während einige Systeme nur erkennen, *daß* Änderungen stattgefunden haben, sind andere in der Lage, stattgefundene Änderungen nachzuvollziehen. Dabei sind die nachvollzogenen Änderungen unterschiedlich differenziert. In Kapitel 4 wird gezeigt, daß Unterschiede in Informationsbeständen häufig durch eine Folge elementarer Operationen dargestellt werden, mit denen aus dem einen Informationsbestand der andere erzeugt werden kann. Diese

[1] Bäume setzen eine azyklische Struktur voraus, wohingegen mehrere HTML-Dokumente zyklisch strukturiert sein können: zwei Dokumente können Verweise auf das jeweils andere Dokument enthalten.

Operationen werden auf einen Informationsbestand angewendet und beziehen sich jeweils auf als Elemente bezeichnete Bestandteile der Informationsbestände. Betrachtet man beispielsweise ein Text-Dokument als Informationsbestand, so kann sich so eine Operation auf einzelne Zeichen, Worte, Zeilen, Absätze oder ähnlich beziehen, je nachdem, wie fein die Granularität der Betrachtung ist [NCK+92]. Wenn Operationsfolgen (sog. *edit-scripts*) verwendet werden, lassen sich die Dienste zum Change Management nach dem Umfang ihres Operationen-Repertoires unterscheiden. Ein *Operationen-Repertoire* ist die Menge von Änderungsoperationen, aus denen die durch einen *edit-script* beschriebene Folge von Änderungsoperationen bestehen kann. Bei dem in dieser Arbeit beschriebenen Dienst stehen beispielsweise die Operationen *insert, delete, update, move, copy* und *glue* zur Verfügung. Während die Funktionsweise der ersten fünf Operationen intuitiv klar ist, sei zur *glue*-Operation an dieser Stelle nur so viel gesagt, daß es sich hierbei gewissermaßen um das Gegenstück der *copy*-Operation handelt: die Ausführung einer *glue*-Operation entfernt eines von zwei einander sehr ähnlichen Elementen, so daß man auch von einem *Einschmelzen* sprechen kann. Genauere Beschreibungen dieser Operationen finden sich in 4.3.2 und 6.3. Denkbar sind sicherlich noch weitere Operationen, die möglicherweise auch auf die zu untersuchenden Informationsarten abgestimmt sein können.

Mit dem Ziel, eine Klassifizierung vorzunehmen, ist eine Einteilung dieser Operationen in Gruppen sinnvoll. Dabei werden drei Gruppen gebildet, deren erste die Grundoperationen *update, insert* und *delete* enthält, die zweite zusätzlich die *move*- und die dritte zusätzlich die *copy*- und *glue*-Operation. Darauf aufbauend lassen sich existierende Dienste zum Erkennen von Unterschieden in Informationsbeständen in vier Gruppen einteilen:

Dienste, die nur erkennen, *daß* sich etwas geändert hat:

In diese Gruppe fällt der oben erwähnte Dienst *Mind it* [Tec]. Der Dienst *WebPluck* [Hil98] findet sich nicht in dieser Gruppe, da er keine Änderungen erkennt, sondern lediglich die jeweils aktuellen Inhalte zu einer neuen Seite zusammenstellt.

Dienste, die das Ändern, Einfügen und Entfernen von Elementen erkennen:

Zu dieser Gruppe gehören die oben erwähnten Dienste *gnuDiff* [GNU84], *Webguide* [DBCK96] und *NetAttaché Pro* [Dev]. Sie weisen den Anwender auf geänderte, entfernte und eingefügte Elemente hin, indem sie in einem der beiden Dokumente entsprechend hervorgehoben werden. Der Dienst *XML Diff and Merge Tool* von IBM-ALPHAWORKS [IBM99] nimmt in dieser Gruppe eine gesonderte Rolle ein, da er das interaktive Zusammenführen der beiden Dokumente ermöglicht.

Dienste, die zusätzlich das Verschieben von Elementen erkennen:

Der in [CRGMW96] beschriebene Dienst ist in der Lage, neben dem Ändern, Einfügen und Entfernen von Elementen auch das Verschieben von Bereichen zu erkennen. Da das Verfahren die zu vergleichenden Informationsbestände als Bäume auffaßt, bezieht sich das Verschieben von Bereichen auf Teilbäume.

Dienste, die zusätzlich das Kopieren und Einschmelzen von Elementen erkennen:

Ergänzend zu dem Erkennen stattgefundener Verschiebungen erkennt das in [CGM97] beschriebene Verfahren auch das Kopieren und Einschmelzen von Bereichen. Auch hier werden die zu vergleichenden Informationsbestände als Bäume betrachtet, so daß sich die Bereiche auf Teilbäume beziehen.

3.4 Art des Dienstes

In Abhängigkeit der Stellung des Dienstes in dem umgebenden System lassen sich folgende drei Gruppen bilden:

stand-alone: In diese Gruppe fallen Dienste, wie beispielsweise das System *NetAttaché Pro* [Dev] oder *gnuDiff* [GNU84], die einen Beitrag zum Change Management liefern, ohne Bestandteil eines größeren Systems zu sein.

Teil eines größeren Systems: Dienste, wie die Komponente *AIDE* des Systems *Webguide* [DBCK96], die Veränderungen von HTML-Seiten registrieren soll.

Forschungsprototyp: Systeme, wie die in [CGM97] und [CRGMW96] beschriebenen, sind Bestandteile eines größeren Forschungsprojektes (*The C3 Project* der *Stanford Database Group* [Cha97]).

In die Gruppe der Dienste, die Bestandteil eines größeren Systemes sind, fallen solche, die als Mehrwertdienst im Rahmen Digitaler Bibliotheken eingesetzt werden. Die Nutzung derartiger Dienste kommerziell angebotener Digitaler Bibliotheken kann kostenpflichtig sein.

Kapitel 4

Existierende Algorithmen zum Change Management

In Kapitel 3 wird eine Klassifizierung von Werkzeugen und Diensten zum Change Management vorgenommen. Diese bedienen sich unterschiedlicher Algorithmen, um Änderungen zwischen Informationsbeständen zu analysieren. Einige dieser Algorithmen sollen hier genauer beschrieben werden. Sie werden teilweise von dem in Kapitel 6 beschriebenen Verfahren verwendet.

Die in diesem Abschnitt untersuchten Algorithmen unterscheiden sich in dem Umfang, in dem sie auf die Strukturen der jeweiligen zu vergleichenden Informationsbestände eingehen. Im Folgenden werden grob zwei Gruppen unterschieden:

Sequenzorientierte Algorithmen: Solche, die die zu vergleichenden Informationsbestände als Sequenzen betrachten. Derartige Strukturen werden auch als *flach* bezeichnet [CGM97]. Hierunter fallen Algorithmen, die im weitesten Sinne Listen von Elementen (beispielsweise Zeichenketten) miteinander vergleichen.

Baumorientierte Algorithmen: Algorithmen, deren zu vergleichenden Informationsbestände als hierarchisch strukturiert betrachtet werden, vergleichen jeweils zwei Bäume miteinander. Hierbei ist ferner zu unterscheiden, ob es sich um geordnete oder ungeordnete Bäume handelt, ob also die Reihenfolge der sich auf einer Ebene befindenden Elemente für den Vergleich von Bedeutung ist oder nicht.

Im ersten Teil dieses Kapitels werden Repräsentationsformen für Differenzen von Informationsbeständen behandelt. Der zweite und dritte Teil beschäftigen sich mit unterschiedlichen Verfahren zum Erkennen von Unterschieden. Im zweiten Teil werden oben angesprochene sequenzorientierte und im dritten Teil baumorientierte Algorithmen beschrieben. Ein Überblick über die besprochenen Verfahren zu diesem Thema wird schließlich im vierten Teil dieses Kapitels gegeben. Ferner wird dort die in dieser Arbeit erfolgte Entwicklung eines eigenen Verfahrens motiviert.

4.1 Methoden zur Repräsentation von Differenzen

Allgemein läßt sich das Erkennen von Änderungen in Informationsbeständen durch die vier Teilschritte *Analyse, Zuordnung, Repräsentation der Unterschiede* und *Bewertung der erkannten Unterschiede* beschreiben.

Analyse: Die zu vergleichenden Informationsbestände müssen zunächst analysiert werden.

Zuordnung: Bei einer Zuordnung wird versucht, möglichst große ähnliche Teile der jeweiligen Informationsbestände einander zuzuordnen. Bei Sequenzen sind dies möglichst lange Teil-

sequenzen und bei Bäumen nach Möglichkeit komplette Teilbäume. Je größer die einander zugeordneten Teile sind, desto kompakter sind in der Regel die Änderungen.

Repräsentation der Unterschiede: Nachdem Gemeinsamkeiten der Informationsbestände erkannt wurden, können die Unterschiede herausgearbeitet und beispielsweise in Form eines *edit-scripts* dargestellt werden.

Bewertung der erkannten Unterschiede: Da es in der Regel mehrere Möglichkeiten gibt, die Differenzen in unterschiedlichen Informationsbeständen zu beschreiben, werden diese nach bestimmten Kriterien (Umfang, Kosten der Änderungsoperationen) bewertet. Diese Bewertung bildet die Grundlage für die Auswahl einer der Beschreibungen.

Nachfolgend wird anhand der Realisation durch *edit-scripts* näher auf den Teilschritt *Repräsentation der Unterschiede* eingegangen.

Die Differenz zweier ganzer Zahlen läßt sich durch eine dritte Zahl angeben. Für die Darstellung der Unterschiede strukturierter Informationsbestände bedarf es hingegen komplexerer Strukturen zur Angabe ihrer Differenz. Der Unterschied zweier Informationsbestände A und B wird häufig durch eine Sequenz von Operationen $es = (op_1, ..., op_n)$ mit $op_i \in OPR$ beschrieben [WF74, Mye86, ZS89, SZ90, WMMM90, SWZS94, ZWS96, CGM97](die Menge OPR bildet dabei das *Operationen-Repertoire*). Derartige Sequenzen werden als *edit-scripts* (oder als *Änderungsfolgen*) bezeichnet [Mye86, WMMM90, CGM97] und werden auch in dem hier entwickelten Verfahren angewendet.

Eine einzelne Operation op formt einen Informationsbestand A_0 in einen Informationsbestand A_1 um ($A_0 \xrightarrow{op} A_1$). Die Anwendung einer Folge derartiger Operationen $es = (op_1, op_2, ..., op_n)$ auf einen Informationsbestand A_0 kann durch schrittweises Anwenden der einzelnen Operationen verstanden werden ($A_0 \xrightarrow{op_1} A_1 ... \xrightarrow{op_n} A_n$). Die Wirkung dieser Anwendung wird abkürzend durch $A_0 \overset{es}{\Longrightarrow} A_n$ beschrieben. Diese Operationsfolge ist ein *edit-script* der A_0 in A_n umformt [CGM97].

Es gibt mehr als nur einen *edit-script*, der einen Informationsbestand A in den Informationsbestand B überführt. Betrachtet man beispielsweise die beiden Zeichenketten S_A = "abc" und S_B = "aac", so lassen sich beispielsweise die drei folgenden (in ihrer Wirkung äquivalenten) *edit-scripts* angeben:

- ändere das zweite Zeichen in S_A von "b" zu "a" (*update*).
- entferne das zweite Zeichen in S_A, und füge ein neues Zeichen "a" in S_A an zweiter Stelle ein (*delete, insert*).
- entferne alle Zeichen aus S_A, und füge die drei Zeichen "a", "a" und "c" ein.

Es sind sogar unendlich viele *edit-scripts* denkbar, die S_A in S_B überführen (in ihrer Wirkung aber äquivalent sind): das Paar zweier Operationen ($op_{ins}; op_{del}$), die ein einzelnes Zeichen einfügen und anschließend wieder löschen, kann beliebig häufig ausgeführt werden.
Aus einer unendlich großen Menge ES von *edit-scripts*, die A in B überführen, sind bei der Ermittlung des Unterschieds von A zu B jedoch nur diejenigen $es \in ES$ mit bestimmten Eigenschaften (hauptsächlich die Anzahl der in es vorhandenen Operationen) von Interesse. Unterschiedliche, in ihrer Wirkung aber äquivalente *edit-scripts* können die gleiche Länge aufweisen (insbesondere können es auch mehrere sein, die die minimale Länge aufweisen), sie unterscheiden sich oft nicht nur in der Reihenfolge der auszuführenden Operationen, sondern auch in der Art der Operationen. Daher sollten sich den einzelnen Operationen $op_i \in OPR$ Kosten zuordnen lassen, um *edit-scripts* gleicher Länge bewerten zu können.

Neben der Verwendung von *edit-scripts* gibt es noch weitere Möglichkeiten, die Unterschiede zwischen zwei Informationsbeständen darzustellen. Bei wenig komplexen Strukturen, wie den oben erwähnten Zahlen läßt sich die Differenz durch Angabe eines weiteren Informationsbestandes der

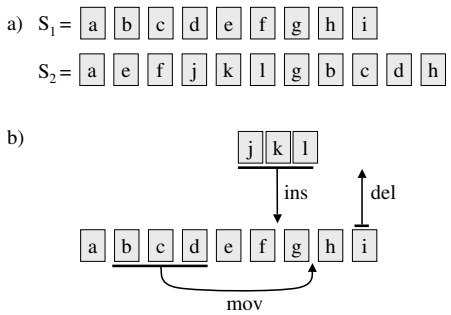

Abbildung 4.1: a) Zwei miteinander zu vergleichende Sequenzen. b) Darstellung der Differenzen der beiden Sequenzen

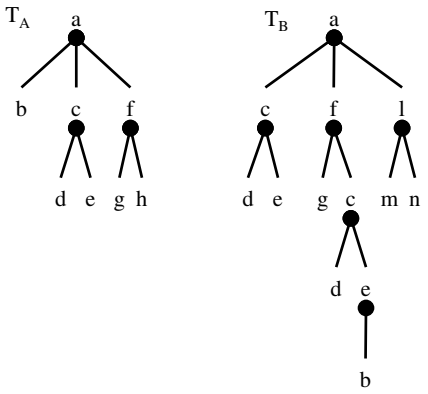

Abbildung 4.2: Zwei miteinander zu vergleichende Bäume

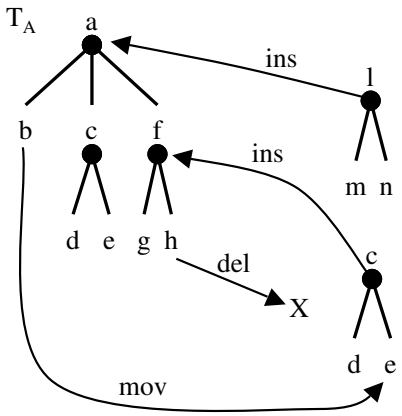

Abbildung 4.3: *delta tree*: Darstellung der Differenz der Bäume aus Abbildung 4.2

selben Art darstellen. Bei Sequenzen läßt sich der Basisbestand so beschriften, daß die Differenz zwischen Basisbestand und Zielbestand aus dieser Beschriftung erkenntlich ist (Abbildung 4.1). Bei der Untersuchung der Unterschiede zwischen Bäumen lassen sich Differenzen durch sogenannte *delta trees* darstellen [CRGMW96]. Dieses ist der Basisbaum, dessen Knoten mit den vorzunehmenden Änderungen beschriftet werden (Abbildung 4.3). Sobald jedoch das Kopieren eines Bestandteils des Basisbestandes stattfindet, lassen sich Differenzen nicht mehr auf diese Weise darstellen. Wird ein sich im Basisbestand befindliches Element m kopiert, erscheinen im Zielbestand sowohl m als auch seine Kopie m'. Würden Differenzen im Basisbestand ähnlich wie in den Abbildungen 4.1 und 4.3 dargestellt werden, wäre nicht klar, ob m, m' oder beide von Änderungen am Knoten m betroffen sind, da die zeitliche Reihenfolge der Operationen aus der Darstellung nicht hervorgeht.

Sowohl die im Folgenden vorgestellten Verfahren, als auch das in Kapitel 6 beschriebene eigene Verfahren verwenden *edit-scripts* zur Darstellung von Differenzen.

4.2 Vergleich von Sequenzen

Sequenzen sind geordnete Folgen von Symbolen eines beliebigen Alphabets. Ein Beispiel für derartige Strukturen stellen Zeichenketten dar (geordnete Folge einzelner Zeichen). Die Unterschiede zweier derartiger Strukturen lassen sich durch *edit-scripts* beschreiben, die im Verlauf der Behandlung weiterer Algorithmen zum Vergleich komplexerer Strukturen noch erweitert werden.

Das Problem, Unterschiede in Sequenzen von Elementen aufzufinden, ist recht intensiv behandelt worden[1]. Hier wird das in [WMMM90] beschriebene Verfahren kurz vorgestellt, da dieses eines der neueren Verfahren ist und die Differenz zweier Sequenzen als *edit-script* beschreibt.

Für ein gegebenes Alphabet Σ, bezeichne Σ^* die Menge aller Wörter, die sich durch Hintereinanderschreiben (Konkatenation) von Symbolen aus Σ bilden lassen. Dieses ist nichts anderes als die Menge aller endlichen Folgen (Sequenzen) von Elementen aus Σ. Dies schließt auch die leere Folge, bzw. das leere Wort ein, welches mit ϵ beschrieben wird. Falls beispielsweise $\Sigma = \{a, b\}$, so ist

$$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, ...\}$$

die Menge aller Sequenzen dieses Alphabets [Sch92]. Für alle Paare $\Sigma \times \Sigma$ von Elementen des Alphabets Σ sei die Frage der Äquivalenz beantwortbar. Die Sequenz $A = a_1, a_2, ..., a_n$ mit $a_i \in \Sigma$ läßt sich auch schreiben als $(a_1, ..., a_n)$. Sind $A = (a_1, ..., a_m), B = (b_1, ..., b_n)$ und $C = (c_1, ..., c_o)$ Sequenzen, dann ist die Sequenz C eine *Untersequenz* der Sequenz A, wenn sie durch Entfernen einzelner Elemente aus A hergestellt werden kann. C ist *gemeinsame Untersequenz* der Sequenzen A und B, wenn sie eine Untersequenz sowohl von A als auch von B ist. Die *längste gemeinsame Untersequenz - LCS* ergibt sich als diejenige gemeinsame Untersequenz, die gegenüber allen existierenden gemeinsamen Untersequenzen die größte Länge aufweist [WMMM90].

Zur Darstellung der Differenz zweier Sequenzen werden *edit-scripts* vorgeschlagen, deren einzelne Operationen sich als Funktionen $op : \Sigma^* \rightarrow \Sigma^*$ beschreiben lassen. Genauer läßt sich die *insert*-Operation durch $op_{ins} : (\Sigma^* \times \mathbb{N} \times \Sigma) \rightarrow \Sigma^*$ und die *delete*-Operation durch $op_{del} : (\Sigma^* \times \mathbb{N}) \rightarrow \Sigma^*$ beschreiben. Die natürliche Zahl gibt dabei die Position für das Einfügen beziehungsweise für das Löschen an.

In [WF74] werden die drei dort beschriebenen Operationen *update, insert* und *delete* in Anlehnung an formale Sprachen als *Produktionen* betrachtet. Eine einzelne Änderungsoperation läßt sich dann als geordnetes Paar $(a, b) \neq (\Lambda, \Lambda)$ zweier Sequenzen der Länge ≤ 1 auffassen. Dabei bezeichnet

[1] Siehe Literaturangaben 2, 4, 5, 7, 9, 11, 14 und 16 in [WMMM90].

Λ das leere Zeichen und erlaubt damit die Repräsentation von *delete*- und *insert*-Operationen. Diese Darstellungsweise wird hier nicht weiter verfolgt, da sich die Funktionenschreibweise besser auf komplexere Strukturen übertragen läßt.

Zur Bewertung von *edit-scripts* bei Sequenzen ist die als *Levenshtein-Abstand* bezeichnete Länge eines *edit-scripts* von besonderem Interesse [WMMM90]. Das Problem, einen der kürzesten *edit-scripts* für zwei Sequenzen zu ermitteln (wie oben erwähnt wurde, kann es mehrere gleicher Länge, demnach auch mehrere kürzeste geben), ist analog zu dem Problem, die längste gemeinsame Untersequenz für zwei Sequenzen aufzufinden [WMMM90, Mye86]. Diese beiden Probleme werden im Folgenden beschrieben.

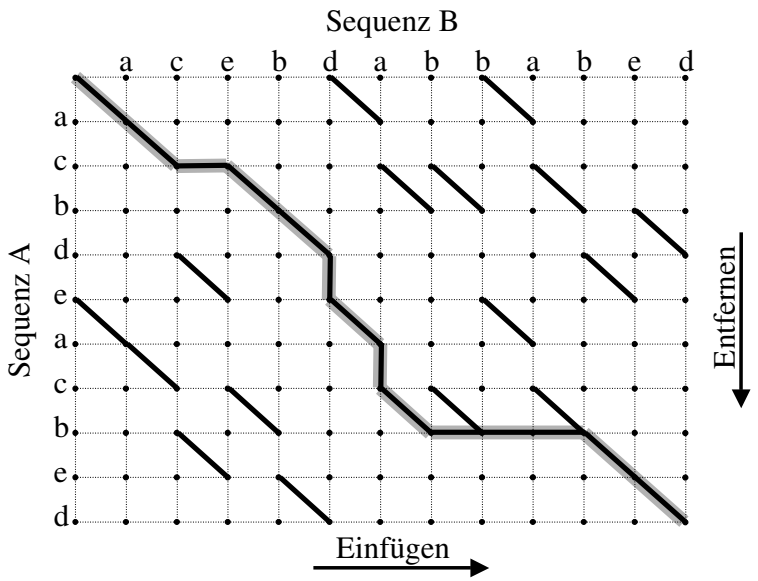

Abbildung 4.4: *edit-Graph* für die Sequenzen A=„acbdeacbed“ und B=„acebdabbabed“

Die Umformung einer Sequenz A in eine Sequenz B läßt sich mit Hilfe eines sogenannten *edit-Graphen* veranschaulichen, wie er in Abbildung 4.4 dargestellt ist. Ein *edit-Graph* ist ein gerichteter Graph mit einem Knoten für jedes Paar $(x, y) \in [0..|A|] \times [0..|B|]$, wobei $|A|$ und $|B|$ die Länge der jeweiligen Sequenz angibt. Von jedem Knoten führt je eine Kante zu seinem rechten Nachbarn (soweit ein solcher existiert) und eine Kante zu seinem unteren Nachbarn (soweit ein solcher existiert). Eine Kante, die von dem Knoten $(2, 2)$ zu dem Knoten $(2, 3)$ verläuft, repräsentiert das Einfügen des Elementes b_3 in der Sequenz B, analog dazu entspricht eine Kante von $(4, 5)$ nach $(5, 5)$ dem Entfernen des Elementes a_5 der Sequenz A. Zusätzlich erhält der Graph diagonale Kanten für je zwei äquivalente Elemente beider Sequenzen: eine diagonale Kante $(x, y) \rightarrow (x+1, y+1)$ existiert genau dann, wenn $a_{x+1} \equiv b_{y+1}$ gilt.

Ein Verfahren zum Auffinden der längsten gemeinsamen Untersequenz der in einem derartigen *edit-Graphen* dargestellten Sequenzen versucht, einen möglichst kurzen Pfad von dem Knoten $(0, 0)$ zum Knoten $(|A|, |B|)$ verlaufend zu finden, was bedeutet, daß ein solcher Pfad möglichst viele diagonale Kanten verwenden sollte, wie in der Abbildung 4.4 hervorgehoben eingezeichnet. Je mehr diagonale Kanten im letztlich gefundenen Pfad von dem Punkt $(0; 0)$ zum Punkt $(|A|; |B|)$ verwendet werden, desto weniger waagerechte *Einfügen*- und senkrechte *Entfernen*-Kanten müssen verwendet werden. Das in [WMMM90] beschriebene Verfahren macht sich die Tatsache zunutze, nicht alle diagonale Kanten bei der Suche nach einem der kürzesten Pfade überprüfen zu müssen,

und erzielt damit eine Laufzeitabschätzung von $O(NP)$, wobei N der Länge der Sequenz B und P der Anzahl der notwendigen Löschoperationen entspricht.

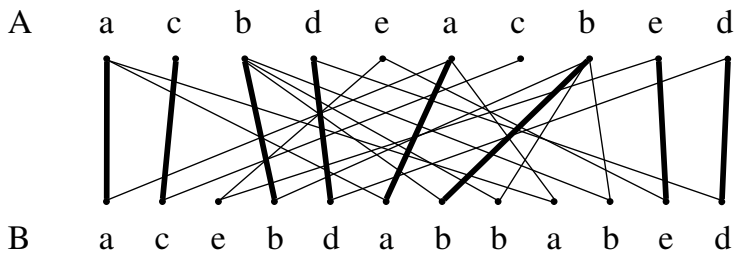

Abbildung 4.5: Bipartiter Graph für die Sequenzen A=„acbdeacbed“ und B=„acebdabbabed“

Eine Darstellungsweise, die in Kapitel 4.3 sehr von Nutzen ist, verwendet einen bipartiten Graphen[2] G_{bip}, dessen beiden Knotenmengen die Elemente der jeweiligen zu vergleichenden Sequenzen repräsentieren und dessen Kanten sich auf die Äquivalenz der entsprechenden Elemente beziehen. Das Problem, eine möglichst lange gemeinsame Untersequenz zu finden, entspricht dann dem Problem, einen planaren Teilgraphen[3] von G_{bip} mit möglichst vielen Kanten zu finden. Lösungen dieses Problems finden sich in [MW98].

4.3 Vergleich von Bäumen

Weisen die zu vergleichenden Informationsbestände hierarchische Strukturen auf, so lassen sie sich mit Hilfe von Bäumen darstellen. Bevor mit der Betrachtung von Verfahren zum Vergleich von Bäumen begonnen wird, sind einige Begriffsklärungen sinnvoll:

Ein Graph $T = (V, E)$ heißt *Baum*, wenn er zyklenfrei und zusammenhängend ist. Meist wird ein bestimmter Knoten $w \in V$ als *Wurzel* bezeichnet. Da T zyklenfrei und zusammenhängend ist, gibt es für zwei Knoten $v_1, v_2 \in V$ genau einen Weg, der diese Knoten verbindet. Für jeden Knoten $v \in V, v \neq w$ gibt es einen v auf dem Weg zur Wurzel nächstgelegenen Knoten $p(v)$, der als *Vater* von v bezeichnet wird. Alle anderen benachbarten Knoten von v bilden die Menge seiner *Kinder* $c(v)$. Für einen Knoten $v \in V$ bezeichnet $p^i(v)$ denjenigen Knoten, der auf dem Weg von v zur Wurzel w liegt und durch einen Weg der Länge i von v getrennt ist. Demnach bezeichnet beispielsweise $p^0(v)$ den Knoten v selbst und $p^2(v)$ den Vater des Vaters von v. Ein Knoten $v \in V$ wird als *Blatt* bezeichnet, wenn v keine Kinder hat. Alle Knoten, die weder Blatt noch Wurzel sind, werden *innere Knoten* genannt. Häufig betrachtet man *beschriftete* Bäume: sei L eine Menge von Bezeichnern, dann gibt man zusätzlich zu den Mengen der Knoten und Kanten eine Funktion $l : V \rightarrow L$ als *Beschriftung* an, die jedem Knoten einen Bezeichner der Menge L zuordnet[Jun90, Aig84, Hal89].
Sei m ein Knoten eines Baumes B, dann bezeichne $T(m)$ den Teilbaum von B, dessen Wurzel m ist. Das Prädikat $\in$ gelte auch für Bäume: $m \in T$ sei *wahr*, wenn die Knotenmenge des Baumes T den Knoten m enthält, und sonst *falsch*.

Bei der Betrachtung des Vergleichs von Bäumen ist es sinnvoll, zwischen geordneten Bäumen und ungeordneten Bäumen zu unterscheiden. Spielt die Reihenfolge der Kinder von Knoten eine Rolle, spricht man von *geordneten* Bäumen, andernfalls von *ungeordneten* Bäumen. Die Abbildung 4.6

[2] **Bipartiter Graph:** Ein bipartiter Graph $G_{bip} = (V_1, V_2, E)$ ist ein Graph mit zwei Knotenmengen V_1 und V_2 und einer Kantenmenge, die nur Kanten enthält, die Knoten aus V_1 mit Knoten aus V_2 verbinden [Jun90].

[3] **Planarer Graph:** Ein Planarer Graph $G = (V, E)$ ist ein Graph, der sich ohne überflüssige Schnittpunkte in der Ebene zeichnen läßt [Jun90].

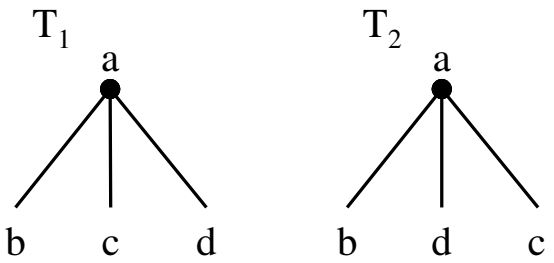

Abbildung 4.6: Geordnete vs. ungeordnete Bäume

zeigt zwei Bäume, die sich nur in der Reihenfolge der Kinder der jeweiligen Wurzel unterscheiden. Faßt man diese beiden Bäume als geordnete Bäume auf, so sind sie unterschiedlich. Faßt man sie hingegen als ungeordnete Bäume auf, so sind sie gleich. Soll beispielsweise ein Text durch einen Baum repräsentiert werden, sollte dieses durch einen geordneten Baum geschehen. Stammbäume aus der Biologie werden hingegen üblicherweise durch ungeordnete Bäume dargestellt. Die Eigenschaft der Ordnung läßt sich noch differenzierter auf die Knoten der Bäume beziehen: während die Reihenfolge der Kinder einzelner Knoten eines Baumes wichtig ist, ist die Reihenfolge der übrigen Knoten des Baumes nicht wichtig.

Abhängig davon, ob sich die zu vergleichenden Informationsbestände durch geordnete oder ungeordnete Bäume repräsentieren lassen, fallen die Lösungen des Problems des Auffindens eines *edit-scripts* in unterschiedliche Problemklassen.
Prinzipiell ist der Vergleich geordneter Bäume einfacher als der ungeordneter Bäume, da sich bei geordneten Bäumen Verfahren zur Auffindung längster gemeinsamer Untersequenzen einsetzen lassen [CRGMW96].
Die folgenden beiden Abschnitte beschreiben jeweils ein Verfahren, das stellvertretend für die Verfahren aus der jeweiligen Problemklasse stehen soll.

4.3.1 Vergleich geordneter Bäume

In diesem Abschnitt wird ein Verfahren beschrieben, das sich mit dem Vergleich geordneter Bäume beschäftigt. Aus einer Anzahl bekannter Verfahren zu diesem Thema [ZS89, SZ90, CRGMW96] wird das in [CRGMW96] beschriebene Verfahren *LaDiff* kurz vorgestellt. Der Grund dafür liegt darin, daß das Verfahren *LaDiff* die Grundlage für das in [CGM97] beschriebene Verfahren *MH-Diff* darstellt, welches wiederum die Grundlage für die eigene Implementierung ist. Darüber hinaus bietet das Verfahren *LaDiff* für die Erzeugung von *edit-scripts* ein umfangreicheres Operationen-Repertoire an als die anderen Verfahren: Neben dem Einfügen, Entfernen und Ändern von Knoten wird auch das Verschieben von Teilbäumen erkannt.

Das Verfahren *La-Diff*

Nach einer kurzen Einleitung, in der die von dem Verfahren verwendeten Operationen für den *edit-script* beschrieben werden, werden die beiden Schritte, aus denen sich das Verfahren zusammensetzt, erläutert. Anschließend erfolgt eine kurze Beschreibung der Einschränkungen des Verfahrens.

Das Operationen-Repertoire:

Das Verfahren untersucht zwei geordnete Baumstrukturen T_A und T_B (wie in Abbildung 4.7) mit dem Ziel, die Differenz dieser in Form eines *edit-scripts* zu ermitteln. Das dabei verwende-

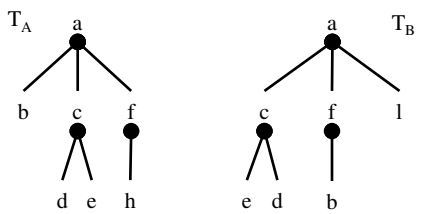

Abbildung 4.7: Zwei miteinander zu vergleichende Bäume T_A und T_B

te Operationen-Repertoire umfaßt zusätzlich zu den bei anderen Verfahren verwendeten *insert-*, *delete-* und *update*-Operationen [ZS89, SZ90] auch die *move*-Operation, die die Aussagekraft des *edit-scripts* erhöht.

Die Operationen beziehen sich auf den Baum T_A und modifizieren diesen schrittweise. Dabei bezeichnen n und p Knoten des Baumes T_A, v eine Beschriftung eines Knotens und k die einzunehmende Position eines Knotens innerhalb der Kinder seines neuen Vaters:

***update*-Operation:** `upd(n,v)` Die *update*-Operation ändert die Beschriftung des Knotens n, so daß sie v entspricht.

***insert*-Operation:** `ins(v,p,k)` Die *insert*-Operation erzeugt einen neuen Knoten mit der Beschriftung v, der Kind der Position k des Knoten p wird.

***delete*-Operation:** `del(n)` Die *delete*-Operation entfernt den Knoten n, wobei davon ausgegangen wird, daß n keine Kinder hat[4].

***move*-Operation:** `mov(n,p,k)` Die *move*-Operation verschiebt den bei n beginnenden Teilbaum $T(n)$, so daß p der neue Vater von n wird, und der Knoten n an der Position k innerhalb der Kinder von p erscheint.

Die Knoten der Bäume brauchen keine eindeutigen Identifizierungen zu tragen; die einzigen vergleichsrelevanten Merkmale der Bäume stellen die Beschriftungen der Knoten sowie deren Kinder dar. Die Beschriftung eines Knotens m wird mit $v(m)$ bezeichnet.

Vorgehen des Verfahrens:

Das Verfahren sieht grob betrachtet zwei Arbeitsschritte vor, die anschließend etwas genauer erläutert werden:

Ermitteln einer Zuordnung: Ähnlich wie auch in den Verfahren zum Vergleich von Zeichenketten, besteht auch hier der erste Schritt darin, äquivalente Elemente der Strukturen einander zuzuordnen.

Erstellen eines *edit-scripts* aus der gefundenen Zuordnung: In insgesamt fünf Phasen wird aus der ermittelten Zuordnung der *edit-script* erstellt.

Ermitteln einer Zuordnung:

Der letztlich zu erstellende *edit-script* wird aus einer Reihe von Operationen bestehen, die schrittweise auf den Baum T_A ausgeführt, einen zum Baum T_B äquivalenten Baum erzeugen.

[4] Die Kinder des zu entfernenden Knoten müssen gegebenenfalls vorher entfernt oder verschoben worden sein.

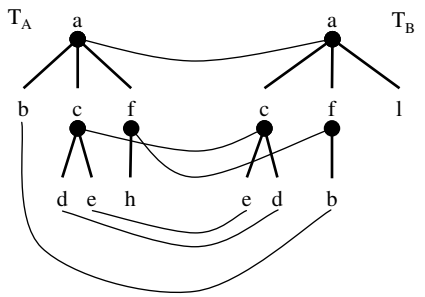

Abbildung 4.8: Die gefundene Zuordnungen für die Bäume T_A und T_B

Beim Ermitteln einer Zuordnung wird versucht, für jeden Knoten $m_a \in T_A$ den zu ihm äquivalenten Knoten $m_b \in T_B$ zu finden. Sollte es für einen Knoten $m_a \in T_A$ keinen äquivalenten Knoten im Baum T_B geben, so besteht einer der Schritte des *edit-scripts* darin, diesen Knoten m_a zu entfernen. Umgekehrt wird bei der Abarbeitung des *edit-scripts* ein Knoten m_b eingefügt, wenn ein Knoten $m_b \in T_B$ keinen Partner in Baum T_A erhält. Durch diese Zuordnung erhält jeder Knoten der Bäume genau einen Partner im jeweils anderen Baum, oder wird als zu entfernender beziehungsweise als einzufügender Knoten betrachtet. In Abbildung 4.8 sind einige der Knoten der beiden Bäume T_A und T_B einander zugeordnet. Ein Knoten aus T_A, der keinem Knoten aus T_B zugeordnet ist (wie hier der Knoten "h"), führt zu einer *delete*-Operation im resultierenden *edit-script*.
Die gefundene Zuordnung läßt sich als eine Menge $\mathcal{K} \subseteq V_A \times V_B$ von Kanten beschreiben[5], die einen Knoten aus T_A mit einem Knoten aus T_B verbinden.
Tragen die Knoten eindeutige Identifizierungen, wird das Finden der Zuordnungen deutlich vereinfacht: Ein Knoten $m_a \in T_A$ wird einem Knoten $m_b \in T_B$ genau dann zugeordnet, wenn sie die gleiche Identifizierung aufweisen. Tragen die Knoten keine eindeutigen Identifizierungen, wird die Zuordnung anhand der Knotenbeschriftung und der Anzahl derjenigen Zuordnungen ermittelt, die Kinder von m_a mit Kindern von m_b verbinden.

Erstellen eines *edit-scripts*:

Das Erstellen des *edit-scripts* verläuft in insgesamt fünf Phasen, wobei dem zu erstellenden *edit-script* in jeder Phase eine Gruppe von Operationen hinzugefügt wird:

***update*-Phase:** In dieser Phase wird nach Paaren $(m; n) \in \mathcal{K}$ gesucht, bei denen $v(m) \neq v(n)$ gilt. Für alle derartigen Paare wird dem *edit-script* jeweils eine entsprechende *update*-Operation hinzugefügt.

***align*-Phase:** In dieser Phase beschäftigt man sich mit den Kindern der Knoten m und n, die einander zugeordnet sind (wie zum Beispiel die beiden mit "c" beschrifteten Knoten in der Abbildung 4.8). In diesem Beispiel sind die mit "d" und "e" beschrifteten Knoten in den Bäumen T_A und T_B nicht in der gleichen Reihenfolge. Durch eine *move*-Operation, die einen der beiden Knoten in T_A verschiebt, läßt sich die in T_B angegebene Reihenfolge erreichen.

***insert*-Phase:** In dieser Phase wird nach Knoten $n \in T_B$ gesucht, denen kein Knoten in T_A zugeordnet ist, deren Vater $pn = p(n)$ jedoch einem Knoten pm in T_A zugeordnet ist. Es wird ein neuer Knoten m mit der gleichen Beschriftung, wie die des Knoten n erzeugt $(v(m) = v(n))$, und dem Knoten pm als neues Kind hinzugefügt, wobei pm der dem Vater des Knoten n zugeordnete Knoten ist. Zusätzlich werden die beiden Knoten m und n einander zugeordnet. In der Abbildung 4.8 ist der mit "l" beschriftete Knoten $l_B \in T_B$ keinem Knoten

[5] Die Mengen V_A und V_B bezeichnen die Knotenmengen der Bäume T_A und T_B

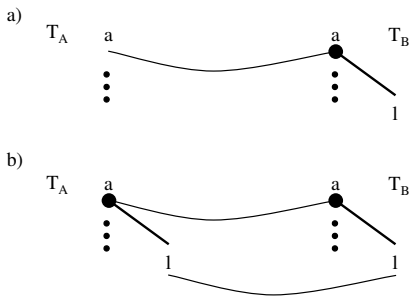

Abbildung 4.9: a) Dem mit "l" beschrifteten Knoten aus T_B ist kein Knoten in T_A zugeordnet. b) In T_A wurde ein neuer mit "l" beschrifteter Knoten eingefügt und dem mit "l" beschrifteten Knoten aus T_B zugeordnet.

in T_A zugeordnet, so daß ein neuer mit "l" beschrifteter Knoten l_A in T_A eingefügt wird und dem Knoten l_B zugeordnet wird (Abbildung 4.9).

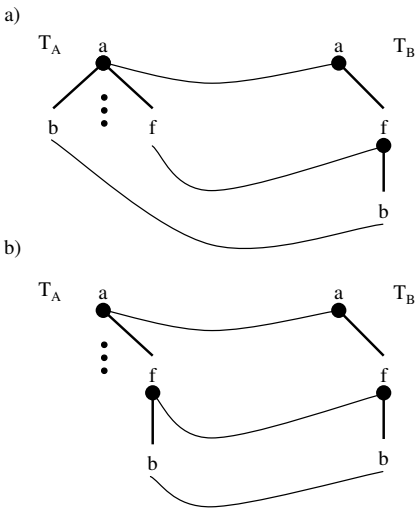

Abbildung 4.10: a) Der mit "b" beschriftete Knoten $b_A \in T_A$ muß verschoben werden, da sein Vater (a_A) nicht dem Vater von b_B zugeordnet ist. b) Das Ziel der Verschiebung ist der Knoten f_A, da dieser dem Vater von b_B zugeordnet ist.

***move*-Phase:** In dieser Phase wird nach einander zugeordneten Knoten $(m; n)$ gesucht, deren Väter $p(m)$ und $p(n)$ nicht einander zugeordnet sind, wie dieses bei den mit "b" beschrifteten Knoten in der Abbildung 4.8 der Fall ist. Das Ziel einer solchen Verschiebung ist der Knoten $pm' \in T_A$, der dem Knoten $p(n)$ zugeordnet ist (siehe Abbildung 4.10).

***delete*-Phase:** In der letzten Phase wird nach Knoten $m \in T_A$ gesucht, denen kein Knoten aus T_B zugeordnet ist und die keine Kinder haben. Diese Knoten werden aus dem Baum T_A entfernt und dem *edit-script* wird eine entsprechende *delete*-Operation hinzugefügt.

Einschränkungen:

Das Verfahren *LaDiff* setzt voraus, daß es für jeden Knoten eines Baumes genau einen ihm zugeordneten Knoten des jeweils anderen Baumes gibt, wenn der Knoten nicht eingefügt beziehungsweise

entfernt wird. Diese Einschränkung führt zu Problemen, wenn es in den Bäumen mehrere gleichbeschriftete Knoten gibt.

Ferner wird angenommen, daß es in Bezug auf die Beschriftungen der Hierarchiestufen (der Knoten im Baum) eine totale Ordnung gibt. Dies vereinfacht das Finden der Zuordnung erheblich: Ist m Kind eines Knoten $pm \in T_A$, der einem Knoten $pn \in T_B$ zugeordnet ist, dann kann m nicht einem „Vorfahren“ von pn zugeordnet sein. Auf der anderen Seite schränkt diese Bedingung die Struktur der Bäume stark ein[6].

Diese beiden Einschränkungen gelten weder für das Verfahren *MH-Diff*, noch für die eigene Implementierung.

4.3.2 Vergleich ungeordneter Bäume

In [ZSS92] wird gezeigt, daß der Vergleich ungeordneter, binärer, beschrifteter Bäume, deren Knoten jeweils mit nur einem von zwei Symbolen beschriftet sind, NP-Vollständig ist. Aus diesem Grund wenden alle hier betrachteten Verfahren für den Vergleich ungeordneter Bäume Heuristiken an.
Das im Abschnitt 4.3.2 beschriebene Verfahren stellt die Grundlage der eigenen Implementierung dar und wird daher ausführlicher beschrieben.

Der Algorithmus *MH-Diff*

Das in [CGM97] beschriebene Verfahren *MH-Diff* untersucht zwei ungeordnete Baumstrukturen T_A und T_B mit dem Ziel, die Differenz dieser beiden in Form eines *edit-scripts* zu ermitteln. Das dabei verwendete Operationen-Repertoire umfaßt zusätzlich zu den bei anderen Verfahren verwendeten *insert-, delete-* und *update*-Operationen [ZS89, SZ90, ZWS96, CRGMW96] auch *move-, copy-* und *glue*-Operationen, die die Aussagekraft des *edit-scripts* erhöhen.
Nach einer kurzen Einleitung wird das Vorgehen des Verfahrens genauer beschrieben. Anschließend werden einige der Einschränkungen des Verfahrens beschrieben.

Der Algorithmus *MH-Diff* stellt eine Weiterentwicklung des in [CRGMW96] beschriebenen Verfahrens dar. Im Gegensatz zu jenem unterliegen die zu vergleichenden Bäume hier weniger Beschränkungen. Insbesondere entfällt die in [CRGMW96] geforderte totale Ordnung der Knoten, sowie die Forderung, daß jeder Knoten nur einmal in seinem Baum vorkommen darf. In Abbildung 4.11 sind zwei miteinander zu vergleichende Bäume T_A und T_B dargestellt.

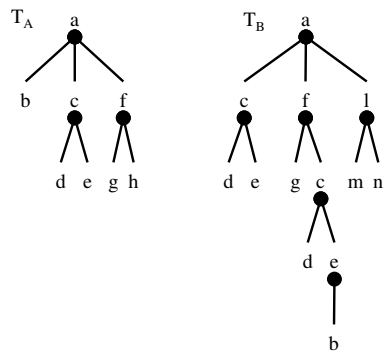

Abbildung 4.11: Zwei miteinander zu vergleichende Bäume T_A und T_B

[6] So sind beispielsweise nicht alle durch XML beschreibbaren Sprachen durch derartige Bäume darstellbar.

Vorgehen des Verfahrens:

Zur Erzeugung des *edit-scripts* sieht der Algorithmus drei Arbeitsschritte vor:

Zuordnung: im ersten Schritt werden die einzelnen Knoten der jeweiligen Bäume so miteinander in Beziehung gesetzt, daß jeder Knoten einen oder mehrere *Partner* im jeweils anderen Baum erhält. Eine derartige Verbindung zwischen einem im Baum T_A vorhandenen Knoten m und einem im Baum T_B vorhandenen Knoten n stellt eine Zuordnung der Gemeinsamkeiten beider Bäume dar.

Beschriftung der Verbindungen: Auf Grundlage dieser gefundenen Zuordnungen werden im zweiten Schritt die Verbindungen mit Beschriftungen versehen.

Erzeugung des *edit-scripts*: Im dritten Schritt wird mit Hilfe der beschrifteten Zuordnungen ein *edit-script* erzeugt.

Im Folgenden werden die drei Schritte genauer erläutert.

Zuordnung der Knoten:

Im ersten Schritt wird eine sogenannte *Kantenüberdeckung* erzeugt, die sich wie folgt definiert: Gegeben sei ein ungerichteter Graph $G = (V, E)$. Eine Kantenmenge $\mathcal{K} \subseteq E$ nennt man Kantenüberdeckung von G, genau dann, wenn gilt: $\forall v_i \in V \; \exists v_j \in V : (v_i, v_j) \in \mathcal{K}$. Intuitiv zeichnet sich eine Kantenüberdeckung $\mathcal{K}$ dadurch aus, daß jeder Knoten des Graphen von mindestens einer Kante aus $\mathcal{K}$ getroffen wird. Eine Kantenüberdeckung ist *minimal*, wenn sie durch das Entfernen einer beliebigen Kante ihre Eigenschaft als Kantenüberdeckung verlieren würde.

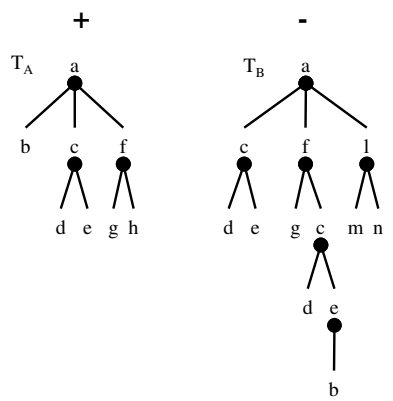

Abbildung 4.12: Zwei miteinander zu vergleichende Bäume T_A und T_B mit *Plus*- und *Minus*-Knoten

Man denke sich einen bipartiten Graphen $G_{Bip} = (V_A, V_B, E)$, dessen Knotenmengen V_A aus den Knoten des Baumes T_A und V_B aus denen des Baumes T_B bestehen. Ferner wird der Menge V_A ein spezieller mit *plus* bezeichneter und der Menge V_B ein spezieller mit *minus* bezeichneter Knoten hinzugefügt (*Plus*- und *Minus*-Knoten). Diese beiden Knoten denke man sich auch den zu vergleichenden Bäumen hinzugefügt (siehe Abbildung 4.12). Die Operationen des zu ermittelnden *edit-scripts* werden schrittweise auf den Baum T_A angewendet, so daß die Verbindung eines Knotens im Baum T_B mit dem *Plus*-Knoten das Einfügen dieses Knotens in den Baum T_A (letztlich einer *insert*-Operation im *edit-script*) symbolisiert. Analog dazu entspricht die Verbindung eines Knotens im Baum T_A mit dem *Minus*-Knoten der *delete*-Operation im *edit-script*. Die

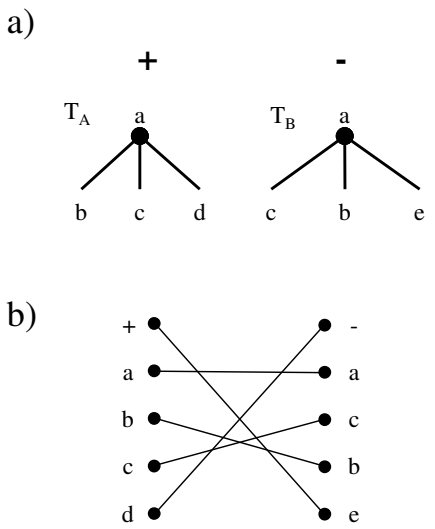

Abbildung 4.13: a) Zwei einfache Bäume T_A und T_B. b) Der aus den Bäumen T_A und T_B resultierende bipartite Graph.

den *Plus-* beziehungsweise *Minus*-Knoten treffenden Kanten, werden *Plus-Kanten* beziehungsweise *Minus-Kanten* genannt. Eine Kante, die den *Plus*-Knoten mit dem *Minus*-Knoten verbindet, wird *Plus-Minus-Kante* genannt. Da G_{Bip} bipartit ist, darf es keine Kanten geben, die zwei Knoten einer Knotenmenge V_A oder V_B miteinander verbinden. Genaugenommen betrachtet man hier also Kopien der Baumknoten, denen die Verbindungen zu ihren im Baum vorhandenen Nachbarn genommen wurde.

Die Ausführung der sich im *edit-script* befindenden Operationen läßt sich so vorstellen, daß sie auf eine Kopie des Baumes T_A angewendet wird, die als *Arbeitsbaum* bezeichnet wird. Dieser Arbeitsbaum wird schrittweise verändert und ist schließlich strukturgleich mit dem Baum T_B.
Das hier verwendete Operationen-Repertoire umfaßt die folgenden sechs Operationen (eine genauere Beschreibung findet sich in [CGM97]). Dabei bezeichnen n, n_1, n_2 und p Knoten des Arbeitsbaumes, v eine Beschriftung eines Knotens und C eine Menge von Knoten.

***update*-Operation:** `upd(n,v)` Die *update*-Operation ändert die Beschriftung des Knotens n, so daß sie v entspricht.

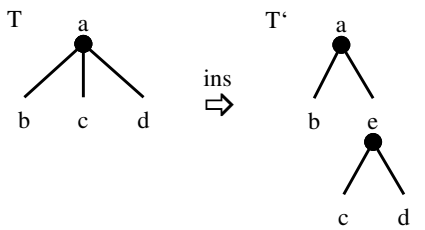

Abbildung 4.14: Ausführung einer *insert*-Operation

***insert*-Operation:** `ins(n,v,p,C)` Die *insert*-Operation erzeugt einen neuen Knoten n mit der Beschriftung v, der Kind des Knoten p wird. Dabei werden diejenigen Kinder von p, die durch die Menge C angegeben werden, so verschoben, daß sie anschließend Kinder von n sind.

Die Abbildung 4.14 zeigt die Ausführung einer *insert*-Operation. Dabei wird ein Knoten n mit der Beschriftung 'e' ($v(n) = 'e'$) eingefügt, so daß er Kind des Knoten p, mit $v(p) = 'a'$ wird. Der Knoten n erhält zwei derjenigen Kinder, die ursprünglich Kinder von p waren. Die Operation sieht aus wie folgt: `ins(n,'e',p,{`c_1`,`c_2`})` mit $v(p) = 'a', v(c_1) = 'c', v(c_2) = 'd'$.

***delete*-Operation:** `del(n)` Die *delete*-Operation entfernt den Knoten n, wobei die Kinder von n nicht entfernt werden, sondern gewissermaßen eine Ebene heraufwandern und Kinder von $P(n)$ werden.

***move*-Operation:** `mov(n,p)` Die *move*-Operation verschiebt den bei n beginnenden Teilbaum $T(n)$, so daß p der neue Vater von n wird.

***copy*-Operation:** `cpy(n,p)` Die *copy*-Operation kopiert den bei n beginnenden Teilbaum $T(n)$ und setzt diese Kopie in den Baum ein, so daß p der neue Vater der Kopie von n wird.

***glue*-Operation:** `glu(`n_1`,`n_2`)` Die *glue*-Operation ist gewissermaßen die zur *copy*-Operation inverse Operation. Sie entfernt den bei n_1 beginnenden Teilbaum $T(n_1)$, wobei davon ausgegangen wird, daß der Teilbaum $T(n_1)$ dem Teilbaum $T(n_2)$ äquivalent ist.

Die Grundidee des Verfahrens besteht in der Annahme, daß eine minimale Kantenüberdeckung des auf diese Weise erzeugten bipartiten Graphen G_{Bip} zu einem *edit-script* führt. Um sich dieses zu veranschaulichen, betrachte man folgende Überlegungen:

Gegeben seien die in Abbildung 4.11 dargestellten zu vergleichenden Bäume T_A und T_B und ein *edit-script* `es=(del(7), ins([l(m,n)], 0, 3), cpy(2,5,1), mov(1,9,0))`, der den Baum T_A in den Baum T_B überführt (siehe Abbildung 4.15).

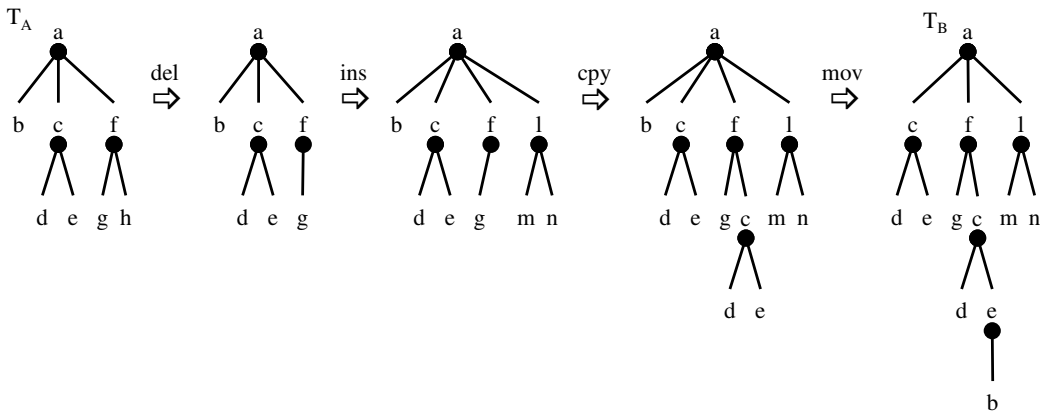

Abbildung 4.15: Ausführung des *edit-scripts*=(del, ins, cpy, mov)

Von dem Baum T_A erstelle man eine als *Arbeitsbaum* bezeichnete Kopie $T_{A'}$ und verbinde jeden Knoten von T_A mit seiner Kopie in $T_{A'}$. Zum Baum T_A gehörend denke man sich den *Plus*-Knoten und zum Baum $T_{A'}$ gehörend den *Minus*-Knoten. Wenn man nun den *edit-script* schrittweise auf $T_{A'}$ anwendet und dabei die Kanten in sinnvoller Weise berücksichtigt, erhält man eine Menge von Kanten zwischen den jeweiligen Bäumen, die einer minimalen Kantenüberdeckung entspricht.

Die Abbildungen 4.16 bis 4.19 zeigen jeweils die Situation (a) vor und (b) nach Ausführung eines der vier Schritte des *edit-scripts*. Die von den jeweiligen Operationen betroffenen Kanten werden in den Abbildungen durch dickere Linien hervorgehoben. Dabei bleibt T_A unverändert, während $T_{A'}$ sukzessive modifiziert wird. In den Abbildungen sind nur diejenigen Kanten eingezeichnet, die von besonderem Interesse sind. Bei der Bearbeitung einer *update*-Operation wird lediglich der

Inhalt des entsprechenden Knoten im Baum $T_{A'}$ geändert. Die mit diesem Knoten verbundene Kante bleibt unverändert. Bei der Bearbeitung einer *move*-Operation wird ein Knoten $n \in T_{A'}$ (mit seinen Kindern) verschoben, so daß er Kind eines anderen Knotens $p \in T_{A'}$ wird. Die mit n verbundene Kante bleibt weiterhin mit n verbunden (Abbildung 4.19). Bei einer *copy*-Operation wird auch der kopierte Knoten mit dem ursprünglichen in T_A verbunden. Zur Erklärung der *insert*- und *delete*-Operationen denke man sich zusätzlich zu den Knoten aus T_A und $T_{A'}$ die *Plus*- und *Minus*-Knoten. Bei einer *delete*-Operation wird ein Knoten n aus $T_{A'}$ entfernt, so daß fraglich ist, was mit der Kante $e = (m; n)$ wird, die n traf. Sie wird in eine Kante $(m; [-])$ verwandelt, also mit dem *Minus*-Knoten verbunden. Analog verfährt man mit der *insert*-Operation (Abbildung 4.17). Bei der *glue*-Operation wird der verbleibende Knoten aus $T_{A'}$ mit zwei Knoten in T_A verbunden (ohne Abbildung).

Nach vollständiger Abarbeitung des *edit-scripts* erhält man einen zu T_B isomorphen Baum $T_{A'}$, dessen Knoten mit denen des Baumes T_A in einer Weise verbunden sind, daß diese Kanten eine minimale Kantenüberdeckung des oben genannten bipartiten Graphen darstellen. Idee des Verfahrens ist es nun, daß es umgekehrt möglich sein sollte, mit Hilfe einer gegebenen Kantenüberdeckung, einen *edit-script* zu ermitteln.

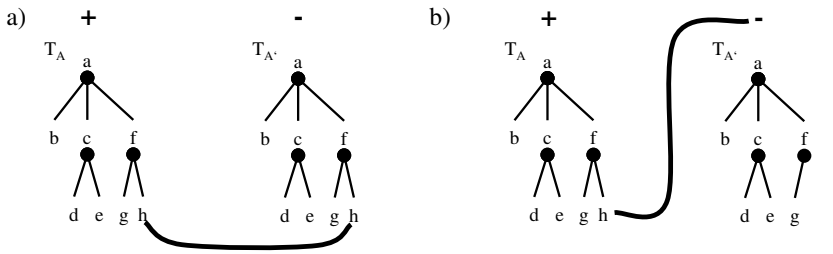

Abbildung 4.16: (a) Vor und (b) nach der *delete*-Operation

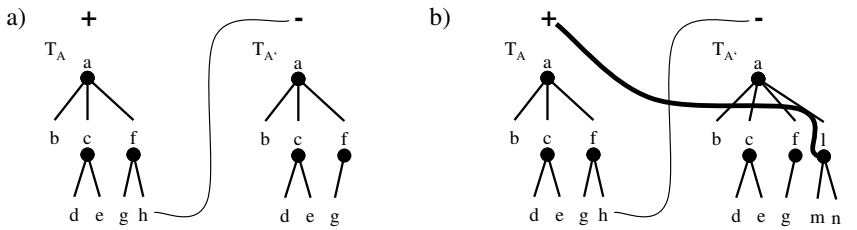

Abbildung 4.17: (a) Vor und (b) nach der *insert*-Operation

Aufbau der Kantenüberdeckung:

Das beschriebene Verfahren *MH-Diff* sieht nun vor, eine minimale Kantenüberdeckung zu finden, um aus dieser einen *edit-script* zu erzeugen. Dabei wird diese Kantenüberdeckung des oben erwähnten bipartiten Graphen G_{Bip} nicht schrittweise aufgebaut. Die Kantenmenge E des bipartiten Graphen erhält für jedes Element von $(V_A \times V_B)$ jeweils eine Kante, was bedeutet, daß jeder Knoten des Baumes T_A (einschließlich des *Plus*-Knoten) mit jedem Knoten des Baumes T_B (einschließlich des *Minus*-Knotens) verbunden wird. Durch diese Partnerbeziehungen erhält man einen *vollständigen bipartiten* Graphen (siehe Abbildung 4.20 und [Jun90]).

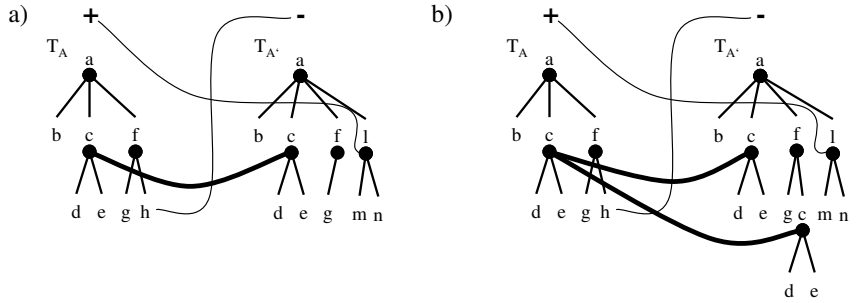

Abbildung 4.18: (a) Vor und (b) nach der *copy*-Operation

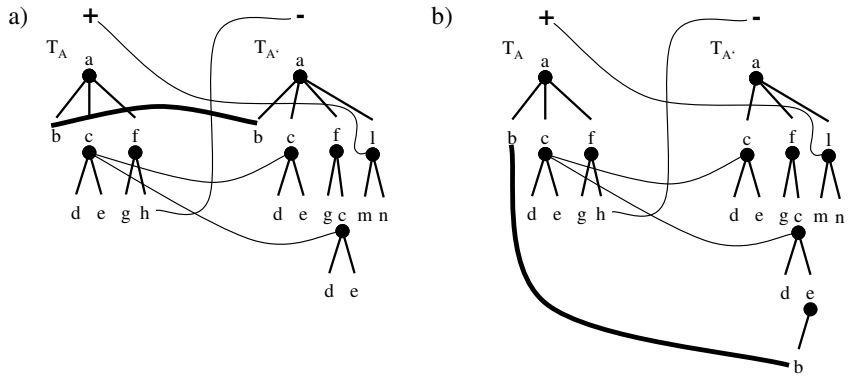

Abbildung 4.19: (a) Vor und (b) nach der *move*-Operation

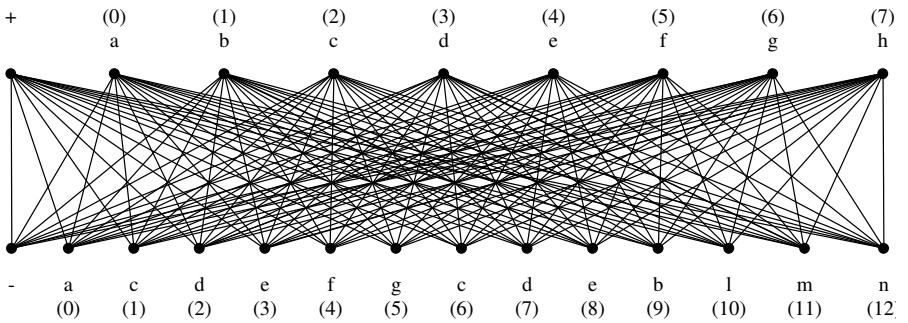

Abbildung 4.20: Vollständiger bipartiter Graph für die Bäume T_A und T_B

Gesucht wird nun eine (oben erwähnte) minimale Kantenüberdeckung dieses bipartiten Graphen. Eine derartige Kantenüberdeckung zeichnet sich dadurch aus, daß sich keine in ihr enthaltene Kante entfernen läßt, ohne ihre Eigenschaft als Kantenüberdeckung zu zerstören. Damit hat sie automatisch folgende Eigenschaft: Sei durch die minimale Kantenüberdeckung $\mathcal{K}$ ein Knoten m des Baumes T_A mit mehr als einem Knoten n_i des Baumes T_B verbunden, dann sind alle Knoten n_i nur mit einem Knoten verbunden - nämlich mit m.

Aus jeder minimalen Kantenüberdeckung läßt sich mindestens ein *edit-script* generieren. Jedoch ist ein derart generierter *edit-script* nicht automatisch minimal. Man denke sich beispielsweise alle Knoten des Baumes T_A mit dem *Minus*-Knoten und alle Knoten des Baumes T_B mit dem *Plus*-Knoten verbunden. Eine derartige Kantenmenge stellt eine minimale Kantenüberdeckung dar, führt jedoch zu dem vermutlich nicht erwünschten *edit-script*, in dem alle Knoten aus T_A entfernt werden und alle Knoten aus T_B eingefügt werden. Demnach sucht man nicht nach *irgendeiner* minimalen Kantenüberdeckung, sondern nach einer, die zu einem minimalen *edit-script* führt. Die Minimalität eines *edit-scripts* wird einerseits durch seine Länge und andererseits auch durch die Art seiner Operationen bestimmt. Jeder Operation op_i des *edit-scripts* sind Kosten $w(op_i)$ zugeordnet. Da jede Kante e einer minimalen Kantenüberdeckung zu einer Operation des *edit-scripts* führt, läßt sich diese Operation als $op(e)$ beschreiben, womit sich die Kosten $w(op(e))$ der Kante e als ihr Gewicht $w(e)$ zuordnen lassen. Für eine Kantenüberdeckung $\mathcal{K} = \{e_1, ..., e_n\}$ entstehen die Kosten $C(\mathcal{K}) = \sum_{e \in \mathcal{K}} w(e)$. Damit wäre es möglich, aus der Menge aller minimalen Kantenüberdeckungen diejenige auszuwählen, deren Kosten im Vergleich zu denen der anderen minimal sind. Gesucht wird also nach einer in zweifacher Hinsicht minimalen Kantenüberdeckung, die im Folgenden *gewichtsminimale Kantenüberdeckung* genannt wird. Sie ist sowohl minimal in Bezug darauf, daß sie eine minimale Kantenüberdeckung darstellt, als auch darauf, daß die Summe ihrer Kantengewichte, verglichen mit der aller anderen minimalen Kantenüberdeckungen, minimal ist[7]. Aus so einer gewichtsminimalen Kantenüberdeckung ergäbe sich ein minimaler *edit-script*. Der tatsächliche Kostenbeitrag einer Kante in einer Kantenüberdeckung $\mathcal{K}$ läßt sich jedoch erst dann ermitteln, wenn man den zu $\mathcal{K}$ gehörenden *edit-script* erzeugt hat. Um diesem Dilemma zu entgehen, schlagen die Autoren folgenden Ansatz vor: man errechnet für jede Kante die zu erwartenden *minimalen* und *maximalen* Kosten und ist somit in der Lage, iterativ Kanten zu entfernen. Hierzu werden zwei *Beschneidungsregeln* (*pruning rules*) angegeben.

Um die Berechnung der minimalen und maximalen Kosten einer Kante zu veranschaulichen, sind zunächst einige Vereinbarungen notwendig: Sei K die aktuelle Kantenüberdeckung und c_i, c_d, c_m, c_c und c_g die jeweiligen Kosten für das *Einfügen*, *Löschen*, *Verschieben*, *Kopieren* und *Einschmelzen* eines Knoten. Ferner bezeichne $C(m)$ die Menge der Kinder des Knoten m und seien die erzwungenen Verschiebekosten (*forced move costs*): $c_{mf}(m', n)$ eines Knoten $m' \in T_A$ in Bezug zu einem Knoten $n \in T_B$ wie folgt definiert:

$$c_{mf}(m', n) = c_m \texttt{ wenn } \neg\exists n' \in C(n), \texttt{ sodaß } [m', n'] \in K, \texttt{ sonst } 0$$

Ähnlich werden die bedingten Verschiebekosten (*conditional move costs*): $c_{m?}(m', n)$ eines Knoten $m' \in T_A$ in Bezug zu einem Knoten $n \in T_B$ definiert:

$$c_{m?}(m', n) = c_m \texttt{ wenn } \neg\exists n' \in C(n), \texttt{ sodaß } [m', n'] \in K, \texttt{ sonst } 0$$

Mit Hilfe dieser Vereinbarungen lassen sich die minimalen und maximalen Kosten einer Kante berechnen:

$$c_{min}([m, n]) = c_u(m, n) + \frac{1}{2} min \left\{ \sum_{m' \in C(m)} c_{mf}(m', n), \sum_{n' \in C(n)} c_{mf}(n', m) \right\} \tag{4.1}$$

[7] Eine Kantenüberdeckung, deren Summe der Kantengewichte minimal ist, ist automatisch eine minimale Kantenüberdeckung, wenn es keine Kante mit dem Gewicht 0 gibt.

$$
\begin{aligned}
c_{max}([m,n]) &= c_u(m,n) \\
&+ \frac{1}{2} \sum_{m' \in C(m)} (c_c(|E(m')| - 1) + c_{m?}(m', n)) \\
&+ \frac{1}{2} \sum_{n' \in C(n)} (c_g(|E(n')| - 1) + c_{m?}(n', m))
\end{aligned} \tag{4.2}
$$

Damit lassen sich die beiden Beschneidungsregeln angeben:

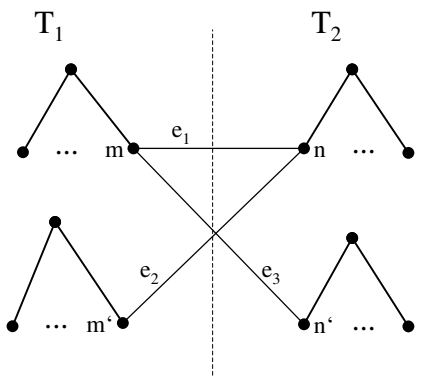

Abbildung 4.21: Beschneidungsregel 1

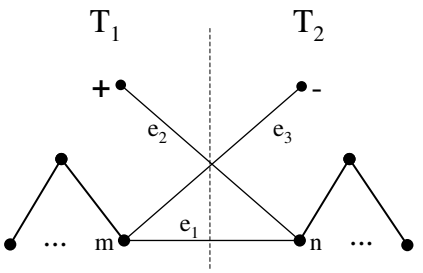

Abbildung 4.22: Beschneidungsregel 2

Beschneidungsregel 1 : Seien die Kanten e_1, e_2 und e_3 wie in Abbildung 4.21 positioniert, und sei $c_t = max\{c_m, c_c, c_g\}$. Wenn $c_{min}(e_1) \geq c_{max}(e_2) + c_{max}(e_3) + 2c_t$, dann entferne e_1.

Beschneidungsregel 2 : Seien die Kanten e_1, e_2 und e_3 wie in Abbildung 4.22 positioniert. Wenn $c_{min}(e_1) \geq c_d(m) + c_i(n)$, dann entferne e_1.
Die beiden Regeln werden durch die Abbildungen 4.21 und 4.22 veranschaulicht. Das Verfahren prüft für jede Kante der aktuellen Kantenüberdeckung, ob eine der beiden Beschneidungsregeln angewendet werden kann. Bei iterativer Anwendung dieser Beschneidungsregeln nähern sich minimale und maximale Kosten einiger Kanten einander. Auf diese Weise ist der Algorithmus in der Lage, Kanten zu entfernen, die sicherlich nicht Teil einer gewichtsminimalen Kantenüberdeckung des bipartiten Graphen sind. Man erhält den *gestutzten induzierten Graph* (*pruned induced graph*), der einige Kanten weniger enthält als der vollständige induzierte Graph. Um aus dem gestutzten induzierten Graphen eine gewichtsminimale Kantenüberdeckung zu erzeugen, wird für jede Kante

ein konstantes Gewicht definiert, welches durch die zuvor beschriebenen minimalen Kosten, die maximalen Kosten, dem Mittelwert der beiden oder einer anderen aus diesen beiden berechenbaren Funktion gegeben sein kann. Während sich die minimalen und maximalen Kosten einer Kante durch das Entfernen anderer Kanten ändern konnten, ändert sich dieses konstante Gewicht der Kanten nicht mehr. Durch Reduktion des Kantenüberdeckungsproblems auf ein *gewichtetes Korrespondenzproblem*[8] eines gewichteten bipartiten Graphen [PS82, Ata99], ist man daraufhin in der Lage, die gewichtsminimale Kantenüberdeckung zu finden. Dieses Korrespondenzproblem wiederum kann beispielsweise mit dem *Ungarischen Algorithmus* mit einer Zeitkomplexität von $O(n^3)$ gelöst werden [Jun90].

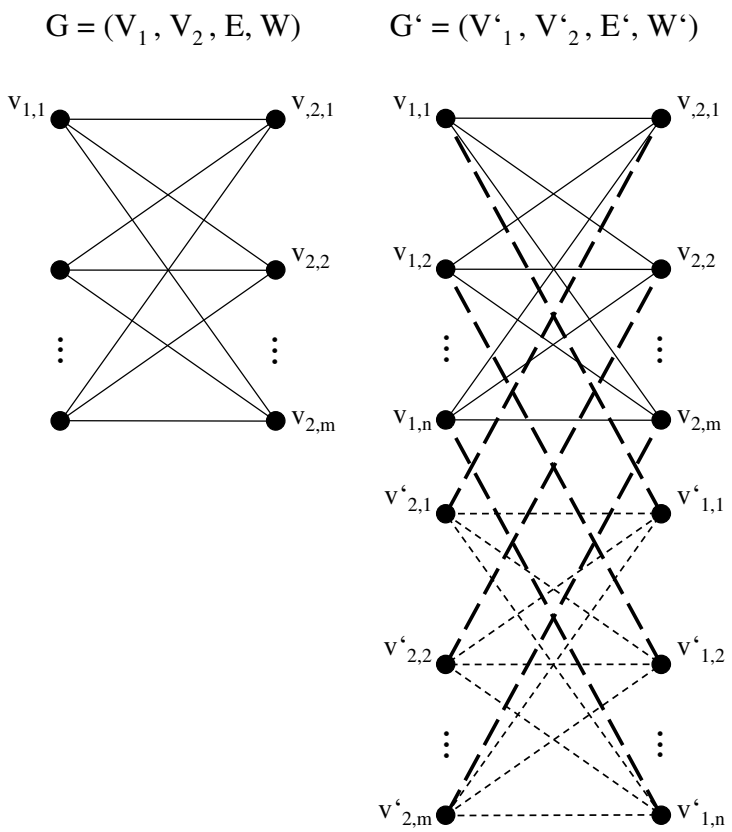

Abbildung 4.23: Verfahren von KHULLER UND RAGHAVACHARI zur Reduktion des Kantenüberdeckungsproblems auf das Problem der vollständigen minimalen Korrespondenz in einem gewichteten bipartiten Graphen

Reduktion des Kantenüberdeckungsproblems:

Die Reduktion des Kantenüberdeckungsproblems auf das Korrespondenzproblem spielt eine wichtige Rolle sowohl in diesem, als auch in dem im Rahmen dieser Arbeit entwickelten Verfahren, so daß es nachfolgend erläutert wird.

[8] **Korrespondenz:** Eine Korrespondenz in einem Graphen G ist eine Menge K von Kanten, von denen keine zwei einen Endpunkt gemeinsam haben [Jun90] (jeder Knoten wird von höchstens einer Kante getroffen). **Gewichtetes Korrespondenzproblem:** Gegeben sei ein Graph $G = (V, E, W)$, wobei $W : E \rightarrow \mathbb{R}$ die Kanten gewichtet. Das gewichtete Korrespondenzproblem besteht darin, eine Menge $E' \subseteq E$ zu finden, so daß $G' = (V, E')$ eine Korrespondenz darstellt und die Summe der Kantengewichte von E' möglichst groß ist.

Um das Kantenüberdeckungsproblem eines gewichteten bipartiten Graphen $G = (V_1, V_2, E, W)$ mit den beiden Knotenmengen V_1 und V_2, der Kantenmenge E und der Wichtungsfunktion $W : E \to \mathbb{R}$ auf das gewichtete Korrespondenzproblem zurückzuführen, wird ein neuer Graph $G' = (V_1', V_2', E', W')$ wie folgt erzeugt (vgl. auch Abb. 4.23). Zunächst enthalten die Knotenmengen V_1' und V_2' exakt die Knoten des ursprünglichen Graphen. Ebenso erhalte die Kantenmenge E' exakt die Kanten aus E mit den selben Gewichten, wie in G (durchgezeichnete Linien in Abbildung 4.23). Zusätzlich erhält die Menge V_1' Kopien aller Knoten aus V_2 und V_2' Kopien aller Knoten aus V_1, wobei alle von V_2 stammenden Kopien je mit allen aus V_1 stammenden Kopien durch Kanten mit dem Gewicht 0 miteinander verbunden werden (die Kopien werden untereinander mit Kanten verbunden, deren Gewicht 0 ist - eng gestrichelte Linien in Abbildung 4.23). Außerdem seien in G' alle originalen Knoten mit ihren Kopien verbunden (Weit gestrichelte Linien in Abbildung 4.23). Das dabei zu verwendende Gewicht entspreche dem Minimum der Kantengewichte der die originalen Knoten im ursprünglichen Graphen treffenden Kanten. Sei mit $E(v)$ die Menge aller den Knoten v treffenden Kanten bezeichnet, dann ist $w'(v_{1,i}, v_{2,i}') = min\{w(e)|e \in E(v_{1,i})\}$ und $w'(v_{2,i}, v_{1,i}') = min\{w(e)|e \in E(v_{2,i})\}$. Alle zur Vollständigkeit des bipartiten Graphen G' fehlenden Kanten werden mit einem Gewicht o eingefügt, wobei o größer ist, als die Summe der Gewichte aller bisherigen Kanten.

Eine zu bevorzugende Kante zeichnet sich hier durch ein geringes Gewicht aus, wohingegen Verfahren zur Lösung des gewichteten Korrespondenzproblems (z.B. der *Ungarische Algorithmus* [Jun90]) diejenigen Kanten auswählen, die ein möglichst hohes Gewicht haben. Aus diesem Grund sind sämtliche Kantengewichte zu invertieren. Dabei ermittelt man zunächst die Summe der Gewichte aller Kanten $\sigma = \sum_{e \in E'} w(e)$ und ermittelt für jede Kante ihr neues Gewicht $w''(e) = \sigma - w'(e)$. Auf den so erhaltenen bipartiten Graphen $G'' = (V_1', V_2', E', W'')$ läßt sich dann ein Verfahren zur Ermittlung einer gewichtsmaximalen Korrespondenz anwenden. Das Ergebnis eines solchen Verfahrens ist eine Teilmenge der ursprünglich in G'' enthaltenen Kanten, mit der sich die gewichtsminimale Kantenüberdeckung $\mathcal{K}$ ermitteln läßt.
Eine genauere Beschreibung dieses Verfahrens findet sich in [Ata99].

Beschriftung der Verbindungen:

Im zweiten Schritt des Verfahrens *MH-Diff* werden die bestehenden Kanten der im ersten Schritt erzeugten gewichtsminimalen Kantenüberdeckung mit den für das Änderungsprotokoll vorgesehenen Beschriftungen versehen. Da die eigene Implementierung nicht direkt auf diesem Teilverfahren basiert, erfolgt hier keine ausführliche Beschreibung. Eine genaue Beschreibung dieses Vorgehens findet sich in [CGM97].

Erzeugung des *edit-scripts*:

Im dritten Schritt wird mit Hilfe der beschrifteten Kanten ein *edit-script* erzeugt. In Abhängigkeit der Beschriftungen der Kanten entstehen aus diesen die Operationen für den *edit-script*. Wie später im Abschnitt 6.5.3 gezeigt wird, ist dabei die Reihenfolge der zu bearbeitenden Kanten von Bedeutung. Da auch in diesem Fall die eigene Implementierung nicht direkt auf das in [CGM97] beschriebene Teilverfahren aufbaut, wird hier nicht näher darauf eingegangen.
Eine Übersicht über die in [CGM97] vorgenommenen Arbeitsschritte findet sich in Abbildung 4.24.

Einschränkungen:

An dieser Stelle wird kurz auf einige Einschränkungen des Verfahrens eingegangen.
Bei dem Vergleich der Bäume wird die Reihenfolge der Kinder aller Knoten für unwichtig betrachtet. Dieses trifft sicherlich für viele Anwendungsfälle zu. Andererseits spielt diese Reihenfolge in vielen anderen Fällen sehr wohl eine Rolle und sollte dann auch von einem vergleichenden Verfahren berücksichtigt werden.

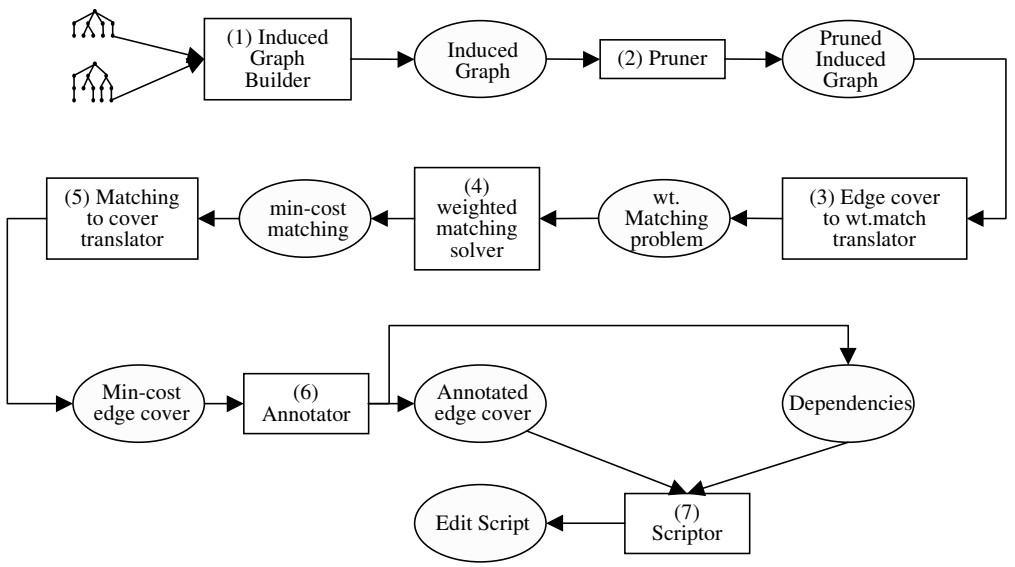

Abbildung 4.24: Vorgehensmodell des in [CGM97] beschriebenen Verfahrens

Die Semantik der *insert*- und *delete*-Operationen, bei denen jeweils ein Knoten direkt und andere Knoten indirekt durch Verschieben betroffen sind, erscheint ungewöhnlich und nicht intuitiv. Wird ein Knoten n in den Baum T so eingefügt, daß er Kind des Knotens $p \in T$ wird und die Kinder $C \subseteq C(p)$ sind anschließend Kinder von n (wie in Abbildung 4.14), dann werden tatsächlich $1 + |C|$ Operationen ausgeführt: eine Operation, die n einfügt gefolgt von $|C|$ Operationen, die einige Kinder von p verschieben.

Die Gleichung 4.1 macht deutlich, daß die untere Grenze der Kosten einer Kante $e = (m; n)$ sehr gering wird, wenn die Kinder der beteiligten Knoten m und n von sehr vielen Kanten getroffen werden, die ein Kind des jeweils anderen Knotens treffen. Umgekehrt macht die Gleichung 4.2 deutlich, daß die obere Grenze der Kosten einer Kante $e = (m; n)$ um so größer wird, je mehr Kinder die beteiligten Knoten m und n haben. Zusätzlich erhöht wird dieser Wert durch die Anzahl der Kanten, die die Kinder von m und n treffen. Dieses führt zu zwei Problemen:

- Die untere Grenze der Kosten einer Kante $c_{min}(e)$ ist in der Regel sehr viel niedriger als die obere Grenze $c_{max}(e)$. Dadurch wird es sehr unwahrscheinlich, daß eine Kante e während der Stutzungsphase entfernt wird. In Testläufen wurden während der Stutzungsphase nur sehr wenige Kanten entfernt. Diese waren alle mit dem *Plus*- oder *Minus*-Knoten verbunden.
- Dadurch, daß die obere Kostengrenze einer Kante $e = (m; n)$ für kinderreiche Knoten m und n sehr hoch wird, ist es wahrscheinlicher, daß diese Kante zugunsten von Kanten entfernt wird, die die beiden Knoten m und n mit gleichbeschrifteten, aber kinderlosen Knoten verbinden (Abbildung 4.25). Aus dieser Zuordnung ergibt sich die Konsequenz, daß die Kinder des Knoten m zu dem Knoten verschoben werden müssen, mit dem n schließlich verbunden ist.

4.4 Überblick der Verfahren

Neben den bereits vorgestellten Verfahren zum Erkennen von Änderungen in Informationsbeständen, gibt es noch eine Reihe weiterer, die sich mit diesem Thema beschäftigen. Interessant sind

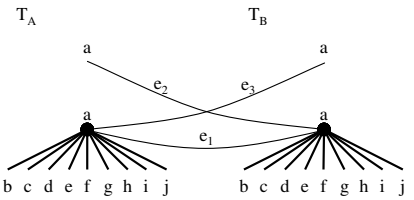

Abbildung 4.25: Die obere Kostengrenze der Kante e_1 wird sehr hoch, da beide beteiligten Knoten sehr viele Kinder haben.

hierbei vor allem diejenigen Verfahren, die hierarchische Strukturen miteinander vergleichen. Von diesen werden hier einige kurz erwähnt.
In [ZS89] und [SZ90] werden Verfahren vorgestellt, die geordnete Bäume miteinander vergleichen, deren Knoten beschriftet sind. Die Differenz zweier Bäume wird hier durch eine sogenannte *edit distance* angegeben. Sie ermittelt sich aus der Länge eines *edit-scripts* und den Kosten der Operationen, aus denen sich der *edit-script* zusammensetzt. In [ZS89] werden den Operationen unterschiedliche und in [SZ90] einheitliche Kosten zugeordnet. Durch die Verwendung einheitlicher Kosten für die Operationen des *edit-scripts* ist das in [SZ90] beschriebene Verfahren deutlich schneller. Betrachtet werden in beiden Verfahren die Operationen *insert, delete* und *update*, die sich jeweils auf einzelne Knoten beziehen.
In [ZWS96] wird der Vergleich ungeordneter Bäume vorgestellt. Es wird bewiesen, daß der Vergleich ungeordneter Bäume NP-hart ist, weshalb sich das dort vorgestellte Verfahren auf ein etwas vereinfachtes Problem bezieht. Die Differenzen werden auch hier mit Hilfe von *edit-scripts* dargestellt, die sich aus den Operationen *insert, delete* und *update* zusammensetzen.

Quelle	**Modell**	**Operationen des *edit-scripts***
[ZS89]	geordnete Bäume	*insert, delete* und *update*
[SZ90]	geordnete Bäume	*insert, delete* und *update*
[CRGMW96]	geordnete Bäume	*insert, delete, update* und *move*
[ZWS96]	ungeordnete B.	*insert, delete* und *update*
[CGM97]	ungeordnete B.	*insert, delete, update, move, copy* und *glue*

Tabelle 4.1: Übersicht der Verfahren zum Vergleich von Bäumen

Die Tabelle 4.1 gibt eine Übersicht der Verfahren wieder, die sich mit dem Vergleich von Bäumen beschäftigen.

Motivation zur Entwicklung eines eigenen Verfahrens

Das Ziel dieser Arbeit ist es, einen Dienst zu entwickeln, der Differenzen zwischen hierarchisch strukturierten Informationsbeständen ermittelt, die - mit Hilfe von XML kodiert - Metadaten darstellen. Von den betrachteten Verfahren erschienen die in [CRGMW96] und [CGM97] beschriebenen am geeignetsten, da sie beschriftete Bäume miteinander vergleichen und jeweils aussagekräftige *edit-scripts* erzeugen. Besonders großen Wert wird hierbei auf die *move*- (von beiden Verfahren unterstützt) und die *copy*-Operation (nur von dem Verfahren *MH-Diff* unterstützt) gelegt. Diese Operationen können zwar auf Folgen von *insert*- und *delete*-Operationen abgebildet werden, ihre Aufnahme in die *edit-scripts* erhöht jedoch deren Aussagekraft (siehe 6.3).

Ähnlich bringen auch angebotene *copy*- und *glue*-Operationen im *edit-script* einen Gewinn an Information, wie in Abschnitt 6.3 beschrieben wird.

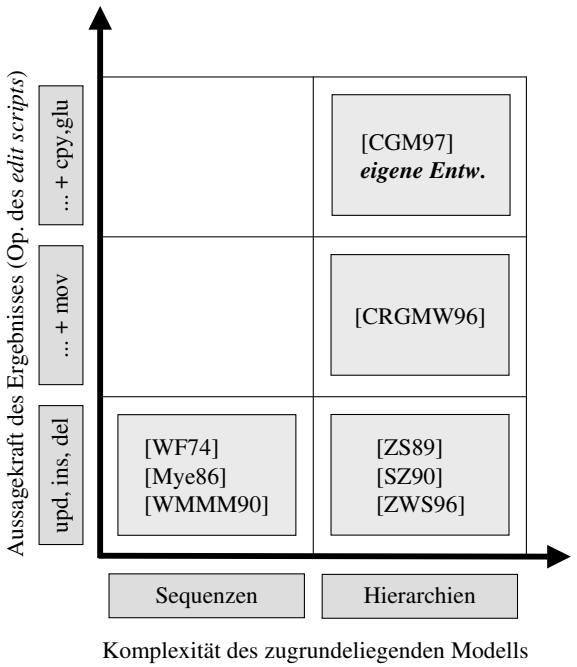

Abbildung 4.26: Grafische Einordnung der betrachteten Verfahren in Bezug auf die Komplexität der Modelle und die Aussagekraft der Ergebnisse

Da die beiden in [CRGMW96] und [CGM97] beschriebenen Verfahren den in den Abschnitten 4.3.1 und 4.3.2 dargestellten Einschränkungen unterliegen, wurde ein eigenes Verfahren entworfen, das im Kapitel 6 beschrieben wird.

Kapitel 5

Verwendete Entwicklungsumgebung und Sprachen

Bevor in den beiden folgenden beiden Kapiteln Algorithmen der eigenen Implementierung sowie einige Aspekte der Realisierung vorgestellt werden, gibt dieses Kapitel einen Überblick über die verwendete Entwicklungsumgebung.

Im ersten Unterabschnitt erfolgt eine kurze Erläuterung, weshalb die Entwicklung in der Sprache *JAVA* durchgeführt wird. Eine zusammenfassende Beschreibung der für die Repräsentation der Informationsbestände verwendeten Metagrammatik XML erfolgt im zweiten Unterabschnitt. Die für den Entwurf verwendete Modellierungssprache *UML* sowie einige ausgewählte Entwurfsmuster werden im dritten und vierten Unterabschnitt vorgestellt.

5.1 JAVA

Bei der Frage nach einer geeigneten Programmiersprache für die Entwicklung des hier beschriebenen Dienstes fiel die Wahl auf die Programmiersprache *JAVA* [Sun95]. Hierfür gibt es eine Reihe von Gründen:

- Die Programmiersprache *JAVA* ist (weitgehend) plattformunabhängig und bietet damit die Möglichkeit, den entwickelten Dienst auf unterschiedlichen Plattformen ablaufen zu lassen, die von *JAVA* unterstützt werden.
- Die Verwendung einer objektorientierten Sprache, bietet für die Entwicklung und mögliche, spätere Erweiterungen und Wartungen große Vorteile im Vergleich zu nicht-objektorientierten Sprachen [RBP+94].
- Sowohl der Übersetzer, als auch die Laufzeitumgebung von *JAVA* sind frei verfügbar.
- Für das *Parsen* von XML-Dokumenten gibt es frei verfügbare Dienste speziell für die Sprache *JAVA* [Edd99a] (wie beispielsweise *XML Parser for JAVA* [IBM98a]), die eine eigene Entwicklung eines *Parsers* überflüssig machen.
- Die Sprache bietet eine Reihe komfortabel anzuwendender Klassenbibliotheken für die Arbeit mit Kollektionen, wie Listen, Mengen und Bäumen, die zur Repräsentation der XML-Dokumente verwendet werden können [Eck98, MS98]. Diese Kollektionsklassen sind in dieser Form allerdings erst ab der Version 2 von *JAVA* verfügbar [Sun00], so daß sowohl an die Entwicklung weiterer Komponenten des Systems, als auch an die Ausführung die Anforderung gestellt wird, diese oder eine aktuellere Version zu verwenden.

Die System-Schnittstellen des hier entworfenen Dienstes beschränken sich auf eine Konfigurationsdatei, die zu vergleichenden XML-Dokumente, ein Ergebnis-XML-Dokument zur Repräsentation der Änderungsfolge sowie Kommandozeilenparameter, womit das System im wesentlichen benutzeroberflächenfrei ist. Dennoch erwies es sich als zwingend erforderlich, eine graphisch orientierte Benutzeroberfläche zur Veranschaulichung der verwendeten Verfahren zu implementieren. Für diese Oberfläche wurde in hohem Maße auf die im *JDK 1.2* enthaltene Klassenbibliothek *SWING* [ELW98] zur Oberflächenprogrammierung zurückgegriffen.

Da die im Rahmen dieser Arbeit entwickelten *JAVA*-Klassen sowohl in Bezug auf ihren Umfang, als auch auf ihre Größe ein beträchtliches Ausmaß erlangt haben, bot es sich an, neben dem rein kommandozeilen-orientierten *JAVA*-Übersetzer eine etwas komfortablere Entwicklungsumgebung einzusetzen. Die Wahl viel hierbei auf *KAWA* [TEK99], ein Produkt der Firma *TEK-TOOLS*. Diese bietet die übliche Projektverwaltung und einen *Debugger* an. Als besonders angenehm wurde hierbei die Tatsache empfunden, daß *KAWA* nicht einen eigenen Übersetzer oder *Debugger* verwendet, sondern gewissermaßen als Adapter zu den jeweiligen Werkzeugen des *JDK* fungiert.

5.2 XML

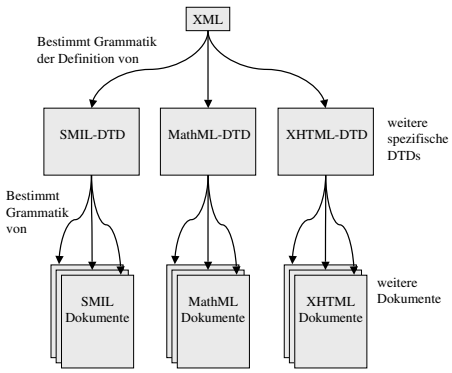

Abbildung 5.1: XML als Metagrammatik für Auszeichnungssprachen

Das World Wide Web Consortium (W3C) erarbeitete mit *XML* (*Extensible Markup Language*) einen Standard, mit dem sich eine Web-Auszeichnungssprache anwendungsspezifisch definieren läßt. XML ist eine standardisierte Sprache zur Notation der Syntax von Auszeichnungssprachen. Somit ist XML eine *Metagrammatik* für kontextfreie Grammatiken - also die Grammatik einer Sprache mit der sich die Regeln von Grammatiken notieren lassen [Tol99] (siehe Abbildung 5.1). XML stellt eine geeignete Möglichkeit dar, hierarchische, semistrukturierte Informationsbestände zu beschreiben. So lassen sich für spezielle Anwendungsgebiete entsprechend spezielle Grammatiken mit XML beschreiben. Die Syntax einer solchen Grammatik wird durch eine für den jeweiligen Anwendungsfall entworfene *Dokumententypdefinition (DTD)* bestimmt. So gibt es beispielsweise eine zur Beschreibung mathematischer Notationen entworfene Auszeichnungssprache *MathML* [IM98]. Ausführliche Beschreibungen von XML finden sich in [Bra99] und [SW99].

5.3 UML

Die *Unified Modelling Language (UML)* ist eine Modellierungssprache. Es handelt sich um eine (überwiegend graphische) Notation, die dazu verwendet wird, die unterschiedlichen Aspekte eines Software-Entwurfs auszudrücken. Eine Modellierungssprache stellt den essentiellen Teil zum Kommunizieren dar [FS97].

Der Entwurf des entwickelten Dienstes läßt sich durch eine Reihe von Diagrammtypen unterstützen. Eingesetzt werden dabei im wesentlichen Paketdiagramme zur Darstellung von Paketen und Klassen und deren Abhängigkeiten, sowie Klassendiagramme zur Darstellung von Typen im System und die verschiedenen Arten statischer Beziehungen zwischen ihnen [Oes97, FS97]. Zur Erfüllung der Anforderungen an den entwickelten Dienst genügt die Form einer Kommandozeilenversion mit nur geringer Interaktion. Da der Dienst jedoch über Komponenten zur Visualisierung der verwendeten Verfahren und zur Konfiguration verfügt, ergeben sich weitere Anwendungsfälle, die im Verlauf des Entwurfs mit Hilfe von Anwendungsfalldiagrammen modelliert werden. Anwendungsfälle sind typische Interaktionen zwischen Benutzern und Computersystemen. Sie werden in Form von Anwendungsfalldiagrammen dargestellt [FS97].

5.4 Entwurfsmuster

Entwurfsmuster bieten einfache und elegante Lösungen für spezifische Probleme des objektorientierten Softwareentwurfs. Sie erfassen Problemlösungen, die im Laufe der Zeit gefunden wurden und sich weiterentwickelt haben. Es sind Entwürfe, auf die man üblicherweise nicht gleich von Anfang an kommt. Sie basieren auf vielen Entwurfsversionen und erneutem Programmieren, die Entwickler auf der Suche nach größerer Wiederverwendung und Flexibilität vorgenommen haben. Entwurfsmuster erfassen diese Lösungen auf konzentrierte und einfach anwendbare Weise [GHJV96].

Im Rahmen der Entwicklungen für diese Arbeit werden einige Entwurfsmuster verwendet. Während durch die Verwendung der Sprache *JAVA* einige Muster beinahe automatisch und ohne eigenes Zutun angewendet wurden (so verwendet man durch die Arbeit mit der Klassenbibliothek *SWING* ganz automatisch zum Beispiel das Beobachter-Muster), entsprang die Verwendung anderer Muster ganz bewußten Entscheidungen (zum Beispiel das Singleton-Muster). Nachfolgend werden einige der benutzten Entwurfsmuster aufgeführt.

Beobachter-Muster: Bei Verwendung des *Beobachter-Musters* (*observer-pattern*) werden mehrere beobachtende Objekte obj_i einem beobachteten Objekt obj^* so zugeordnet, so daß die Objekte obj_i bei Veränderung des Zustands des Objektes obj^* benachrichtigt werden.

Dieses Muster wird bei Verwendung der *SWING*-Klassen beispielsweise zum Auslösen von Aktionen nach Betätigung einer Schaltfläche (Wie zum Beispiel einem *Button*) verwendet. Die Verwendung dieses Musters ergibt sich bei der Oberflächenprogrammierung mit *JAVA* automatisch.

Iterator-Muster: Werden mehrere Objekte als Elemente eines übergeordneten Objektes zusammengefaßt (wie etwa die Elemente einer Menge), bietet das *Iterator-Muster* (*iterator-pattern*) einen sequentiellen Zugriff auf die einzelnen Elemente.

Mit diesem Muster arbeitet man nahezu automatisch, wenn man mit den vom *JDK* angebotenen Kollektionsklassen, wie `List`, `Set` o.ä. arbeitet.

Kompositum-Muster: Zur Erzeugung von Hierarchien, bestehend aus unterschiedlichen Objekten, wird das *Kompositum-Muster* (*composite-pattern*) verwendet. Es ermöglicht eine einheitliche Behandlung der verschiedenen Objekte durch Verwendung einer diesen Objekten gemeinsamen Schnittstelle.

Die Repräsentation eines Baumes, der ein XML-Dokument darstellt, erfolgt durch Verwendung dieses Musters. Die Knoten des Baumes repräsentieren entweder XML-Elemente (Elementknoten) oder Textkörper (Textkörperknoten). Die beiden Klassen zur Repräsentation der Knotentypen sind Subklassen einer Klasse `XNode` und fügen zu dem in dieser Klasse definierten Verhalten individuelles Verhalten hinzu. Dies geschieht durch das Hinzufügen von Parametern, Methoden, aber auch durch das Überschreiben bereits in `XNode` definierter

Methoden. Ein Klient kann alle Knoten des Baumes einheitlich als Objekte der Klasse `XNode` behandeln.

Konkrete Fabrik: Eine *konkrete Fabrik* ist ein Teil des Musters *Abstrakte Fabrik* (*abstract factory*), das eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte bietet, ohne ihre konkreten Klassen zu benennen. Die konkrete Fabrik stellt eine Operation zur Erzeugung von Objekten einer Klassenfamilie zur Verfügung.

Die Knoten der Bäume, die XML-Dokumente darstellen, sind mit Knoten des jeweils anderen Baumes durch Kanten verbunden. Diese Kanten werden durch Objekte repräsentiert, die mit Hilfe einer *Fabrik*-Klasse nach diesem Muster erzeugt werden, da die Verwendung unterschiedlicher Algorithmen die Erzeugung von Kantenobjekten entsprechend unterschiedlicher Klassen erfordert.

Ähnlich werden die verschiedenen Objekte zur Repräsentation der Operationen des *edit-scripts* mit Hilfe einer konkreten Fabrik erzeugt.

Singleton-Muster Das *Singleton-Muster* (*singleton-pattern*) stellt sicher, daß es von einer Klasse nur eine Instanz gibt und bietet einen globalen Zugriff auf diese an.

Die oben erwähnten Fabrik-Klassen sind zusätzlich nach dem *singleton-pattern* entworfen, um zu garantieren, daß in einer Laufzeitumgebung nur eine Fabrik für jeden Zweck existiert.

Kapitel 6

Entwicklung Generischer Dienste für das Change Management

Nachdem in Kapitel 5 die verwendete Entwicklungsumgebung beschrieben wurde, wird sich dieses Kapitel mit der Darstellung eines eigenen Verfahrens beschäftigen, welches den Vergleich semistrukturierter Metadaten ermöglicht. Einige Aspekte der Implementierung dieses Verfahrens werden im Kapitel 7 erläutert.

Im Kapitel 4 wurden einige Verfahren beschrieben, deren Ziel es ist, Informationsbestände miteinander zu vergleichen. Dabei wurde in unterschiedlich hohem Maße Rücksicht auf die Struktur der miteinander zu vergleichenden Informationsbestände gelegt. Aufgrund der abschließend im Abschnitt 4 erwähnten Gründe verwendet das hier entwickelte Verfahren das in [CGM97] beschriebene als Rahmenwerk, realisiert jedoch einzelne Teile aus diesem in anderer Weise.

Der erste Teil dieses Kapitels wird den groben Ablauf dieses Verfahrens kurz erläutern. Es werden darin die verschiedenen zu behandelnden Schritte kurz dargestellt. Im Abschnitt 6.2 werden das Modell und die Annahmen des realisierten Ansatzes beschrieben. Insbesondere wird dort auf die Repräsentation von XML-Dokumenten durch Bäume eingegangen. Die verschiedenen Operationen, aus denen die zu erzeugenden *edit-scripts* bestehen, werden im Abschnitt 6.3 beschrieben. Im Abschnitt 6.4 wird die Repräsentation der von dem Verfahren zu ermittelnden Änderungen und Metadaten als XML-Dokumente beschreiben. Der Abschnitt 6.5 erläutert schließlich die verschiedenen Algorithmen und deren Zusammenspiel.

6.1 Ablauf des Verfahrens

Um für zwei miteinander zu vergleichende Bäume T_A und T_B die Differenz in Form eines *edit-scripts* zu erhalten, wird ähnlich wie in [CGM97] vorgegangen. Der wohl wesentlichste Bestandteil dieser Vorgehensweise ist die Lösung des Problems über eine *gewichtsminimale Kantenüberdeckung* eines gewichteten, bipartiten Graphen. Grob betrachtet, umfaßt die Architektur des Systems folgende Schritte (siehe auch Abbildung 6.1):

1. *Parsen* der XML-Dokumente (aus jedem Dokument wird jeweils ein Baum erstellt)
2. Erstellen des induzierten Graphen (Einfügen von Kanten, die Knoten des einen Baumes mit Knoten des anderen verbinden)
3. Erstellen eines gewichteten Korrespondenzproblems (durch temporäre Erzeugung eines vollständigen gewichteten bipartiten Graphen)

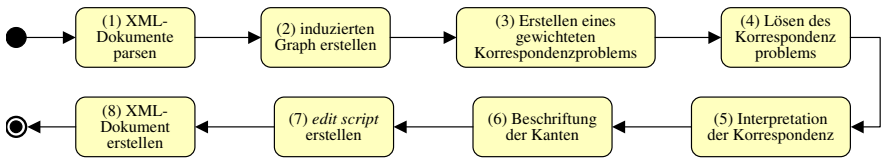

Abbildung 6.1: Vorgehensweise des entwickelten Verfahrens

4. Lösen des Korrespondenzproblems
5. Interpretation der Korrespondenz (Identifikation der verbleibenden Kanten im induzierten Graphen)
6. Beschriften der verbleibenden Kanten
7. Erstellung des *edit-scripts*
8. XML-Dokument zur Repräsentation des *edit-scripts* erstellen

Zunächst werden die beiden miteinander zu vergleichenden XML-Dokumente mit Hilfe eines *Parsers* in Bäume transformiert (Schritt eins).

So wie in [CGM97] wird auch hier ein *induzierter Graph* aus den beiden Bäumen erstellt. Mit dessen Hilfe wird für jeden Knoten m_A des Baumes T_A versucht, mindestens einen Partner-Knoten m_B im Baum T_B zu finden und umgekehrt, so daß sich die in Abschnitt 4.3.2 beschriebene Kantenüberdeckung ergibt. Bei der Ermittlung einer gewichtsminimalen Kantenüberdeckung spielen den Kanten zugeordnete Gewichte eine große Rolle. Während die Gewichte in [CGM97] jedoch die aus den jeweiligen Kanten resultierenden Kosten (beziehungsweise deren unteren und oberen Schranken) repräsentieren, sollen sie hier ein Maß für die *Ähnlichkeit* der durch sie verbundenen Knoten darstellen. Eine gefundene gewichtsminimale Kantenüberdeckung enthält nur noch diejenigen Kanten, die vergleichsweise ähnliche Knoten miteinander verbinden (Schritte zwei bis fünf).

Die in der gewichtsminimalen Kantenüberdeckung verbleibenden Kanten werden mit Beschriftungen versehen, die im wesentlichen den Operationsbezeichnungen des endgültigen *edit-scripts* entsprechen. Diese Beschriftungen sind jedoch nur von temporärer Natur und bereiten die Erstellung des *edit-scripts* vor (Schritt sechs).

Mit Hilfe der beschrifteten Kanten der Kantenüberdeckung kann ein *edit-script* erstellt werden. Dabei entstehen für jede nicht mit *"nil"* beschriftete Kante ein oder zwei Änderungsoperationen im *edit-script*. So führt die Bearbeitung einer mit *"mov"* beschrifteten Kante zu einer *move*-Operation im *edit-script*- eventuell gefolgt von einer *update*-Operation, falls die durch die Kante verbundenen Knoten unterschiedlichen Inhalt haben. Eine mit *"nil"* beschriftete Kante führt zu einer *update*-Operation, falls die durch die Kante verbundenen Knoten unterschiedlichen Inhalt haben. Bei dieser Vorgehensweise ist die Reihenfolge von auszuführenden und damit dem *edit-script* hinzuzufügenden Operationen zu berücksichtigen (Schritt sieben).

Schließlich wird der *edit-script* in Form eines XML-Dokuments als Datei gespeichert, um so anderen Anwendungen die Möglichkeit zu geben, mit den ermittelten Differenzen weiterzuarbeiten (Schritt acht). Die Wahl des XML-Formats führt dabei zu einer hohen Kompatibilität.

6.2 Modell und Annahmen des realisierten Ansatzes

XML-Dokumente repräsentieren aufgrund ihrer Wohlgeformtheit Bäume, deren Knoten die laut DTD definierten Elemente oder Textkörper sind. Die Hierarchie eines solchen Baumes spiegelt die Schachtelung der Elemente im Dokument wieder. Notwendig ist also eine Objektdefinition, mit der einerseits die Baumstruktur selber und generische Operationen darauf, andererseits die spezifischen Knoten einer Auszeichnungssprache mit den jeweiligen Attributen repräsentierbar sind [Tol99]. Daraus ergeben sich folgende, durch die Baumstrukturen zu erfüllende Kriterien:

- Die zu verwendenden Bäume müssen beschriftet sein.
- Der Name und die Attribute von Elementen müssen durch Baumknoten (*Elementknoten*) repräsentiert werden. Die Beschriftung von Knoten, die ein Element darstellen, repräsentiert sowohl den Elementnamen als auch die Attribute des Elements.
- Elementen untergeordnete Elemente und Textkörper müssen durch Kinder des betreffenden Elementknotens repräsentiert werden.
- Textkörper werden immer durch Blätter (*Textkörperknoten*) dargestellt (sie haben demnach keine Kinder).

Durch eine derartige Darstellung der XML-Dokumente durch Bäume ist man in der Lage, das Problem, die Differenz zweier XML-Dokumente A und B in Form eines *edit-scripts* es_{AB} zu erhalten, dadurch zu lösen, daß man die Differenz zweier entsprechend beschrifteter Bäume T_A und T_B ermittelt.

Da XML-Dokumente Informationen semistrukturiert darstellen (die Textkörper sind in sich nicht strukturiert, sondern lediglich eine Folge von Zeichen), ist bei erkannten Unterschieden zwischen zwei Textkörpern gegebenenfalls der Einsatz nachfolgender Verfahren sinnvoll, um einen genaueren Vergleich der betreffenden Textkörper anzustellen.

Wie in Abschnitt 4.3 beschrieben wurde, gibt es sowohl Anwendungsfälle, in denen die Reihenfolge untergeordneter Komponenten eine Rolle spielt, als auch solche, in denen diese Reihenfolge ohne Bedeutung ist. Abhängig von der Bedeutung dieser Reihenfolge werden geordnete oder ungeordnete Bäume betrachtet. Das hier beschriebene Verfahren läßt sich für beide Fälle einsetzen, wobei an einigen Stellen die Unterscheidung entsprechend vorgenommen wird. Der implementierte Dienst läßt sich anwendungsfallspezifisch konfigurieren, ob die zu vergleichenden Bäume als geordnet oder ungeordnet anzusehen sind.

In Abschnitt 4 wurden einige existierende Vorschläge zur Lösung des Problems, die Differenz zweier Bäume zu ermitteln erwähnt. Von diesen eignet sich das in [CGM97] beschriebene Verfahren *MH-Diff* am ehesten für eine Implementierung. Aufgrund der am Ende des Abschnitts 4.3.2 angeführten Gründe kam es zu der Entwicklung eines eigenen Verfahrens, das hier vorgestellt wird. Dabei wurde die grobe Vorgehensweise des Verfahrens *MH-Diff* übernommen, während die Algorithmen der einzelnen Komponenten verändert beziehungsweise ersetzt worden sind.

6.3 Das Operationen-Repertoire

Das im Rahmen dieser Arbeit entworfene Verfahren liefert einen *edit-script*, dessen zugrundeliegendes Operationen-Repertoire mit Ausnahme der *insert*- und *delete*-Operationen, dem in Abschnitt 4.3.2 beschriebenen aus [CGM97] entspricht.

Es werden sechs Typen von Operationen unterschieden, die auf Bäume angewendet werden. Die Operationen zum Einfügen und Entfernen von Knoten bilden das Minimum an Operationen, die

für ein Operationen-Repertoire notwendig sind. Sie arbeiten nicht nur mit einzelnen Knoten, sondern mit Teilbäumen, aufgrund der hierarchischen Struktur. Eine Operation zum Ändern des Inhalts eines Knoten drückt aus, daß geringfügige Änderungen an dessen Inhalt vorgenommen werden. Da die zugrundeliegenden Strukturen hierarchisch sind, erhöhen die beiden Operationen zum Verschieben und Kopieren von Teilbäumen die Ausdruckskraft des resultierenden *edit-scripts*. Eine Operation zum Einschmelzen von Teilbäumen bildet gewissermaßen das Gegenstück zu der Operation zum Kopieren von Teilbäumen. Durch die Wahl dieses Operationen-Repertoires sind die resultierenden *edit-scripts* ausdrucksstärker, als wenn sie sich aus einer weniger mächtigen Menge von Operationen zusammensetzten.

Zum Anwenden eines *edit-scripts* wird eine als Arbeitsbaum bezeichnete Kopie des Basisbaumes erstellt und schrittweise durch die einzelnen Operationen modifiziert. Dabei wird eine Numerierung der Knoten verwendet, die sich auf eine implizit vorzunehmende *Preorder*-Traversierung des Arbeitsbaumes bezieht und nach jeder die Struktur verändernden Operation ungültig wird. Bei den Operationen werden die Knoten über ihre Numerierung identifiziert. Ausnahmen bilden die *insert*- und die *update*-Operation, die neben Zahlen auch komplexere Strukturen als Parameter erhalten. Einige Operationen (*insert, move* und *copy*) fügen einen Teilbaum in eine bestehende Baumstruktur ein, wobei ein Parameter <#targetparent> die Nummer des Knotens angibt, dessen Kind der einzufügende Knoten werden soll. Dieser Knoten wird als *Zielvater* bezeichnet. Er kann bereits vor dem Ausführen der Operation Kinder haben, so daß sich bei der Betrachtung geordneter Bäume die Frage ergibt, an welche Position der einzusetzende Teilbaum eingefügt werden soll. Ein (bei der Betrachtung ungeordneter Bäume wegfallender) Parameter <#childpos> gibt die bei 0 beginnende Position an, die der einzusetzende Knoten nach der Operation haben soll. Fehlt dieser Parameter, wird der einzusetzende Knoten als letzter Knoten der Liste der Kinder des Zielvaters hinzugefügt.

Nachfolgend werden die Operationen des Repertoires im einzelnen beschrieben und jeweils durch eine Abbildung verdeutlicht.

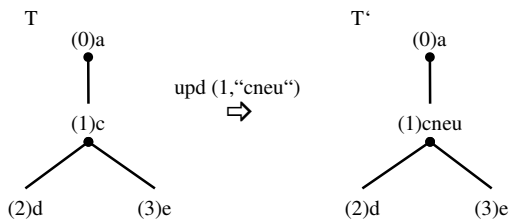

Abbildung 6.2: *update*-Operation

update: Die Operation `upd (#node, newcontent)` verändert den Inhalt eines einzelnen Knoten. Abhängig davon, ob es sich um ein Element- oder ein Textkörperknoten handelt, müssen unterschiedliche Operationen vorgesehen werden.

Parameter: Knotennummer und der neue Inhalt.

Im Beispiel in Abbildung 6.2 wird durch den Befehl `upd (1,"cneu")` dem Knoten mit der Nummer 1 ein neuer Inhalt gegeben.

insert: Die Operation `ins(subtree, #targetparent, [#childpos])` fügt einen neuen Teilbaum (nicht unbedingt nur einen einzelnen Knoten) in den Baum ein.

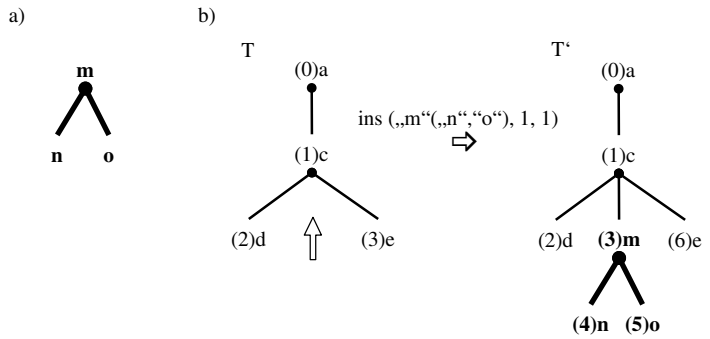

Abbildung 6.3: *insert*-Operation: a) Einzusetzender Teilbaum b) Ausführung der Operation

Parameter: einzufügender Teilbaum, Zielvater, optional: einzunehmende Position des Teilbaumes bezüglich nebengeordneter Knoten.

Im Beispiel in Abbildung 6.3 wird durch den Befehl `ins(”m”(”n”,”o”), 1, 1)` ein neuer Teilbaum eingesetzt, so daß er Kind des Knotens mit der Nummer 1 wird und dort die Position 1 erhält.

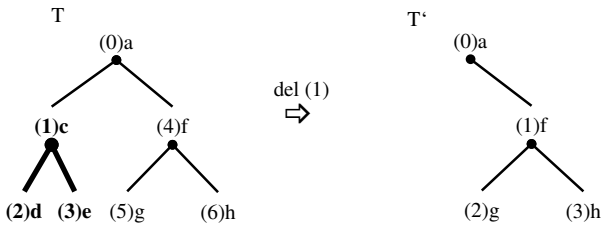

Abbildung 6.4: *delete*-Operation

delete: Die Operation `del(#node)` löscht den Teilbaum, dessen Wurzel die angegebene Nummer trägt. Während die *delete*-Operation im Verfahren *MH-Diff* [CGM97] nur einen einzelnen Knoten löscht (dessen Kinder werden dabei Kinder seines Vaters), wird hier ein ganzer Teilbaum entfernt. Ähnlich, wie die *insert-, move-, copy-* und *glue*-Operation, bezieht sich diese Operation auf Teilbäume, da dieses dem intuitiven Verständnis der Bearbeitung hierarchischer Strukturen entspricht. So bezieht sich beispielsweise das Einfügen, Entfernen, Verschieben oder Kopieren eines Kapitels in einem Text automatisch auf die Komponenten, die dem Kapitel untergeordnet sind (Unterkapitel, Abbildungen, Tabellen, Absätze und so weiter).

Parameter: Nummer des Knotens, der die Wurzel des zu entfernenden Teilbaumes bildet.

Im Beispiel in Abbildung 6.4 wird durch den Befehl `del (1)` der Teilbaum entfernt, der mit dem Knoten mit der Nummer 1 beginnt.

move: Die Operation `mov(#node, #targetparent, [#childpos])` verschiebt einen Teilbaum an eine neue Position im Baum.

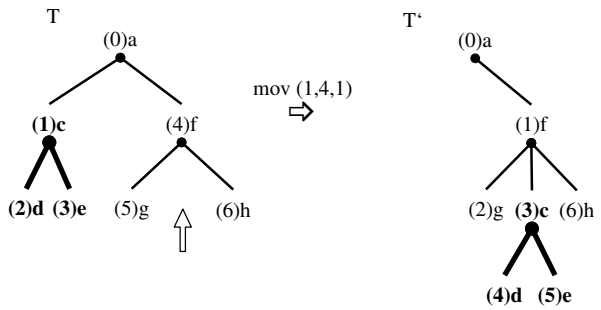

Abbildung 6.5: *move*-Operation

Parameter: Knotennummer der Wurzel des zu verschiebenden Teilbaumes, Knotennummer des Zielvaters und optional die einzunehmende Position des Teilbaumes innerhalb nebengeordneter Knoten.

Im Beispiel in Abbildung 6.5 wird durch den Befehl `mov (1,4,1)` der Teilbaum verschoben, der mit der Nummer 1 beginnt, so daß er Kind an Position 1 des Knotens mit der Nummer 4 wird.

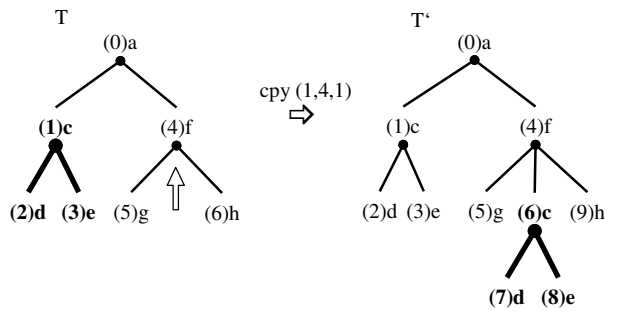

Abbildung 6.6: *copy*-Operation

copy: Die Operation `cpy(#node, #targetparent, [#childpos])` kopiert einen Teilbaum an eine neue Position im Baum. Aus einem Teilbaum $T(n)$ entstehen zwei identische Teilbäume $T(n)$ und $T(n')$, die anschließend beide unabhängig voneinander durch andere Operationen modifiziert werden können, so daß schließlich zwei ähnliche (bzw. identische, wenn es keine weiteren Modifikationen gab) Teilbäume im Zielbaum existieren.

Parameter: Knotennummer der Wurzel des zu kopierenden Teilbaumes, Knotennummer des Zielvaters und optional die einzunehmende Position des Teilbaumes innerhalb nebengeordneter Knoten.

Im Beispiel in Abbildung 6.6 wird durch den Befehl `cpy (1,4,1)` der Teilbaum kopiert, der mit der Nummer 1 beginnt, so daß seine Kopie Kind an Position 1 des Knoten mit der Nummer 4 wird.

glue: Die Operation `glu(#node, #originalnode)` entfernt - ähnlich der *delete*-Operation - einen Teilbaum. Jedoch wird hier davon ausgegangen, daß der entfernte Teilbaum $T(n')$ einem anderen Teilbaum $T(n)$ hinreichend ähnlich ist, um ihn „zu diesem verschmelzen zu lassen“, wobei der Teilbaum $T(n)$ unverändert bleibt. Diese Operation entspricht gewissermaßen einer umgekehrten Ausführung der *copy*-Operation. Der zu entfernende Teilbaum $T(n')$ muß

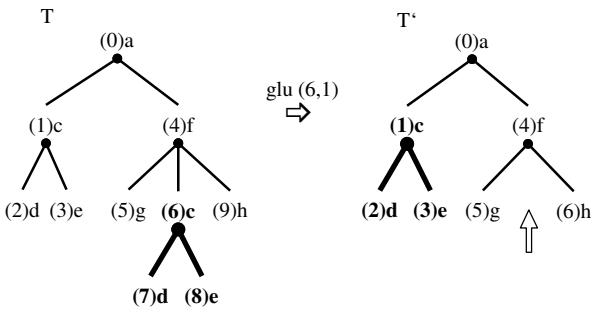

Abbildung 6.7: *glue*-Operation

nicht identisch mit dem Teilbaum $T(n)$ sein - es genügt eine hinreichend große Ähnlichkeit zu diesem, damit sich das Verfahren gegen ein Entfernen von $T(n')$ und für die *glue*-Operation entscheidet.

Obwohl die *glue*-Operation der *delete*-Operation in ihrem Resultat entspricht, gibt es für ihre Existenz einen wichtigen Grund: genau so, wie die Nachricht, „ein Teilbaum wird kopiert" mehr Information enthält, als „ein Teilbaum wird eingefügt", enthält die Nachricht „ein Teilbaum wird zu einem anderen verschmolzen" mehr Information, als die Nachricht „ein Teilbaum wird entfernt".

Parameter: Knotennummer des zu entfernenden Knoten und Knotennummer des anschließend verbleibenden (originalen) Knoten.

Im Beispiel in Abbildung 6.7 wird durch den Befehl `glu (6,1)` der Teilbaum entfernt, der mit dem Knoten mit der Nummer 6 beginnt. Da dieser Teilbaum sehr ähnlich ist, wie der, der mit dem Knoten mit der Nummer 1 beginnt, sagt man, er sei zu diesem *eingeschmolzen*.

Ein *edit-script* ist potentiell umkehrbar. Prinzipiell haben alle Operationen jeweils einen zu ihr inversen Partner: die zur *insert*-Operation inverse Operation ist die *delete*-Operation, die zur *copy*-Operation die *glue*-Operation und die *update*- und die *move*-Operationen sind jeweils zu sich selbst invers. Ein *edit-script* ließe sich demnach rückwärts abarbeitend nutzen, um den Zielbaum schrittweise in den Basisbaum zu modifizieren. Genaugenommen fehlen jedoch allen Operationen dazu nötige Informationen, um aus ihnen die jeweils inverse Operation vollständig zu generieren. Mit ein wenig Aufwand könnten diese Informationen jedoch bereitgestellt werden, wenn beispielsweise eine *update*-Operation nicht nur Informationen für den neuen Inhalt des zu ändernden Knoten, sondern auch die des alten Knoten enthielte. Auf ähnliche Weise müßten alle Operationen erweitert werden. Etwas größeren Aufwand stellt die Erweiterung der *glue*-Operation dar. Unmittelbar vor der Ausführung der *glue*-Operation müssen zwei ähnliche (nicht unbedingt identische) Teilbäume $T(n)$ und $T(n')$ vorliegen, so daß durch die *glue*-Operation der Teilbaum $T(n')$ zu $T(n)$ eingeschmolzen wird. Unmittelbar nach der *copy*-Operation existieren zwei identische Teilbäume. Insofern stellt die *glue*-Operation nicht wirklich die zur *copy*-Operation inverse Operation dar. Um sie zu einer solchen zu machen, müßte ihre Anwendung voraussetzen, daß die Teilbäume $T(n)$ und $T(n')$ identisch sind, was bedeutet, daß unmittelbar vor ihrer Anwendung der Teilbaum $T(n')$ modifiziert wird, bis er $T(n)$ entspricht, um anschließend entfernt zu werden. Diese Vorgehensweise wäre dann jedoch nicht mehr intuitiv nachvollziehbar, weshalb hier darauf verzichtet wird, zu jeder Operation eine zu ihr inverse anzugeben.

Gäbe es für jede Operation eine zu ihr inverse, ließe sich die schrittweise Ausführung des *edit-scripts* in beide Richtungen durchführen, um so auch ohne tatsächlich vorhandene Informationen über Zwischenversionen zwischen zwei Informationsbeständen schrittweise *hin- und herlaufen* zu

können. Damit ließe sich beispielsweise die Modifikation eines Basisbestands zum Zielbestand interaktiv untersuchen.

6.4 Repräsentation relevanter Information

Die Schnittstelle zur Umgebung des hier entwickelten Verfahrens wird einerseits durch die Repräsentation der erkannten Änderungen und andererseits durch die Repräsentation der miteinander zu vergleichenden Informationsbestände beschrieben. Im Abschnitt 6.4.1 wird die hier verwendete Repräsentation der Änderungen durch XML dargestellt und im darauf folgenden Abschnitt 6.4.2 wird kurz auf die Repräsentation der miteinander zu vergleichenden Metadaten durch XML eingegangen.

6.4.1 Repräsentation der Änderungen

Da der im Rahmen dieser Arbeit entwickelte Dienst Bestandteil anderer Systeme sein soll, ist ein Austausch der *edit-scripts* notwendig. Um diesen zu gewährleisten, ist es wichtig, ein einheitliches, bestimmten Kriterien genügendes Format für die Repräsentation als Schnittstelle zu entwerfen.

- Es sollte die Repräsentation von *edit-scripts* ermöglichen, die in ihrer Anwendung auf ein XML-Dokument ein deterministisches Verhalten zeigen. Um dieses zu gewährleisten, muß die Repräsentation formal genug sein.
- Dennoch ist eine einfache, für menschliche Anwender intuitiv verständliche Repräsentation wichtig, um die Entwicklung weiterer sie nutzenden Dienste zu erleichtern.
- Die Erweiterbarkeit des Formats spielt eine wichtige Rolle, um beispielsweise zukünftigen Anwendungen die Möglichkeit zu geben, das Operationen-Repertoire zu erweitern.
- Schließlich sollte die Anwendung des Formats nicht auf einzelne Systeme beschränkt sein.

Ähnlich wie bei der Wahl des Formats der dem Vergleich zugrundeliegenden Repräsentationen, stellt auch hier die Anwendung von XML eine geeignete Lösung dar, da sich mit dieser Metasprache ein Format generieren läßt, das den oben genannten Kriterien genügt. Die einzelnen *edit-scripts* werden durch XML-Dokumente repräsentiert, die in ihrer Struktur einem Format, nämlich einer dafür entworfenen *Document-Type-Definition (DTD)* genügen. Damit eignet sich diese Form nicht ausschließlich für die Darstellung von *edit-scripts* für XML-Dokumente, sondern ganz allgemein für die geordneter oder ungeordneter Bäume.

Die dafür entworfene DTD sieht zunächst die optionale Angabe einiger Metadaten vor. So können für das Datum, die Uhrzeit und die Dateinamen der für den Vergleich vorgesehenen Dokumente sowie für einen Kommentar jeweils XML-Elemente eingesetzt werden. Abgesehen von diesen Angaben definiert die DTD den *edit-script* als eine beliebige Wiederholung der Auswahl einer der Operationen *upd*, *ins*, *del*, *mov*, *cpy* und *glu*:

```
<!ELEMENT es (date?, time?, fnames?, comment?
              (upd|ins|del|mov|cpy|glu)*)>
```

Die *Document Type Definition* wird nachfolgend ausschnittweise dargestellt. Parallel dazu werden Beispiele der Repräsentation durch XML-Dokumente einzelner Operationen des *edit-scripts* aufgeführt[1].

[1] Eine zusammenhängende Darstellung der *Document Type Definition*, sowie der hier angegebenen Beispiele findet sich im Anhang B.

Die *update-Operation* wird in der DTD definiert als:

```
<!ELEMENT upd (charcontent|attribute*)>
<!ATTLIST upd
    nodenr CDATA #REQUIRED>

<!ELEMENT charcontent (#PCDATA)>

<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
    attname  CDATA #REQUIRED
    attvalue CDATA #REQUIRED
    atttype  CDATA #REQUIRED>
```

Handelt es sich bei dem zu ändernden Objekt um einen Textkörperknoten, enthält das Element `upd` den neuen Inhalt des Textkörperknoten. Handelt es sich hingegen um einen Elementknoten, enthält das Element `upd` die Liste der Attribute, die der Elementknoten nach Ausführung der Operation haben soll. Diese Attributliste wird durch beliebige Wiederholung des Elements `attribute` erreicht, das selbst leer ist, aber drei Attribute besitzt, die gemeinsam ein Attribut definieren. Die Darstellung als XML-Dokument sieht dann folgendermaßen aus:

```
<?xml version="1.0"?>
<!DOCTYPE es SYSTEM "es.dtd">

<es>
  <upd nodenr="1">
    <attribute attname="n" attvalue="neu" atttype="CDATA" />
  </upd>
</es>
```

In der *Document Type Definition* wird die *insert*-Operation definiert als:

```
<!ELEMENT ins (node)>
<!ATTLIST ins
    targetparent CDATA #REQUIRED
    childpos     CDATA #IMPLIED>

<!ELEMENT node (elnode|data)>

<!ELEMENT elnode (attribute*, (node*))>
<!ATTLIST elnode
    name CDATA #REQUIRED>

<!ELEMENT data (#PCDATA)>
```

Das Element `ins` hat zwei Attribute: eines gibt die Nummer des Knotens an, der der Vater des einzusetzenden Teilbaumes werden soll. Spielt die Reihenfolge der Kinder dieses Zielvaters eine Rolle, wird unter Verwendung des zweiten Attributs dieses Elements die Position bezüglich der Kinder des Zielvaters beschrieben. Durch die Definition der Elemente `node, elnode` und `data` lassen sich Teilbäume definieren, wobei ein Element `elnode` für einen Elementknoten und das Element `data` für einen Textkörperknoten steht. Das in Abbildung 6.3 beschriebene Einfügen eines Teilbaumes läßt sich mit XML dann folgendermaßen darstellen:

```
<?xml version="1.0"?> <!DOCTYPE es SYSTEM "es.dtd">

<es>
  <ins targetparent="1" childpos="1">
    <node>
      <elnode name="D">
        <attribute attname="n" attvalue="m" atttype="CDATA" />
        <node>
          <elnode name="D">
            <attribute attname="n" attvalue="n" atttype="CDATA" />
          </elnode>
        </node>
        <node>
          <elnode name="D">
            <attribute attname="n" attvalue="o" atttype="CDATA" />
          </elnode>
        </node>
      </elnode>
    </node>
  </ins>
</es>
```

Da die Syntax für die Repräsentation der übrigen Operationen sehr ähnlich ist, wird hier auf eine weitere Darstellung verzichtet und auf Anhang B verwiesen.

6.4.2 Repräsentation der Metadaten

Im Abschnitt 5.2 wurden einige Eigenschaften von XML erwähnt. Diese führten dazu, XML als Grundlage für die zu vergleichenden Informationsbestände, also der Metadaten zu wählen. Dadurch wird das in Abschnitt 2.3 geforderte Kriterium der Generizität erfüllt, denn es ist zu erwarten, daß es für viele Dokumentenformate Konvertierungsmöglichkeiten von und nach XML oder für die entsprechenden Anwendungen geeignete Exportfilter geben wird.
Ein weiterer Grund für die Entscheidung für XML, ist die Existenz verfügbarer *Parser* für *JAVA*. Aus einer mittlerweile recht umfangreichen Fülle potentiell verwendbarer *Parser* [Edd99a] fiel die Wahl auf den von IBM-ALPHAWORKS angebotenen *XML Parser for JAVA*. Dieses ist ein in *JAVA* entwickelter validierender *XML-Parser*. Mit den zu diesem *Parser* gehörenden Klassen lassen sich XML-Dokumente erzeugen, modifizieren und validieren [IBM98a].
Es ist nicht einfach, mit XML (ungeordnete) Mengen von Elementen darzustellen. Soll beispielsweise ausgedrückt werden, daß einem Element `X` die drei Elemente `A,B,C` in beliebiger Reihenfolge nachgeordnet sind, so sind hierfür sechs (= 3!) Terme in der DTD notwendig:

```
<!ELEMENT X ((A,B,C) | (A,C,B) |
             (B,A,C) | (B,C,A) |
             (C,A,B) | (C,B,A))>
```

Zur Darstellung semistrukturierter Daten, bei denen die Reihenfolge aller nachgeordneten Elemente keine Rolle spielen, läßt sich dieses in XML durch ein Attribut des übergeordneten Elements festlegen. Ist hingegen die Reihenfolge nur einiger nachgeordneter Elemente ohne Bedeutung, läßt sich dies nicht mehr darstellen.
Aus diesem Grunde ist XML für die Darstellung semistrukturierter Daten, bei denen die Reihenfolge nachgeordneter Elemente keine Rolle spielt, nur bedingt geeignet.

6.5 Verwendete Algorithmen

Ein wesentlicher Bestandteil dieses Verfahrens besteht in der Bildung einer gewichtsminimalen Kantenüberdeckung des induzierten Graphen, dessen Kantengewichtung wiederum von zentraler Bedeutung ist (Schritt zwei in Abbildung 6.1). Wie bereits angedeutet, stellt das Gewicht einer Kante ein Maß für die *Ähnlichkeit* der durch sie verbundenen Knoten dar. Die Kante zweier einander sehr ähnlichen Knoten erhält ein niedrigeres Gewicht als die zweier einander nicht so ähnlichen Knoten. Mit Hilfe eines induzierten Graphen mit Kanten, deren Gewichte die Ähnlichkeit der durch sie verbundenen Knoten repräsentieren, läßt sich eine gewichtsminimale Kantenüberdeckung erzeugen, die zu einem kompakten *edit-script* führt.

Im ersten Teil dieses Unterkapitels wird beschrieben, wie der hier verwendete induzierte Graph erstellt wird und vor allem, wie sich die genannten Kantengewichte berechnen.

Bei Erhalt einer gewichtsminimalen Kantenüberdeckung müssen die verbleibenden Kanten so beschriftet werden, daß sich daraus ein *edit-script* bilden läßt. Der zweite Teil dieses Unterkapitels wird sich mit dieser Beschriftung beschäftigen.

Im dritten Teil wird erläutert, wie sich aus einer beschrifteten, gewichtsminimalen Kantenüberdeckung ein *edit-script* generieren läßt.

Der letzte Teil liefert schließlich eine grobe Zeitabschätzung des verwendeten Verfahren.

6.5.1 Erzeugung des induzierten Graphen und Bildung der Kantengewichte

Bei der Erzeugung des induzierten Graphen wird wie folgt vorgegangen: es wird ein bipartiter Graph $G_{Bip} = (V_A, V_B, E, W)$ erzeugt, dessen Knotenmengen V_A aus den Knoten des Baumes T_A und V_B aus denen des Baumes T_B bestehen. Ferner wird der Menge V_A ein spezieller mit *plus* bezeichneter (*Plus*-Knoten) und der Menge V_B ein spezieller mit *minus* bezeichneter Knoten (*Minus*-Knoten) hinzugefügt. Die Knoten aus V_A werden mit Knoten aus V_B verbunden. Eine Kante, die den *Plus*-Knoten mit dem *Minus*-Knoten verbindet, wird *Plus-Minus-Kante* genannt. Die den *Plus*- beziehungsweise *Minus*-Knoten treffenden Kanten, werden *Plus-Kanten* beziehungsweise *Minus-Kanten* genannt. Eine in der gewichtsminimalen Kantenüberdeckung verbleibende Kante $e = (m, n)$ läßt sich so interpretieren, daß der Knoten n aus dem Knoten m hervorgegangen ist (siehe auch Abschnitt 4.3.2).

Da es sich hier letztendlich um den Vergleich von zwei XML-Dokumenten handelt, repräsentiert ein Knoten ein bestimmtes Element oder einen Textkörper eines XML-Dokuments. Handelt es sich um ein Element, sind dessen Name und Attribute bekannt. Den beiden miteinander zu vergleichenden XML-Dokumenten liegt eine gemeinsame DTD zugrunde. Die Bedeutung dieser DTD ist jedoch nicht bekannt, so daß sich keine Aussage darüber machen läßt, wie ähnlich zwei nicht gleichnamige Elementtypen sind. Daher wird hier davon ausgegangen, daß zwei Elemente mit unterschiedlichen Elementnamen so unterschiedlich sind, daß die sie repräsentierenden Knoten nicht durch eine Kante miteinander verbunden werden. Entsprechendes gilt für Kanten zwischen Elementknoten und Textkörperknoten: auch sie werden nicht eingefügt. In einer Erweiterung des Verfahrens ließe sich die Bedeutung dieser Dokument-Typ-Definition so einbringen, daß Elemente ähnlichen Typs dennoch vergleichbar sind und somit durch eine Kante miteinander verbunden werden.

Das Gewicht der Kante e einer Kantenüberdeckung $\mathcal{K}$ ($e = (m, n) \in \mathcal{K}$) wird hier als Maß für die Abweichung der durch sie verbundenen Knoten m und n aufgefaßt. Die Kante zweier einander sehr ähnlichen Knoten erhält ein niedrigeres Gewicht als die zweier einander nicht so ähnlichen Knoten. Dieses Kantengewicht setzt sich aus vier Grundkomponenten (`con`, `chi`, `env` und `sib`)

zusammen, die Werte des Intervalls [0..1] sind. Für eine Kante $e = (m; n)$ repräsentiert der Wert `con` die Abweichung der beiden Knoten m und n in Bezug auf ihren Inhalt (*content*), der Wert `chi` die Abweichung der beiden Knoten in Bezug auf ihre Kinder (*children*), der Wert `env` die Abweichung der beiden Knoten m und n in Bezug auf ihre Umgebung (*environment*) und der Wert `sib` die Abweichung der beiden Knoten in Bezug auf die Reihenfolge bezüglich der Kinder ihrer Väter. Jeweils zwei dieser Komponenten ((`con` und `chi`) bzw. (`env` und `sib`)) werden unter Berücksichtigung einer Gewichtung zu einer vermengt (`conchi` und `envsib`). Die beiden Ergebnisse werden wiederum zu einem Wert `conchienvsib` kombiniert, aus dem dann das endgültige Kantengewicht errechnet wird (siehe Abbildungen 6.8 und 6.9).

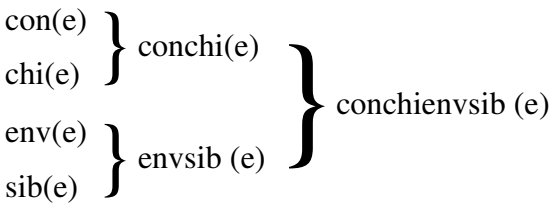

Abbildung 6.8: Bildung des Gewichts einer Kante e

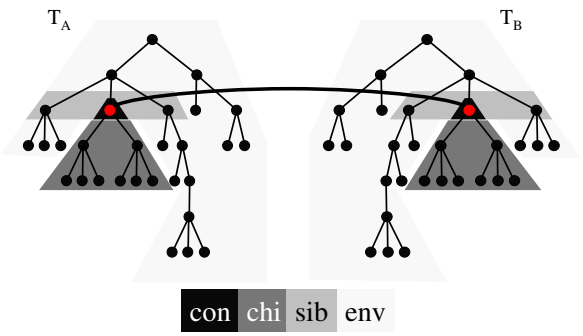

Abbildung 6.9: Bedeutung der Grundgewichte `con`, `chi`, `sib` und `env`

Die Ermittlung der Grundgewichte verläuft in zwei Phasen. In der ersten Phase wird der Baum T_A *postorder* durchlaufen, wobei für alle Kanten der besuchten Knoten die Inhalts-, Nachfolge- und Ordnungsgewichte (`con`, `chi`, `sib`) ermittelt werden. Bei dem Besuch eines Knotens sind damit die drei Werte aller Kanten seiner Kinder bereits ermittelt. In einer zweiten Phase wird der Baum T_A *preorder* durchlaufen, wobei für alle Kanten der besuchten Knoten das Umgebungsgewicht (`env`) ermittelt wird.

Im Folgenden werden die vier Grundwerte `con`, `chi`, `env` und `sib` genauer beschrieben. Mit Ausnahme der *Plus-* und *Minus*-Kanten, haben alle Kanten des induzierten Graphen diese vier Grundwerte, weshalb sie wie eine von einer Kante e abhähngige Funktion $con(e)$, $chi(e)$, $env(e)$ und $sib(e)$ geschrieben werden.

Das Inhaltsgewicht `con`:

Der Wert $con(e)$ einer Kante $e = (m; n)$ repräsentiert die Abweichung der beiden Knoten m und n in Bezug nur auf ihren Inhalt (Abbildung 6.10). Dabei werden gewissermaßen die Beschriftungen der zu vergleichenden Knoten betrachtet. Bei Textkörperknoten gehen die Inhalte der betreffenden Textkörper (Abbildung 6.11) und bei Elementknoten die Attribute der betreffenden Elemente (Abbildung 6.12) in diesen Vergleich ein.

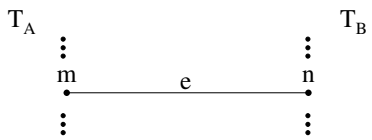

Abbildung 6.10: Bildung von $con(e)$ einer Kante $e = (m; n)$: Der Wert beruht lediglich auf den Inhalten der beiden verbundenen Knoten.

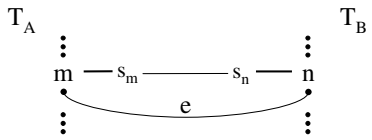

Abbildung 6.11: Bildung von $con(e)$ einer Kante $e = (m; n)$: Wenn m und n Textkörperknoten sind, müssen deren Inhalte s_m und s_n berücksichtigt werden.

Der Vergleich zweier Textkörperknoten ist verhältnismäßig einfach: mit Hilfe eines Verfahrens zur Analyse der Differenz zweier Zeichenketten (siehe 4.2) läßt sich die Abweichung der beiden Textkörper ermitteln und durch einen Wert innerhalb des Intervalls [0..1] darstellen, indem man beispielsweise die Länge der längsten gemeinsamen Untersequenz in Relation zur Länge der längeren der beiden Zeichenketten stellt. Denkbar ist auch die Anwendung von Verfahren, die die Ähnlichkeit (beziehungsweise Abweichung) zweier Texte in Abhängigkeit verschiedener Sprachspezifika ermitteln [ZD96].

Handelt es sich bei den Knoten m und n um Element-Knoten, müssen die Mengen der ihnen zugeordneten Attribute miteinander verglichen werden. Da zwei Elementknoten nur dann durch eine Kante miteinander verbunden sind, wenn ihre Elementnamen gleich sind, werden die Elementnamen für die Gewichtung der Kanten nicht berücksichtigt. Bei der Berücksichtigung der Attribute sind zwei Verfahren anwendbar:

digitales Verfahren: Es wird bestimmt, ob zwei Attribute zueinander äquivalent sind oder nicht. Dabei werden die unten aufgeführten Punkte berücksichtigt. Für das Knotenpaar m und n wird die Anzahl der äquivalenten Attribute der betroffenen Elemente in Relation zur Gesamtzahl der Attribute beider Elemente gesetzt.

analoges Verfahren: Für zwei Attribute läßt sich ihre Ähnlichkeit durch einen Wert des Intervalls [0..1] ausdrücken, wobei ein geringer Wert wieder eine hohe Ähnlichkeit ausdrückt. Man denke sich einen gewichteten bipartiten Graphen $G_{bip} = (V_m, V_n, E, W)$, dessen Knotenmenge V_m die Menge der Attribute von m und dessen Knotenmenge V_n die Menge der Attribute von n darstellt (siehe Abbildung 6.12). Mit der Kantenmenge E und einer Wichtungsfunktion $W : E \rightarrow [0..1]$ werden die Ähnlichkeiten der jeweiligen Attribute ausgedrückt. Durch Vervollständigung dieses bipartiten Graphen erhält man ein gewichtetes Korrespondenzproblem, das sich leicht lösen läßt [Jun90].

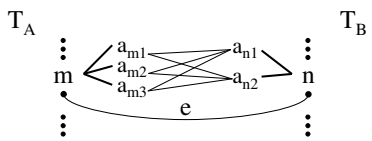

Abbildung 6.12: Bildung von $con(e)$ einer Kante $e = (m; n)$: Wenn m und n XML-Elemente repräsentieren, müssen die Attribute der Elementknoten berücksichtigt werden.

Zur Zeit stehen beide Verfahren für den entwickelten Dienst zur Verfügung und lassen sich wahlweise für den Vergleich einsetzen.

Zur Klärung der Frage, ob zwei Attribute zueinander äquivalent sind (bzw. um ihre Abweichung zu ermitteln), werden die drei Bestandteile Attributname, -typ und -inhalt der beiden Attribute miteinander verglichen. Zusätzliche Angaben, die zu den Attributen gehören, wie etwa das *Flag „REQUIRED“* oder *„IMPLIED“* werden von dem Verfahren nicht berücksichtigt.

Zur Zeit wird die Namens- und Typäquivalenz (ähnlich der geforderten Äquivalenz der Elementtypen für die Bestimmung der Ähnlichkeit zwischen Elementknoten) für weitere Äquivalenz- bzw. Ähnlichkeitsuntersuchungen vorausgesetzt. Die Ermittlung der Äquivalenz der Attributinhalte (bzw. deren Abweichung) kann die Groß- und Kleinschreibung berücksichtigen oder ignorieren. Sie ließe sich beispielsweise elementtyp- und attributtypspezifisch konfigurieren.

Ähnlich, wie bei der Ermittlung der Abweichung von Textkörperknoten, ist auch hier die Anwendung von Verfahren, die die Ähnlichkeit (beziehungsweise Abweichung) zweier Texte in Abhängigkeit verschiedener Sprachspezifika ermitteln, denkbar.

Zwei Attribute lassen sich als äquivalent bezeichnen (wie es für das digitale Verfahren benötigt wird), wenn sie sich hinreichend ähnlich sind, also wenn ihre Abweichung einen bestimmten Wert unterschreitet. Dieses ist im Verfahren genau dann der Fall, wenn die drei Bestandteile Attributname, -typ und -inhalt genau gleich sind.

Durch die Verwendung eines der beiden Verfahren zur Ermittlung einer Zuordnung der Attribute, läßt sich bei Elementknoten ihre Ähnlichkeit in Bezug auf ihren Inhalt bestimmen.

Der Wert $con(e)$ ist somit lediglich von den Inhalten der beiden verbundenen Knoten abhängig, nicht von zuvor eventuell berechneten Werten für die Väter oder Kinder der beteiligten Knoten m und n.

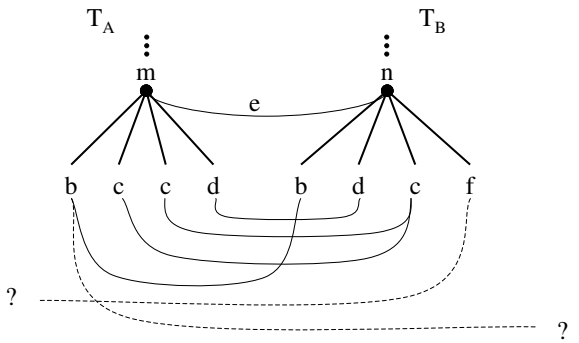

Abbildung 6.13: Bildung von $chi(e)$ einer Kante $e = (m; n)$: Darstellung der Knoten m und n und deren Kinder als Teilbäume.

Das Nachfolgergewicht `chi`:

Der Wert $chi(e)$ stellt die Abweichung der beiden Knoten m und n nur in Bezug auf ihre Kinder dar. Hat genau einer der beiden Knoten m oder n keine Kinder, so läßt sich der Wert $chi(e)$ mit dem Maximalwert $1,0$ angeben. Haben beide Knoten m und n keine Kinder, so läßt sich der Wert $chi(e)$ mit dem Minimalwert $0,0$ angeben. Haben beide Knoten m und n Kinder, so wird der Wert $chi(e)$ wie folgt errechnet:

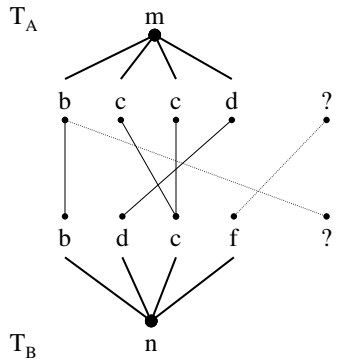

Abbildung 6.14: Bildung von $chi(e)$ einer Kante $e = (m; n)$: Darstellung der Knoten m und n und deren Kinder als bipartiter Graph.

Für die Bestimmung dieses Wertes wird vorausgesetzt, daß für alle Kanten e aller Kinder von m ($C(m)$) der jeweilige Wert $conchi(e)$ bereits bestimmt wurde. Die Knoten $C(m)$ werden den Knoten $C(n)$ in einem bipartiten Graphen einander gegenübergestellt. Von Interesse sind diejenigen Kanten $e \in E_{Cm-Cn}$, die Knoten aus $C(m)$ mit Knoten aus $C(n)$ verbinden. Alle übrigen Kanten sind für die Berechnung von $chi(e)$ ohne Bedeutung. Die Abbildung 6.13 zeigt einen entsprechenden Ausschnitt aus zwei Bäumen mit den Knoten m und n und deren jeweiligen Kinder. Die Kanten aus E_{Cm-Cn} sind durchgezeichnet dargestellt, während die übrigen gestrichelt gezeichnet sind. Abbildung 6.14 verdeutlicht die Situation: hier stehen sich die jeweiligen Kinder der Knoten m und n einander gegenüber. Abhängig davon, ob die Reihenfolge der Kinder der Knoten m und n von Bedeutung ist, ergeben sich zwei etwas unterschiedliche Problemstellungen:

Beachtung der Reihenfolge: Die Knotenansammlungen $C(m)$ und $C(n)$ im bipartiten Graphen werden als geordnet betrachtet. Es wird versucht, eine Teilmenge $E'_{Cm-Cn} \subseteq E_{Cm-Cn}$ zu finden, die die bestmögliche (kreuzungsfreie) Zuordnung der Kinder der Knoten m und n darstellt. Dabei bezieht sich „bestmöglich“ auf eine möglichst geringe Summe der `conchi`-Werte der benutzten Kanten. Es ergeben sich somit folgende Anforderungen für die gesuchte Kantenmenge E'_{Cm-Cn}:

- Möglichst viele Knoten aus $C(n)$ und $C(m)$ werden von einer Kante aus E'_{Cm-Cn} getroffen.
- Kein Knoten aus $C(n)$ und $C(m)$ wird von mehr als einer Kante aus E'_{Cm-Cn} getroffen.
- Keine zwei Kanten aus E'_{Cm-Cn} überkreuzen sich.
- Die Summe $\sum_{e \in E'_{Cm-Cn}} conchi(e)$ ist minimal.

Dieses Problem läßt sich auf das in [MW98] beschriebene zurückführen und lösen. Eine hier entwickelte - nicht optimale - eigene Lösung wird im Anhang unter A beschrieben. Eine Implementierung des Verfahrens aus [MW98] würde den Rahmen dieser Arbeit sprengen und konnte daher nicht vorgenommen werden.

Ignorieren der Reihenfolge: es wird versucht, eine Teilmenge $E'_{Cm-Cn} \subseteq E_{Cm-Cn}$ zu finden, die auch hier die bestmögliche Zuordnung der Kinder der Knoten m und n darstellt. Im Gegensatz zum oben genannten Fall der Beachtung der Reihenfolge, sind Überkreuzungen der Kanten hier erlaubt, so daß dieser Punkt bei den Anforderungen fehlt:

- Möglichst viele Knoten aus $C(n)$ und $C(m)$ werden von einer Kante aus E'_{Cm-Cn} getroffen.
- Kein Knoten aus $C(n)$ und $C(m)$ wird von mehr als einer Kante aus E'_{Cm-Cn} getroffen.

- Die Summe $\sum_{e \in E'_{Cm-Cn}} conchi(e)$ ist minimal.

Dieses Problem ist das Korrespondenzproblem eines gewichteten bipartiten Graphen und kann beispielsweise mit dem *Ungarischen Algorithmus* mit einer Zeitkomplexität von $O(n^3)$ gelöst werden [Jun90].

Das Kombinationsgewicht `conchi`:

Unter Berücksichtigung einer Kombinationsgewichtung werden die beiden Werte $con(e)$ und $chi(e)$ einer Kante e zu einem Wert $conchi(e)$ zusammengefaßt. Durch geeignete Wahl dieser Kombinationsgewichtung ist man in der Lage, die Ähnlichkeit des Inhalts zweier Knoten für wichtiger zu erachten, als die Ähnlichkeit der beiden Knoten bezüglich ihrer Kinder oder umgekehrt. Es hat sich gezeigt, daß Gewichtungen, bei denen der Ähnlichkeit der Inhalte eine größere Bedeutung beigemessen wird, als der der Kinder (etwa im Verhältnis $60 : 40$) gute Zuordnungen liefern. Bei diesem Wichtungsverhältnis wird der Wert $conchi(e)$ wie folgt berechnet:

$$conchi(e) = \frac{60 \times con(e) + 40 \times chi(e)}{100}.$$

Der Wert dieser Kombinationsgewichtung läßt sich, wie die der anderen Kombinationsgewichte `envsib` und `conchienvsib` im implementierten Dienst konfigurieren (siehe Abschnitt 7.8).

Durch die *postorder*-Traversierung des Basisbaumes und Bestimmung der Inhalts- und Nachfolgergewichte und deren Kombination für alle Kanten $(m; n)$ der besuchten Knoten m, ist die Voraussetzung, daß der Wert $conchi(e)$ für alle Kanten e aller Kinder von m bereits bestimmt wurde, erfüllt. Es kann sein, daß der Wert $conchi(e)$ für Kanten der Kinder von n möglicherweise zum Zeitpunkt der Bestimmung des Inhalts- und Nachfolgergewichts der Kante $(m; n)$ noch nicht bestimmt wurde. Dieses ist jedoch ohne Bedeutung, da für die Bestimmung des Nachfolgergewichts der Kante e nur diejenigen Kanten wichtig sind, die zwischen den Knoten $C(m)$ und $C(n)$ liegen. Für diese ist der Wert $conchi(e)$ bereits berechnet, da der Baum T_A *postorder* durchlaufen wird.

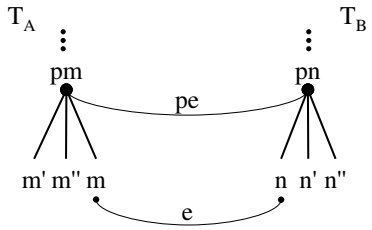

Abbildung 6.15: Bildung von $env(e)$ einer Kante $e = (m; n)$: Der Wert $env(e)$ wird bestimmt durch den Wert $conchienvsib(pe)$.

Das Umgebungsgewicht `env`:

Der Wert $env(e)$ soll die Abweichung der beiden Knoten m und n in Bezug auf ihre nach oben und zur Seite gerichteten Umgebung repräsentieren. Diese Umgebung eines Knotens bildet in erster Linie dessen Vater, dann Geschwister, Großvater und so weiter. Demnach entspricht der Wert $env(e)$ dem Gesamtgewicht der Väterverbindung, also der Kante $(pm; pn)$ zwischen den Vätern pm und pn von m und n (Abbildung 6.15), falls diese existiert. Handelt es sich bei den Knoten m und m jeweils um die Wurzel des jeweiligen Baumes, so wird der Wert auf das Minimum $(0, 0)$ gesetzt, da die Umgebung übereinstimmt. Handelt es sich bei genau einem der beiden Knoten m oder n um die Wurzel des entsprechenden Baumes, wird der Wert auf das Maximum $(1, 0)$ gesetzt, da die Umgebung nicht übereinstimmt.

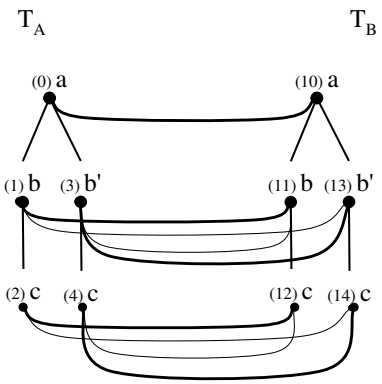

Abbildung 6.16: Beispiel für die Bildung des Wertes $env(e)$: Der niedrigere Wert der Kante (1;11) gegenüber der Kante (1;13) überträgt sich auf die Kante (2;12) gegenüber der Kante (2;14).

Nach Ermitteln des Gesamtgewichts der Kante $(pm; pn)$ lassen sich die env-Werte aller Kanten ermitteln, die zwei Kinder von pm und pn verbinden.
Zur Veranschaulichung der Wirkung des Wertes $env(e)$ sei auf Abbildung 6.16 verwiesen (die dort in Klammern stehenden Nummern dienen lediglich der leichteren Identifizierung und sind für das Verfahren selber ohne Bedeutung): Die Knoten (2) und (4) dieses Beispieles seien in Bezug auf ihren Inhalt und ihre Kinder identisch, die Knoten (1) und (3) sind in Bezug auf ihren Inhalt nicht ganz identisch, aber in Bezug auf ihre Kinder. Außerdem sei ein Knoten (i) des Baumes T_A in Bezug auf den Inhalt und auf die Kinder identisch mit einem Knoten $(i + 10)$ des Baumes T_B. Die `env`-Werte der Kanten $(2; 12)$, $(2; 14)$, $(4; 12)$ und $(4; 14)$ sollen dazu dienen, beispielsweise der dicker gezeichneten Kante $(2; 12)$ einen niedrigeren Gesamtwert zu geben als der Kante $(2; 14)$ und damit der Ähnlichkeit der Väter Rechnung zu tragen, obwohl die Knoten (12) und (14) inhaltlich und in Bezug auf ihre Kinder gleich sind. Die Werte der Kanten der Väter *vererben* sich gewissermaßen auf die der Kinder.

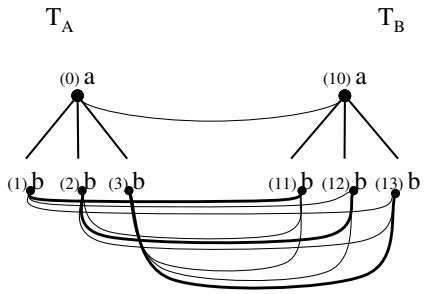

Abbildung 6.17: Bildung von $sib(e)$ einer Kante $e = (m; n$ bei der Berücksichtigung der Reihenfolge der Kinder eines Knotens

Das Ordnungsgewicht `sib`:

Der Wert $sib(e)$ einer Kante $e = (m; n)$ repräsentiert die Abweichung zweier Knoten in Bezug auf ihre Position innerhalb aller Kinder der beiden Väter $P(m)$ und $P(n)$. Diese Abweichung spielt nur dann eine Rolle, wenn die Reihenfolge der Kinder von $P(m)$ und $P(n)$ wichtig ist. Andernfalls wird der Wert $sib(e)$ nicht genutzt. Die Abbildung 6.17 zeigt drei inhaltlich identische mit "b" beschriftete Knoten (1, 2 und 3) des Baumes T_A , die jeweils mit den drei genauso beschrifteten Knoten (11, 12 und 13) des Baumes T_B verbunden sind. In Bezug auf ihre Kinder sind die Knoten mit den Nummern $1, 2, 3, 11, 12$ und 13 ebenfalls identisch. Bei Betrachtung der Abbildung leuchtet

es unmittelbar ein, daß ein Knoten mit der Nummer $i \in \{1..3\}$ in der gewichtsminimalen Kantenüberdeckung mit dem Knoten mit der Nummer $i+10$ verbunden sein sollte (in der Abbildung 6.17 als dick gezeichnete Linien dargestellt), doch wird dieses von keinem der bislang ermittelten Werte `con, chi` oder `env` irgendwelcher Kanten erzwungen. Um dennoch der Bedeutung der Reihenfolge von Knoten innerhalb ihrer Geschwister Rechnung zu tragen, wird der `sib`-Wert einer Kante eingeführt, der wie die anderen Werte `con, chi` und `env` zur Bildung des Gesamtgewichts einer Kante beiträgt. Bei der Ermittlung des Nachfolgergewichts der Väter $chi((pm;pn))$ (im Beispiel $chi((0;10))$ findet eine reihenfolgenorientierte Zuordnung der Knoten $1,2;3;11;12$ und 13 statt, sofern die Reihenfolge eine Rolle spielt - ist dies nicht der Fall, braucht das Ordnungsgewicht nicht berechnet zu werden. Diese Zuordnung kann für die Bildung der Ordnungsgewichte der Kanten genutzt werden, die die Kinder der Knoten pm und pn verbinden. Es wird bei der Ermittlung des Nachfolgergewichts eine Teilmenge $E'_{Cpm-Cpn} \subseteq E_{Cpm-Cpn}$ gebildet (siehe oben). Die `sib`-Werte dieser ausgewählten Kanten erhalten einen niedrigen (günstigen) Wert $0,0$ und die der übrigen einen hohen Wert $1,0$.

Bei Anwendung auf den in der Abbildung 6.17 dargestellten Fall werden die Verbindungen $(1;11)$, $(2;12)$ und $(3;13)$ gegenüber den übrigen aufgewertet: sie erhalten den `sib`-Wert $0,0$, die übrigen dargestellten Verbindungen erhalten den `sib`-Wert $1,0$.

Das Kombinationsgewicht `envsib`:

Unter Berücksichtigung einer Kombinationsgewichtung werden die beiden Werte $env(e)$ und $sib(e)$ einer Kante e zu einem Wert $envsib(e)$ zusammengefaßt. Ein Wichtungsverhältnis von $50:50$, bei dem der Wert $envsib(e)$ durch

$$envsib(e) = \frac{50 \times env(e) + 50 \times sib(e)}{100}$$

berechnet wird, führte in Tests zu guten Zuordnungen.

Mit Hilfe dieser Kombinationsgewichtung ist man in der Lage, die Ähnlichkeit der Knoten bezüglich ihrer Vorfahren für wichtiger zu erachten, als die Ähnlichkeit bezüglich ihrer Position innerhalb der Kinder ihres Vaters oder umgekehrt. Durch die *Preorder*-Traversierung des Baumes T_A für die Bestimmung des Wertes $env(e)$ wird sichergestellt, daß bei Betrachtung einer Kante $(m;n)$ der Wert $conchienvsib(P(m);P(n))$ (also der Väter) bereits ermittelt wurde.

Das Kombinationsgewicht `conchienvsib`:

Unter Berücksichtigung einer Kombinationsgewichtung werden die beiden Werte $conchi(e)$ und $envsib(e)$ einer Kante e zu einem Wert $conchienvsib(e)$ zusammengefaßt. In Tests wurden mit einem Wichtungsverhältnis von $(60:40)$ gute Ergebnisse erzielt, so daß sich der Wert $conchienvsib(e)$ durch

$$conchienvsib(e) = \frac{60 \times conchi(e) + 40 \times envsib(e)}{100}$$

errechnet. Dieser Wert stellt die Grundlage für das endgültige Gewicht der entsprechenden Kante dar.

Behandlung der *Plus*- und *Minus*-Knoten

Der Wert $conchienvsib(e)$ aller Kanten e dient dem nachfolgenden Verfahren zur Ermittlung der gewichtsminimalen Kantenüberdeckung als Maßzahl für die Güte der betreffenden Kante. Eine besondere Stellung nehmen die *Plus*-, die *Minus*-, und die *Plus-Minus*-Kanten ein. Die Gewichte der *Plus*- und *Minus*-Kanten werden so gewählt, daß sie sich in das Gewichtsschema der übrigen Kanten einfügen. Da die *Plus*-Kanten sehr ähnlich gewichtet werden, wie die *Minus*-Kanten, wird

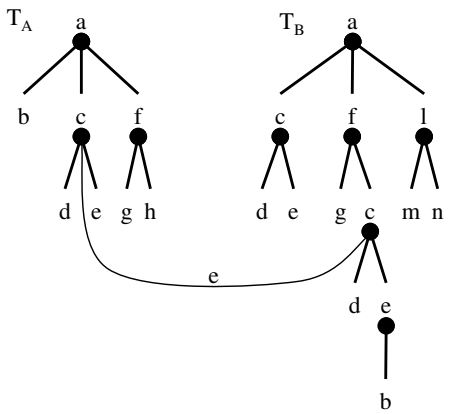

Abbildung 6.18: Bildung von *conchienvsib*(e) einer Kante $e = (m; n)$: in den Wert *conchienvsib*(e) gehen die Werte aller Partnerbeziehungen der übrigen Knoten ein. Je näher ein Knoten an einem der Knoten m oder n liegt, desto stärker ist sein Einfluß auf den Wert der Kante e.

hier nur auf die Gewichtung der *Plus*-Kanten eingegangen. Sowohl für Kanten, die den *Plus*-Knoten mit einem Elementknoten aus T_B verbinden als auch solche, die den *Plus*-Knoten mit einem Textkörperknoten verbinden, gibt es jeweils ein *Mindestgewicht*, zu dem weitere Gewichtsteile hinzuaddiert werden, abhängig davon, wie viele Kinder der betroffene Knoten in T_B hat, beziehungsweise, wie groß der Textkörper ist, den der Knoten repräsentiert. Sowohl die Mindestwerte, als auch die zeichenweise bzw. kindweise hinzukommenden Werte lassen sich als Parameter konfigurieren (siehe Abschnitt 7.8).

Um zu garantieren, daß der *Plus*- und *Minus*-Knoten in der zu ermittelnden gewichtsminimalen Kantenüberdeckung jeweils von mindestens einer Kante getroffen werden, erhält die *Plus-Minus*-Kante ein Gewicht von $0,0$. Man denke sich den Fall des Vergleichs zweier einander sehr ähnlicher Bäume, bei denen die beste Zuordnung keine *Plus*- oder *Minus*-Kanten vorzusehen bräuchte (da der kompakteste *edit-script* keine *insert*- oder *delete*-Operationen enthält). Bekäme die *Plus-Minus*-Kante ein Gewicht, das über dem Gewicht anderer Kanten E' liegt, deren Gewichte größer als $0,0$ sind, könnte es geschehen, daß der *Plus*- oder *Minus*-Knoten in der gewichtsminimalen Kantenüberdeckung mit Knoten der Bäume T_A oder T_B verbunden wird, was zu einer *insert*- oder *delete*-Operation im *edit-script* führte, obwohl dies nicht nötig wäre.

Bemerkungen zum Berechnungsverfahren

Da die nachfolgenden Verfahren zur Reduktion des Kantenüberdeckungsproblems auf ein Korrespondenzproblem und die anschließende Lösung des Korrespondenzproblems nicht korrekt arbeiten, wenn der induzierte Graph mehrere Kanten gleichen Gewichts hat oder Kanten mit dem Gewicht $0,0$ hat, werden die endgültigen Kantengewichte in zweierlei Hinsicht modifiziert: zunächst wird beim Ermitteln des endgültigen Wertes *conchienvsib*(e) einer Kante dafür gesorgt, daß keine zwei Kanten den selben Wert erhalten, indem sämtliche bisher vergebenen Kantengewichte in einer *Hash*-Tabelle gehalten werden und ein zu vergebenes aber bereits vorhandenes Gewicht so oft um einen sehr geringen Betrag $\epsilon \to 0, \epsilon > 0$ angehoben werden, bis es einmalig ist. Das Verfahren zur Ermittlung einer gewichtsminimalen Kantenüberdeckung wird durch künstliche Veränderung der Kantengewichte beeinflußt. Im Falle der hier vorgenommenen Gewichtsveränderungen bleibt jedoch eine negative Beeinflussung aus. Durch das künstliche Anheben der Kantengewichte einiger Kanten findet zwar eine Vorentscheidung statt, doch drückte die Gleichheit von Gewichten mehrerer Kanten aus, daß diese Kanten gleichwertig sind, weshalb eine willkürliche Bevorzugung von Kanten das Verfahren nicht negativ beeinflußt. Einen negativen Einfluß auf die Ermittlung

der gewichtsminimalen Kantenüberdeckung hätte diese Vorgehensweise jedoch, wenn für zwei Teilmengen von Kanten $E_1, E_2 \in E$ mit $W(E_1) = \sum_{e \in E_1} w(e) < W(E_2) = \sum_{e \in E_2} w(e)$ nach dem möglicherweise mehrfachen Anheben des Gewichtes einer Kante $e \in E_1$ um ϵ die Ungleichung $W(E_1) \geq W(E_2)$ gelten würde (wenn durch das Anheben des Gewichtes einer Kante also Ungleichungen bezüglich der Kantengewichte von Teilmengen von Kanten umgekehrt würden). Liegt der Wert ϵ dicht genug bei 0, wird diese Beeinflussung vermieden.
Um zu vermeiden, daß es Kanten mit dem Gewicht $0,0$ gibt, werden sämtliche Kantengewichte um den selben sehr kleinen Betrag $\delta \to 0, \delta > 0$ angehoben, was das Verfahren nicht beeinflußt, wenn der Wert δ klein genug ist.

6.5.2 Beschriftung der Kanten

Nachdem die Kanten des induzierten Graphen jeweils mit einem Kantengewicht versehen wurden, läßt sich mit Hilfe des unter 4.3.2 beschriebenen Verfahren das Kantenüberdeckungsproblem auf ein Korrespondenzproblem reduzieren und lösen. Die in der ermittelten Kantenüberdeckung möglicherweise noch enthaltene *Plus-Minus*-Kante wird entfernt, da sie nur für die Erstellung der Kantenüberdeckung benötigt wurde.
Um aus der erhaltenen gewichtsminimalen Kantenüberdeckung einen *edit-script* zu ermitteln, müssen die Kanten in einem nächsten Schritt beschriftet werden. Dabei wird jede Kante mit einer der sechs Beschriftungen *"nil", "ins", "del", "mov", "cpy"* oder *"glu"* versehen. Die Beschriftung einer Kante läßt sich also als eine Funktion $A : \mathcal{K} \to \{''\texttt{nil}'', ''\texttt{ins}'', ''\texttt{del}'', ''\texttt{mov}'', ''\texttt{cpy}'', ''\texttt{glu}''\}$ auffassen.

Zum Erstellen der Beschriftungen werden die Knoten des Baumes T_A *preorder* durchlaufen.

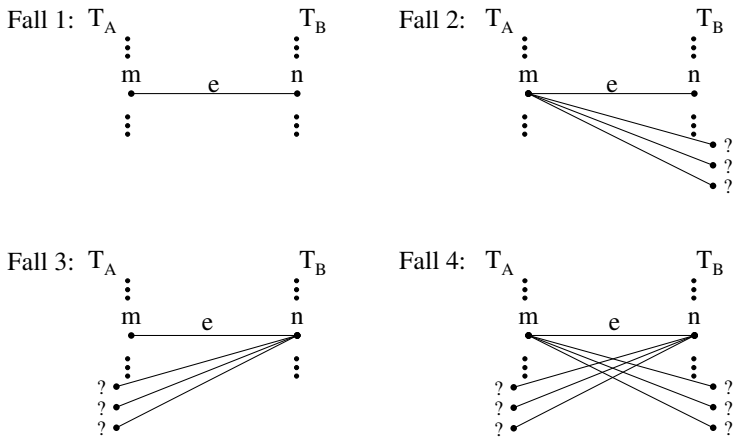

Abbildung 6.19: Vier unterschiedliche Fälle bei der Beschriftung der Kanten

Beschriftung einer Kante e Bei der Beschriftung einer Kante sind (ähnlich, wie in [CGM97]) theoretisch vier, praktisch jedoch nur drei verschiedene Fälle voneinander zu unterscheiden. Dazu seien für eine Kante $e = (m; n)$ der Kantenüberdeckung $\mathcal{K}$ zunächst die beiden Mengen $M_{(m;n)}$ und $N_{(m;n)}$ folgendermaßen definiert: $M_{(m;n)} := \{m' \in T_A : (m'; n) \in \mathcal{K}\}$ und analog dazu $N_{(m;n)} := \{n' \in T_B : (m; n') \in \mathcal{K}\}$. Demnach enthält die Menge $M_{(m;n)}$ diejenigen Knoten aus T_A, die mit dem Knoten n verbunden sind und die Menge $N_{(m;n)}$ diejenigen aus T_B, die mit dem Knoten m verbunden sind. In Abhängigkeit der Mächtigkeiten dieser beiden Mengen ergeben sich vier Fälle:

Fall 1 $|M_{(m;n)}| = 1, |N_{(m;n)}| = 1$: Die beiden Knoten m und n sind nur miteinander verbunden.

Fall 2 $|M_{(m;n)}| = 1, |N_{(m;n)}| > 1$: Der Knoten m ist neben dem Knoten n mit weiteren Knoten aus T_B verbunden. Der Knoten n ist nur mit dem Knoten m verbunden.

Fall 3 $|M_{(m;n)}| > 1, |N_{(m;n)}| = 1$: Der Knoten n ist neben dem Knoten m mit weiteren Knoten aus T_A verbunden. Der Knoten m ist nur mit dem Knoten n verbunden. Dieser Fall wird analog zu Fall 2 behandelt.

Fall 4 $|M_{(m;n)}| > 1, |N_{(m;n)}| > 1$: Dieser Fall tritt nicht auf, da die Kantenüberdeckung $\mathcal{K}$ minimal ist.

Der Fall 1 ist der einfachste dieser Fälle, so daß die Beschriftung der Kanten zunächst anhand dieses Falles dargestellt wird. Anschließend wird die Verallgemeinerung des Verfahrens für die Abdeckung des Falles 2 und 3 beschrieben. Das Verfahren für den Fall 3 verläuft analog zu dem für den Fall 2.

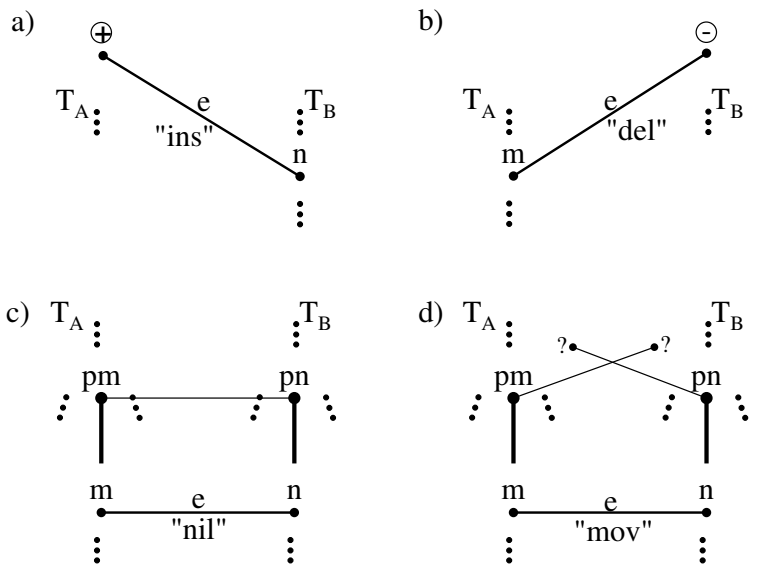

Abbildung 6.20: Fall 1 für die Beschriftung einer Kante e: beide beteiligten Knoten haben jeweils genau einen Partner

Fall 1: Für die Behandlung des Falles 1 wird die betroffene Kante mit einer der vier Beschriftungen *"nil", "ins", "del"* und *"mov"* versehen. Jede dieser vier Beschriftungen führt zu einer entsprechenden Operation des *edit-scripts*. Betrachtet sei eine Kante $e = (m; n)$ der Kantenüberdeckung $\mathcal{K}$, für die der Fall 1 zutrifft (beide Knoten m und n werden jeweils von genau einer Kante getroffen, nämlich der Kante $e = (m; n)$). Die Abbildung 6.20 zeigt die vier möglichen Situationen für diesen Fall:

a) Ist der Knoten m der *Plus*-Knoten und n ein regulärer Knoten des Baumes T_B (also nicht der *Minus*-Knoten), dann wird die Kante $(m; n)$ mit *"ins"* beschriftet. Da die gewichtsminimale Kantenüberdeckung $\mathcal{K}$ für den Knoten n offenbar keinen Partner im Baum T_A enthält, ist dieser Knoten n einzufügen.

b) Ist der Knoten m ein regulärer Knoten aus dem Baum T_A, und n ist der *Minus*-Knoten, wird die Kante $(m; n)$ mit *"del"* beschriftet. Da die gewichtsminimale Kantenüberdeckung $\mathcal{K}$ für den Knoten m offenbar keinen Partner im Baum T_B enthält, ist dieser Knoten m zu entfernen.

c) Die Knoten m und n sind reguläre Knoten der Bäume T_A und T_B und die beiden Väter pm und pn der Knoten m und n sind durch eine Kante miteinander verbunden. Falls $sib(e) = 0,0$ gilt, wird die Kante e mit *"nil"* beschriftet, andernfalls wird die Kante mit *"mov"* beschriftet[2].

   Eine Beschriftung *"mov"* einer Kante $(m;n)$ deutet auf eine *move*-Operation des *edit-scripts* hin, die den Knoten m zu einem anderen Vater oder zu einer anderen Position bezüglich der Kinder des ursprünglichen Vaters verschiebt. Ersteres findet nur dann statt, wenn die Väter der Knoten m und n nicht miteinander verbunden sind, letzteres wenn $sib(e) \neq 0$ gilt.

d) Die Knoten m und n sind reguläre Knoten der Bäume T_A und T_B und die beiden Väter pm und pn der Knoten m und n sind nicht durch eine Kante miteinander verbunden. Die Kante e wird mit *"mov"* beschriftet, da das Fehlen der Verbindung zwischen den Vätern pm und pn der Knoten m und n darauf schließen läßt, daß der Knoten m durch eine Operation des *edit-scripts* zu einem neuen Vater verschoben werden wird.

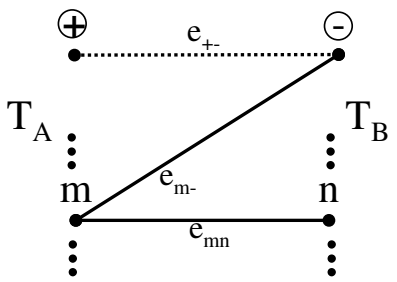

Abbildung 6.21: Ein Knoten $m \in T_A$ kann nicht sowohl mit dem *Minus*-Knoten, als auch mit einem anderen Knoten $n \in T_B$ verbunden sein. Analoges gilt auch für einen Knoten $n \in T_B$ und den *Plus*-Knoten.

Fall 2: Um dem Fall 2 gerecht werden zu können, wird das Verfahren erweitert und eine fünfte Form der Beschriftung (*"cpy"*) eingeführt, die den Vorgang des Kopierens eines Knoten andeutet. Ein Knoten, der mit dem *Minus*-Knoten verbunden ist, kann nicht darüberhinaus mit einem anderen Knoten verbunden sein. Die Abbildung 6.21 zeigt den nicht möglichen Fall, in dem ein Knoten m sowohl mit einem Knoten n, als auch mit dem *Minus*-Knoten verbunden ist. Die Kante e_{+-} trug (nahezu) kein Gewicht zum Gesamtgewicht einer Kantenüberdeckung bei[3], weshalb eine Lösung mit den Kanten e_{+-} und e_{mn} immer günstiger ist, als eine mit den Kanten e_{m-} und e_{mn}. Demzufolge wird eine gewichtsminimale Kantenüberdeckung nicht die Kanten e_{m-} und e_{mn} enthalten, sondern höchstens eine der beiden. Aus diesem Grunde bleiben die für Fall 1a) und 1b) beschriebenen Verfahren zur Beschriftung unverändert. Lediglich die Punkte c) und d) werden erweitert.

Im Folgenden werden diese beiden Fälle 2c) und 2d) beschrieben. Zur korrekten Beschriftung der Kanten $e = (m;n)$, die einen Knoten m treffen, werden diese zunächst in mehrere Partitionen eingeteilt.
Ausgehend von einer Kante $e = (m;n)$ erhält man die Menge N, $|N| > 1$ derjenigen Knoten aus T_B mit denen der Knoten m in der gefundenen gewichtsminimalen Kantenüberdeckung verbunden

[2] $sib((m;n))$ ist genau dann $0,0$, wenn die Reihenfolge der Kinder der Knoten $P(m)$ und $P(n)$ keine Rolle spielt, oder die Knoten m und n bezüglich der Reihenfolge derjenigen Kinder von $P(m)$ und $P(n)$ übereinstimmen, die mit Kindern des jeweils anderen Vaters verbunden sind.

[3] Diese Kante ist zum Beginn der Beschriftung entfernt worden, wurde jedoch bei der Bildung der gewichtsminimalen Kantenüberdeckung berücksichtigt.

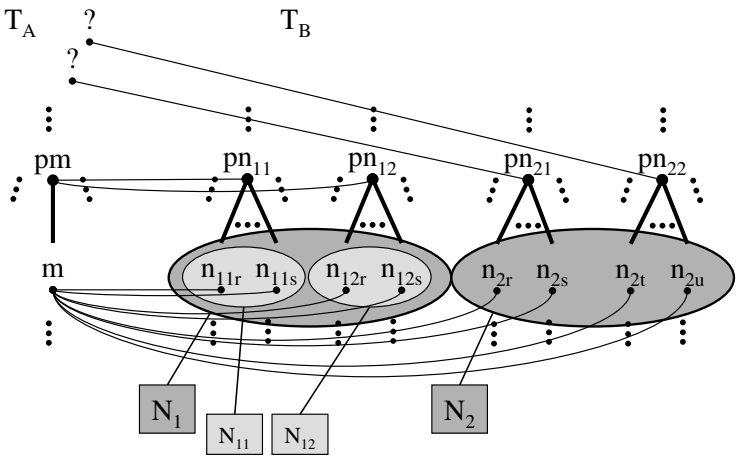

Abbildung 6.22: Fall 2 für die Beschriftung der Kanten: einer der beteiligten Knoten hat mehr als einen Partner.

ist. Diese Menge läßt sich als Vereinigung zweier Untermengen $N = N_1 \cup N_2$ auffassen, die sich wie folgt definieren (siehe hierzu auch Abbildung 6.22):

$$N_1 := \{n' \in T_B : (m; n') \in \mathcal{K} \wedge (P(m); P(n')) \in \mathcal{K}\}$$

$$N_2 := \{n' \in T_B : (m; n') \in \mathcal{K} \wedge (P(m); P(n')) \notin \mathcal{K}\}$$

Die Menge N_1 enthält damit diejenigen Verbindungspartner von m, deren Väter mit dem Vater von m verbunden sind, während die Menge N_2 diejenigen Partner von von m enthält, deren Väter nicht mit dem Vater von m verbunden sind.
Die Menge N_1 sei zusätzlich so partitioniert, daß alle Knoten einer Partition den selben Vater haben:

$$N_1 = \cup_{j=1}^{k} N_{1j} : \; n_{1jr}, \; n_{1js} \in N_{1j} \Rightarrow P(n_{1jr}) = P(n_{1js})$$

Die Menge N_2 wird nicht weiter partitioniert.

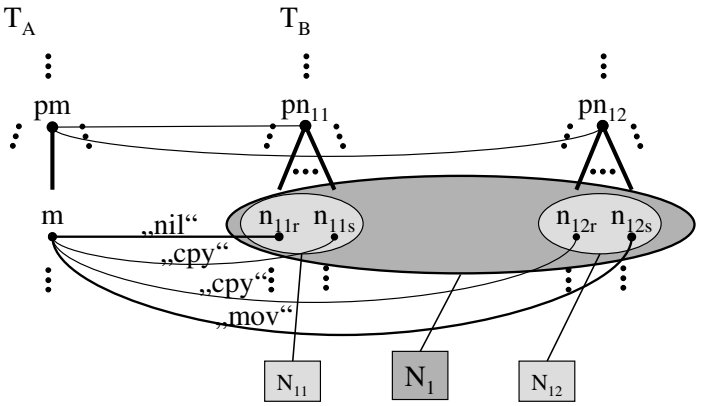

Abbildung 6.23: Beschriftung der Kanten für die Menge N_1

Die Beschriftung aller Kanten, die den Knoten m treffen verläuft in zwei Schritten (hier Fall 2c und Fall 2d genannt), die den oben genannten Punkten c) und d) entsprechen. Dabei wird in Fall 2c) die Menge N_1 mit ihren Partitionen und in Fall 2d) die Menge N_2 bearbeitet.

Fall 2c) Aus jeder der Partitionen N_{1j} wird derjenige Knoten n^* gewählt, der mit der für seine Partition günstigsten Kante $e^* = (m; n^*)$ mit dem Knoten m verbunden ist. Die Kante e^* wird mit der Beschriftung *"nil"* beschriftet, falls $sib(e^*) = 0,0$ gilt. Andernfalls wird e^* mit *"mov"* beschriftet. Die Kanten aller übrigen Knoten dieser Partition, die den Knoten m treffen, werden mit der Beschriftung *"cpy"* versehen. In Abbildung 6.23 hat die Menge N_1 genau zwei Partitionen, die jeweils eine Anzahl von Knoten enthalten, die mit dem Knoten m verbunden sind. Die Kante $(m; n_{11r})$ wird mit *"nil"* beschriftet, da sie - so wird hier für das Beispiel angenommen - ein geringeres Gewicht aufweist als die Kante $(m; n_{11s})$ und ihr `sib`-Wert $0,0$ ist. Die Kante $(m; n_{12s})$ hat ein geringeres Gewicht als die Kante $(m; n_{12r})$, und ihr `sib`-Wert ist nicht $0,0$, so daß diese Kante mit *"mov"* beschriftet wird. Alle übrigen Kanten werden mit *"cpy"* beschriftet. Ist die Menge N_1 nicht leer, wird in diesem Schritt mindestens eine Kante mit *"nil"* oder *"mov"* beschriftet, womit gewissermaßen der Verbleib des Knoten m dokumentiert ist. Diese Tatsache wird in Schritt 2d) berücksichtigt. Die Abbildung 6.24 bezieht sich nicht auf das selbe Beispiel, wie die Abbildung 6.23, da sich für die Abbildung 6.24 andernfalls der wenig interessante Fall ergäbe, in dem sämtliche mit dem Knoten m verbundene Kanten, die Knoten der Menge N_2 treffen, mit *"cpy"* beschriftet würden.

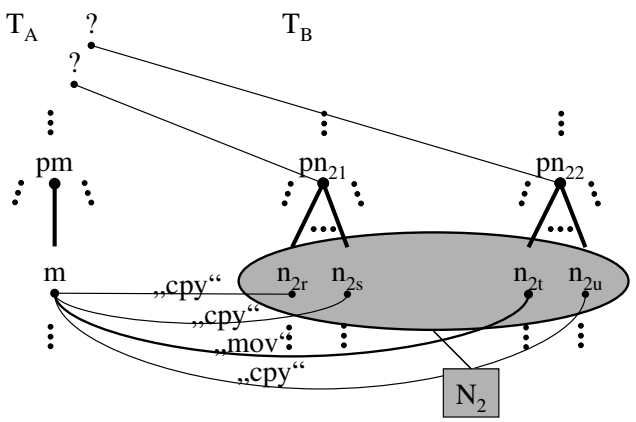

Abbildung 6.24: Beschriftung der Kanten für die Menge N_2

Fall 2d) Falls die Menge N_1 leer ist und damit bei Schritt 2c) keine Kante mit *"nil"* oder *"mov"* beschriftet wurde (der Verbleib des Knoten m somit noch nicht dokumentiert wurde), wird aus der Menge N_2 ein Knoten n^* gewählt, der mit der günstigsten Kante $e^* = (m; n^*)$ mit dem Knoten m verbunden ist. Die Kante e^* wird mit *"mov"* beschriftet. Die Kanten aller übrigen Knoten der Menge N_2, die den Knoten m treffen, werden mit der Beschriftung *"cpy"* versehen. Bei der Abbildung 6.24 hat man davon auszugehen, daß im Schritt 2c) keine Kante mit *"nil"* oder *"mov"* beschriftet wurde, also der Primärpartner des Knoten m noch nicht gefunden wurde. Durch die Beschriftung einer Kante mit *"mov"* wird in diesem Schritt dieser Primärpartner gesetzt.

Fall 3: Die Behandlung einer gewählten Kante e, auf die der Fall 3 zutrifft, verläuft analog zu der Behandlung einer Kante, auf die der Fall 2 zutrifft. Es wird lediglich anstelle einer Beschriftung *"cpy"* die Beschriftung *"glu"* verwendet, um den Vorgang des Einschmelzens eines Knoten darzustellen.

6.5.3 Ermittlung des *edit-scripts*

Nachdem eine beschriftete, gewichtsminimale Kantenüberdeckung des induzierten Graphen gefunden wurde, ist es möglich, hieraus einen *edit-script* zu erzeugen, der den Baum T_A schrittweise in den Baum T_B transformiert.

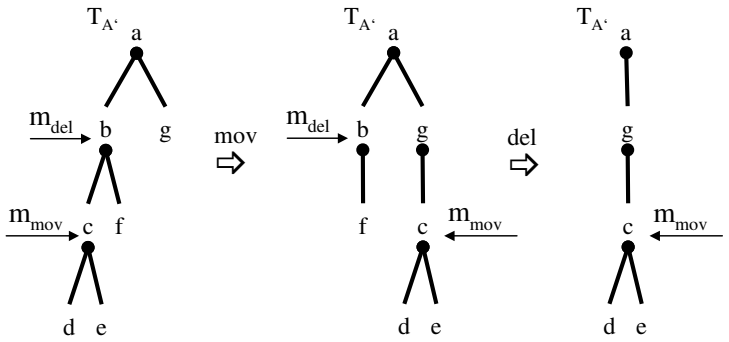

Abbildung 6.25: Beachtung der Ausführungsreihenfolge einiger Operationen des *edit-scripts*: Ausführung der *move*-Operation (`mov(2,6,0)`) vor Ausführung der *delete*-Operation (`del(1)`)

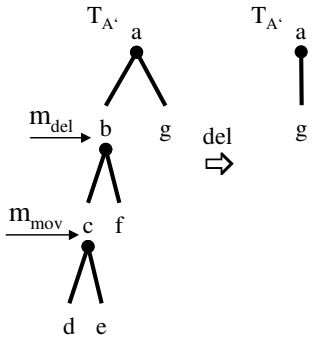

Abbildung 6.26: Beachtung der Ausführungsreihenfolge einiger Operationen des *edit-scripts*: Ausführung der *delete*-Operation (`del(1)`) vor Ausführung der *move*-Operation (die dann nicht mehr möglich ist)

Hierbei spielt die Ausführungsreihenfolge der sich im *edit-script* befindenden Operationen eine wichtige Rolle: es macht beispielsweise einen Unterschied, ob das Entfernen eines Teilbaumes $T(m_{del})$ vor oder nach dem Verschieben eines sich innerhalb $T(m_{del})$ befindenden Teilbaumes $T(m_{mov})$ aus $T(m_{del})$ heraus stattfindet[4] (siehe Abbildungen 6.25 und 6.26). Die Bearbeitung einiger Kanten muß demnach auf die Bearbeitung anderer Kanten warten, so daß wie in Abbildung 6.27 dargestellt vorgegangen wird. Nachfolgend wird die Vorgehensweise für jede mögliche Kantenbeschriftung - dargestellt durch die oben eingeführte Funktion A - erläutert. Anschließend werden einige Regeln zur Einhaltung der erwähnten Reihenfolgen beschrieben.

$e = (m; n)$, $A(e) =$ `nil`**:**Weisen Die Knoten m und n unterschiedlichen Inhalt auf, wird bei der Bearbeitung der Kante lediglich der Inhalt des betreffenden Knoten m verändert, so daß er dem des Knoten n entspricht (bei Elementknoten die Attribute und bei Textkörperknoten die Zeichenkette). Ist das der Fall, wird eine entsprechende *update*-Operation erzeugt und dem *edit-script* hinzugefügt. Die Kante e wird als *„bearbeitet“* markiert.

[4] $T(m)$ bezeichnet den bei dem Knoten m beginnenden Teilbaum.

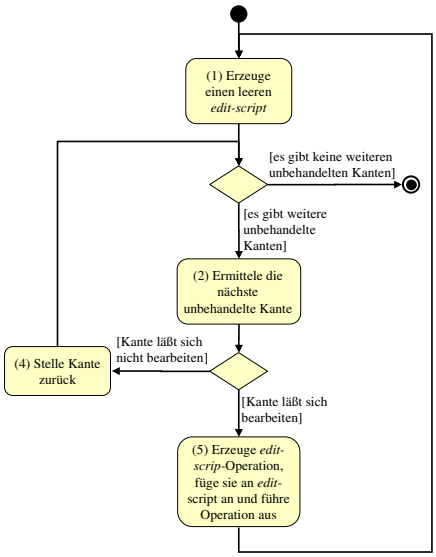

Abbildung 6.27: Erstellung eines *edit-scripts*: prinzipielle Vorgehensweise

Da eine *update*-Operation die Struktur des Baumes nicht verändert, kann es nicht sein, daß die Bearbeitung anderer Kanten auf diese Operation warten muß. Umgekehrt ist es jedoch möglich, daß die Bearbeitung dieser Kante auf andere Operationen warten muß.

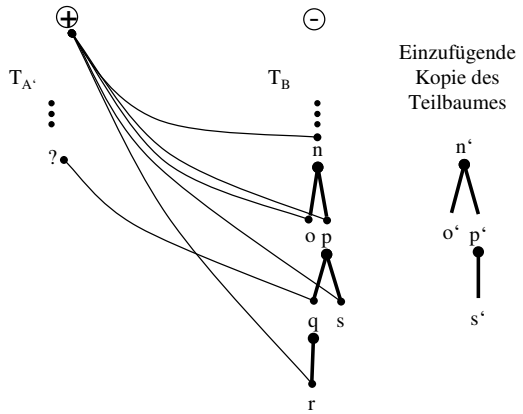

Abbildung 6.28: Erstellung eines *edit-scripts*: Erzeugung einer Kopie des mit dem *Plus*-Knoten verbundenen Teilbaums $T(n)$

$e = (m; n)$, $A(e) =$ `ins`**:** Um zu vermeiden, daß der zu erstellende *edit-script* viele einzelne *insert*-Operationen enthält, die letztlich einen zusammenhängenden Teilbaum einfügen, wird zunächst versucht, diesen insgesamt einzufügenden Teilbaum zu identifizieren. Ist der Vater von n ($P(n)$)mit dem *Plus*-Knoten verbunden, nimmt $P(n)$ die Rolle von n ein. Dieses wird so oft wiederholt, bis der Vater von n nicht mit dem *Plus*-Knoten verbunden ist. Damit ist n die Wurzel eines Teilbaumes aus T_B, bei dem die Wurzel und einige weitere Knoten mit dem *Plus*-Knoten verbunden sind. Es wird zunächst der einzufügende Teilbaum erstellt. Zu diesem Zweck wird eine Kopie des Teilbaumes $T(n)$ erstellt, wobei die Kopie eines Knotens $n_i \in T(n)$ mit n_i' bezeichnet wird. Ist ein Knoten des Teilbaums $T(n)$ nicht mit dem *Plus*-Knoten verbunden, soll der kopierte Teilbaum keine Kopie dieses Knoten mit seinen

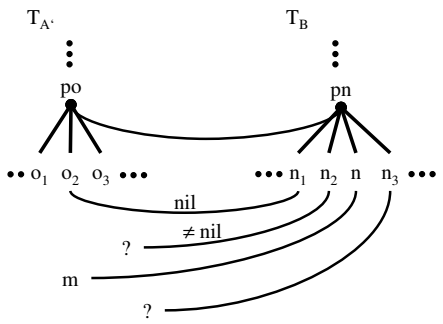

Abbildung 6.29: Ermittlung des Primärpartners und der Zielposition eines einzufügenden Knoten

Nachfolgern haben. Jede Kante $e_{ins} = ([+]; n_i)$ wird entfernt und durch eine mit *"nil"* beschriftete Kante $e_{nil} = (n'_i; n_i)$ ersetzt.

Die Abbildung 6.28 zeigt die Erzeugung der Kopie, sie sieht für die Knoten q und r keine Kopien vor, da q nicht mit dem *Plus*-Knoten verbunden ist und r Kind von q ist.

Als nächstes ist derjenige Knoten aus $T_{A'}$ zu identifizieren, der der Vater (Zielvater) des einzusetzenden Knotens werden soll. Dieses ist der Primärpartner des Knotens $P(n)$[5]. Sollte es keinen Primärpartner zu $P(n)$ geben, oder wenn der Primärpartner seinerseits von mehreren Kanten getroffen wird (wenn er also kopiert werden soll), so muß die Bearbeitung dieser Kante so lange warten, bis ein Primärpartner zu $P(n)$ angegeben werden kann und dieser nur von einer Kante getroffen wird.

Für die zu erzeugende Operation des *edit-scripts* ist die Knotennummer des Zielvaters wichtig, die er bei einer *Preorder*-Traversierung einnimmt. In der Abbildung 6.29 wird dieser Knoten mit *po* bezeichnet.

Spielt die Reihenfolge der Kinder der Knoten *po* und *pn* eine Rolle, so ist zusätzlich die Position zu ermitteln, die der einzusetzende Knoten bezüglich der Kinder von *po* einzunehmen hat. Alle Kinder des Zielvaters, die von einer mit *"nil"* beschrifteten Kante getroffen werden, befinden sich in beiden Bäumen gewissermaßen an derselben Stelle (die Beschriftung *"nil"* drückte aus, daß keine strukturellen Veränderungen nötig sind). Dies bedeutet, daß die gesuchte Position relativ zu solchen Kindern des Zielvaters zu geschehen hat, die von einer mit *"nil"* beschrifteten Kante getroffen werden. Dazu ermittelt man denjenigen Knoten $o_i \in C(po)$, der mit einer mit *"nil"* beschrifteten Kante mit demjenigen Knoten $n_i \in C(pn)$ verbunden ist, der links von n, aber am weitesten rechts steht, denn dieser befindet sich in beiden Bäumen an der selben Stelle und ist derjenige Knoten, hinter dem der einzusetzende Knoten einzufügen ist. In der Abbildung 6.29 ist dieses der Knoten n_1, da n_2 nicht von einer mit *"nil"* beschrifteten Kante getroffen wird. Bezeichne $pos(o_i)$ diejenige Position des Knotens o_i innerhalb aller Kinder von $P(o_i)$, dann ergibt sich die Kind-Position des einzufügenden Teilbaumes mit $pos(o_i)+1$, falls ein Knoten o_i gefunden wurde und sonst 0. Mit Hilfe des Zielvaters und der Kind-Position ist es möglich, eine entsprechende *insert*-Operation für den *edit-script* zu erzeugen und den bei dem Knoten n' beginnenden Teilbaum in den Baum $T_{A'}$ den ermittelten Werten entsprechend einzufügen.

Da die Bearbeitung einer mit *"ins"* beschrifteten Kante e die Existenz der Knoten $n_i \in T(n)$ betrifft, muß die Bearbeitung all der einen Knoten n_i treffenden Kanten warten, bis die Kante e vollständig bearbeitet wurde.

[5] der Primärpartner eines Knotens $n \in T_B$ ist derjenige Knoten, mit dem n verbunden ist. Wird n von mehreren Kanten getroffen, so gibt es keinen Primärpartner von n.

$e = (m; n)$, $A(e) =$ `del`**:** Die Bearbeitung einer mit *"del"* beschrifteten Kante ist verhältnismäßig einfach. Es wird die Knotennummer des zu entfernenden Knotens m ermittelt (*Preorder*-Traversierung), eine entsprechende *delete*-Operation für den *edit-script* erstellt und der Teilbaum $T(m)$ aus dem Baum $T_{A'}$ entfernt. Die Kanten, die die entfernten Knoten trafen, werden ebenfalls entfernt.

Da die Bearbeitung dieser Kante e die Existenz der Knoten $m_i \in T(m)$ betrifft, muß die Bearbeitung der Kante e auf die vollständige Bearbeitungen all jener Kanten e_i warten, die einen Knoten m_i treffen und nicht mit dem *Minus*-Knoten verbunden sind.

$e = (m; n)$, $A(e) =$ `mov`**:** Nach der Beschreibung der Behandlung von mit *"ins"* beschrifteten Kanten ist die Bearbeitung einer mit *"mov"* beschrifteten Kante recht einfach. Zunächst wird wie oben beschrieben der Zielvater gesucht (der Primärpartner von $P(n)$). Falls dieser nicht ermittelbar ist (wenn $P(n)$ mehrere Partner hat), muß die Bearbeitung der Kante e warten, bis der Zielvater bestimmbar ist. Ebenso wie oben beschrieben, wird die Kindposition ermittelt, die der zu verschiebene Teilbaum $T(m)$ innerhalb der Kinder des Knoten $P(n)$ einzunehmen hat. Die Kante e erhält anstelle der Beschriftung *"mov"* die Beschriftung *"nil"*.

Da die Bearbeitung dieser Kante e nicht die Existenz anderer Knoten betrifft (es werden lediglich Knoten verschoben), braucht die Bearbeitung keiner Kante auf die von e zu warten.

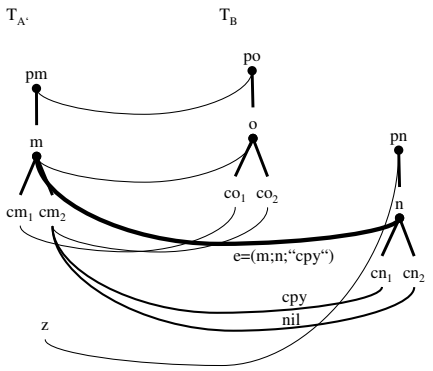

Abbildung 6.30: Bearbeitung einer mit *"cpy"* beschrifteten Kante: Ausgangssituation

$e = (m; n)$, $A(e) =$ `cpy`**:** Für die Behandlung einer mit *"cpy"* beschrifteten Kante $e = (m; n)$ ist zunächst eine Kopie des Teilbaumes $T(m)$ zu erstellen. Diese Teilbaumkopie beginnt mit dem kopierten Knoten m'. Wie bei der Behandlung einer mit *"ins"* beschrifteten Kante beschrieben, ist der Knoten im Baum $T_{A'}$ zu ermitteln, der der neue Vater des einzusetzenden Teilbaumes $T(m')$ werden soll. Die Abbildung 6.30 zeigt die Ausgangssituation dieser Behandlungen, in der der Zielvater die Bezeichnung z trägt. Genau wie ebenfalls oben beschrieben, wird gegebenenfalls die Kind-Position ermittelt, so daß der Teilbaum $T(m')$ in den Baum $T_{A'}$ eingesetzt werden kann. Die Kante $e = (m; n)$ wird ersetzt durch eine Kante $e' = (m'; n)$, die die Beschriftung *"nil"* trägt. Alle übrigen Kanten $(m_i; n_i)$, $m_i \in T(m) \wedge n_i \in T(n)$ werden entfernt und jeweils durch Kanten $(m'_i; n_i)$ ersetzt, wobei m'_i gerade die Kopie des Knoten m_i bezeichne. Die Beschriftung der Kanten $(m'_i; n_i)$ sei die gleiche, wie die der ersetzten Kante $(m_i; n_i)$. Für diejenigen Knoten $m'_i \in T(m')$, die von keiner Kante getroffen werden, werden neue Kanten $(m'_i; [-])$ eingesetzt und mit der Beschriftung *"del"* versehen. Die Abbildung 6.31 zeigt das Ergebnis dieser Bearbeitungen des Beispiels.

Die Bearbeitung einer Kante $e^* = (m^*; n^*)$, vor der Bearbeitung der Kante $e = (m; n)$, wobei $m^* \in T(m)$ oder $z^* \in T(m)$ (z^* bezeichnet den für die Bearbeitung der Kante e^* ermittelten

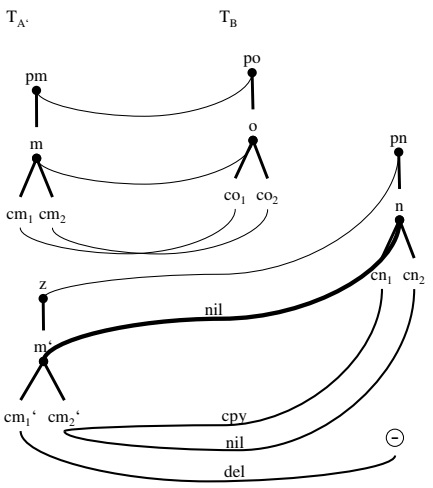

Abbildung 6.31: Bearbeitung einer mit *"cpy"* beschrifteten Kante: Einsetzen des kopierten Teilbaumes

Zielvater) würde Veränderungen an dem Teilbaum $T(m)$ vornehmen, bevor dieser kopiert wird. Wird dieser (bereits veränderte Teilbaum) kopiert, enthalten sowohl das Original als auch die Kopie die Veränderungen, was nicht erwünscht ist, da die vorgenommenen Veränderungen nach dem Kopieren möglicherweise an dem Original ($T(m)$) oder der Kopie ($T(m')$) wieder rückgängig gemacht werden müßten. Daher muß die Bearbeitung all dieser oben genannten Kanten e^* auf die vollständige Bearbeitung der Kante e warten.

$e = (m; n)$, $A(e) =$ `glu`**:** Die Bearbeitung einer mit *"glu"* beschrifteten Kante ist recht einfach: ähnlich, wie bei einer mit *"del"* beschrifteten Kante, wird der Teilbaum $T(m)$ aus dem Baum $T_{A'}$, sowie alle Kanten $e^* = (m^*, n^*)$, $m^* \in T(m)$ entfernt.

Da die Bearbeitung dieser Kante e die Existenz der Knoten $m_i \in T(m)$ betrifft, muß die Bearbeitung der Kante e auf die vollständige Bearbeitung all jener Kanten e_i warten, die einen Knoten m_i treffen und nicht mit einem Knoten aus $T(n)$ verbunden sind.

Die Bearbeitung vieler Kanten $e = (m; n)$ führt zur Erzeugung neuer, mit *"nil"* beschrifteter Kanten oder zur Ersetzung bestehender Kanten durch solche, die mit *"nil"* beschriftet sind. Es ist sinnvoll, diese entstandenen, noch nicht bearbeiteten Kanten direkt im Anschluß an die Bearbeitung der Kanten zu behandeln, die zu deren Erzeugung führten. Auf diese Weise liegen die Operationen, die die Knoten m und n betreffen im resultierenden *edit-script* dichter beieinander, was die Verständlichkeit erhöht.

In den Abbildungen 6.32 bis 6.37 wird die Erstellung eines *edit-scripts* anhand eines Beispieles dargestellt. Die jeweils bearbeiteten Kanten sind fett dargestellt. Die Bildunterschrift zeigt den aktuellen Stand des *edit-scripts es*. Man beachte, daß sich der Arbeitsbaum $T_{A'}$ nach jedem Teilschritt (Abbildung 6.32 bis 6.37) verändert. Die mit *"ins"* und *"del"*, sowie die mit *"cpy"* und *"glu"* beschrifteten Kanten sind jeweils gleich dargestellt, lassen sich aber dennoch gut voneinander unterscheiden, da die *"ins"*-Kanten mit dem *Plus*-Knoten, die *"del"*-Kanten mit dem *Minus*-Knoten verbunden sind, die *"cpy"*-Kanten einen Knoten aus T_A treffen, der von mehreren Kanten getroffen wird und die *"glu"*-Kanten einen Knoten aus T_B treffen, der von mehreren Kanten getroffen wird.

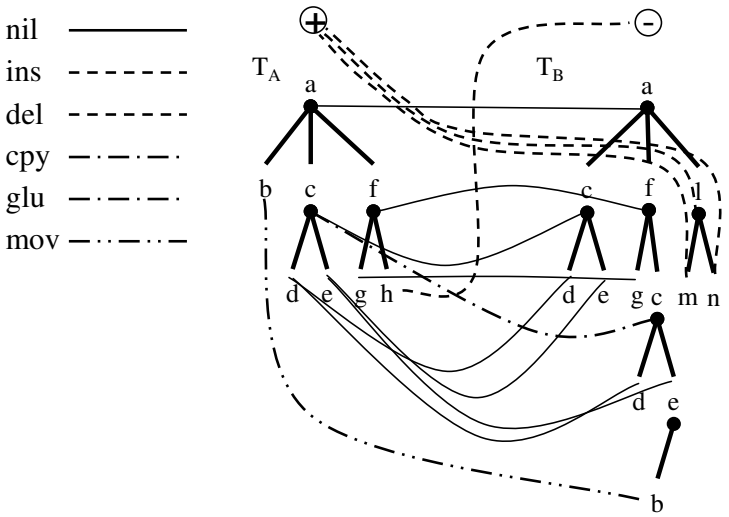

Abbildung 6.32: Erstellung eines *edit-scripts*: Ausgangssituation mit beschrifteter, gewichtsminimaler Kantenüberdeckung; $es = ()$

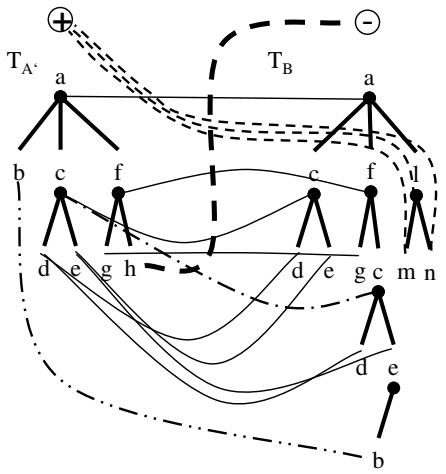

Abbildung 6.33: Erstellung eines *edit-scripts*: Bearbeitung der *"del"*-Kante; $es =$ (`del`(7))

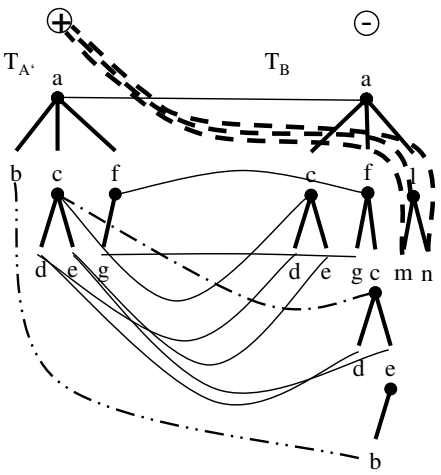

Abbildung 6.34: Erstellung eines *edit-scripts*: Bearbeitung der *"ins"*-Kante; $es = (\texttt{del}(7),\ \texttt{ins}(l[m,n],0,3))$

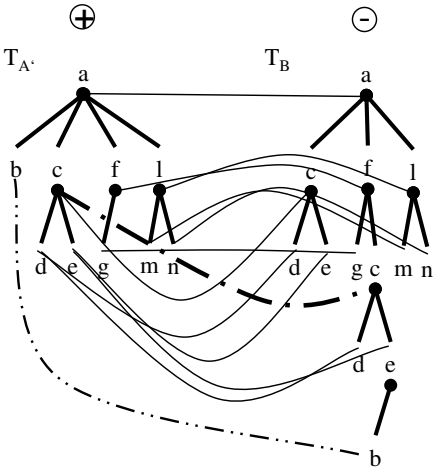

Abbildung 6.35: Erstellung eines *edit-scripts*: Bearbeitung der *"cpy"*-Kante; $es = (\texttt{del}(7),\ \texttt{ins}(l[m,n],0,3),\ \texttt{cpy}(2,5,1))$

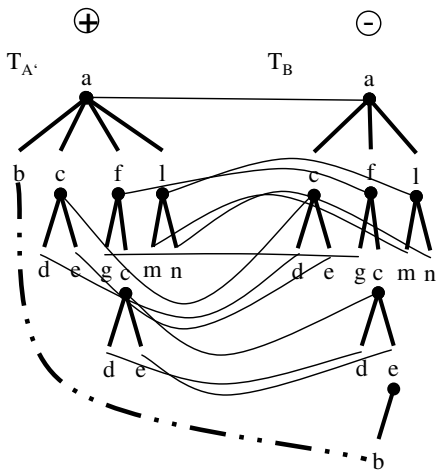

Abbildung 6.36: Erstellung eines *edit-scripts*: Bearbeitung der *"mov"*-Kante; $es = (\texttt{del}(7),\ \texttt{ins}(l[m,n],0,3),\ \texttt{cpy}(2,5,1),\ \texttt{mov}(1,9,0))$

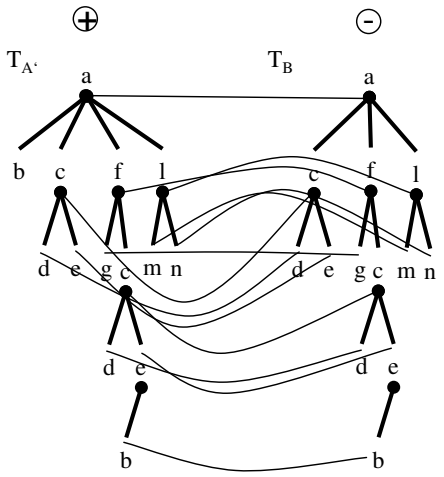

Abbildung 6.37: Erstellung eines *edit-scripts*: der zu T_B isomorphe Baum

6.5.4 Betrachtungen zur Zeitabschätzung

Nachdem in den Abschnitten 6.5.1 bis 6.5.3 die einzelnen Schritte zur Generierung eines *edit-scripts* beschrieben wurden, werden hier einige Überlegungen zu deren zeitlichen Verhalten angestellt. Dieses sind grobe Abschätzungen und keine formalen Beweise. Die Zeitabschätzungen werden für die einzelnen Schritte des Verfahrens angegeben:

1. Erzeugung eines induzierten Graphen
2. Lösen des Korrespondenzproblems
3. Beschriftung der Kanten
4. Erzeugung des *edit-scripts*

Bei den folgenden Zeitabschätzungen für die einzelnen Schritte wird davon ausgegangen, daß a die Anzahl der Knoten des Baumes T_A und b die Anzahl der Knoten des Baumes T_B bezeichnet. Da

für die Zeitabschätzung die Anzahl der Kinder der Knoten wichtig ist, sei c die maximale Anzahl von Kindern aller Knoten: $c = max\{|C(m)| : m \in T_A \vee m \in T_B\}$

Erzeugung eines induzierten Graphen Wie in 6.5.1 beschrieben, erfolgt die Generierung des Gewichts einer Kante $e = (m; n)$ mit $m \in T_A, n \in T_B$ in insgesamt zwei Phasen.

Phase 1: Es werden die Knoten des Baumes T_A in einer *postorder*-Traversierung durchlaufen, wobei sämtliche Kanten $e = (m; n)$ der besuchten Knoten m behandelt werden. Damit ergibt sich zunächst eine Komplexität von $O(ab)$. Für jedes Knotenpaar $(m; n)$ ist eine Zuordnung der Kinder des Knoten m zu den Kindern des Knoten n zu ermitteln. Im Falle des Vergleichs ungeordneter Bäume ist dieses das Korrespondenzproblem (welches mit einer Zeitkomplexität von $O(p^3)$ für insgesamt p Knoten gelöst werden kann [Jun90]). Im Falle des Vergleichs geordneter Bäume ergibt sich das in [MW98] beschriebene Problem, in dem eine Zuordnung der Kinder unter Berücksichtigung der Reihenfolge gefunden werden muß (es kann mit einer Zeitkomplexität von $O(p^2)$ gelöst werden, wobei $p = max\{|C(m)|, |C(n)|\}$ ist [MW98]). Wird für die Anzahl der Kinder eines Knotens pauschal der oben erwähnte Wert c angenommen, erhält man somit eine obere Komplexitätsgrenze von $O(abc^3)$ für die Betrachtung ungeordneter Bäume und $O(abc^2)$ für die Betrachtung geordneter Bäume.

Phase 2: Es werden die Knoten des Baumes T_A in einer *Preorder*-Traversierung durchlaufen, wobei wieder sämtliche Kanten der besuchten Knoten behandelt werden müssen. Damit ergibt sich auch hier die Komplexität von $O(ab)$.

Lösen des Korrespondenzproblems: Das Problem, eine gewichtsminimale Kantenüberdekkung für den induzierten Graphen zu finden, wird in ein Korrespondenzproblem transformiert, dessen zugrundeliegender bipartiter Graph eine Anzahl von $2ab$ Knoten aufweist, womit die Lösung des Korrespondenzproblems mit $O((2ab)^3)$ angegeben werden kann [Jun90].

Die Beschriftung der Kanten: läßt sich mit einem Aufwand von $O(a + b)$ lösen.

Die Erzeugung des *edit-scripts*: läßt sich ebenfalls mit einem zur Größe der beiden Bäume linearen Aufwand von $O(2(a + b))$ lösen.

Somit ergibt sich eine geschätzte Gesamtkomplexität von $O(abc^2)$ für die Betrachtung geordneter und $O(abc^3)$ für die Betrachtung ungeordneter Bäume.

Kapitel 7

Ausgewählte Aspekte der Realisierung

Nachdem im vorigen Kapitel 6 das Verfahren für den in dieser Arbeit implementierten Dienst vorgestellt wurde, werden in diesem Kapitel einige Aspekte der Realisierung dieser Implementierung beschrieben. Diese Beschreibung beschränkt sich auf die wichtigsten Aspekte, des insgesamt rund 160 Klassen umfassenden Dienstes.

Besonderen Wert wurde bei der Implementierung des Dienstes auf folgende Punkte gelegt:

- Flexible Konfigurierbarkeit des Dienstes
- Erweiterbarkeit des Dienstes
- Wiederverwendbarkeit einzelner Komponenten
- Nachvollziehbarkeit der Verfahren durch eine geeignete Visualisierung

Dieses umfaßt unter anderem die Verwendung bereits existierender Komponenten und die Verwendung von Entwurfsmustern.

7.1 Grobarchitektur

Dieser Abschnitt gibt einen groben Überblick über die Architektur der Implementierung.

Abbildung 7.1 zeigt die Grobarchitektur des Systems als Paketdiagramm [FS97, Oes97]. Eine zentrale Rolle nimmt das mit `XTree` bezeichnete Paket ein, deren Bestandteil die in Abschnitt 7.7 vorgestellte Applikation ist. Das Paket `XTree` umfaßt Klassen, die die grobe Steuerung des Ablaufs vornehmen. Durch sie wird das Einlesen der Bäume, der Vergleich und die Ausgabe eines XML-Dokuments zur Repräsentation des *edit-scripts* veranlaßt. Es verwendet hierzu die in der Abbildung umliegenden Pakete. Die Aufgabe des `Parser`-Pakets ist es, jeweils ein XML-Dokument zu analysieren, um daraus einen Baum zu generieren, der dem mit `TreeSystem` bezeichneten Paket übergeben werden kann. Dabei verwendet das `Parser`-Paket existierende Pakete zur Analyse von XML-Dokumenten [IBM98a]. Die eingelesenen Bäume lassen sich mit Hilfe eines dafür entwickelten Pakets darstellen. Dieses ermöglicht neben einer textuellen Darstellung durch vertikal gezeichnete Bäume auch eine graphische durch horizontal gezeichnete Bäume (siehe Abbildung 7.10). Das Paket `TreeSystem` enthält die beiden miteinander zu vergleichenden Bäume, die jeweils ein XML-Dokument repräsentieren. Dieses wird einem Paket `Comparer` übergeben, welches den aus mehreren Schritten (und Paketen) bestehenden Vergleich der beiden Bäume vornimmt. Um die Möglichkeit zu bieten, bei Bedarf andere Verfahren zum Vergleich der Bäume einzusetzen,

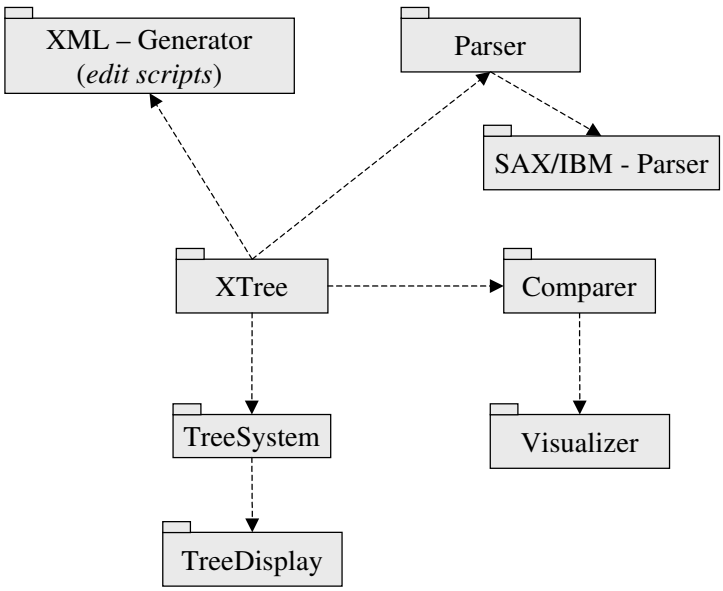

Abbildung 7.1: Grobarchitektur der Implementierung als Paketdiagramm

lassen sich die einzusetzenden `Comparer`-Pakete gegeneinander austauschen (siehe Abschnitt 7.4). Mit dem Ziel, eine Überprüfung der dort vorgenommenen Schritte zu ermöglichen, wurden Pakete zur Visualisierung dieser Vorgänge entwickelt, die das Paket `Visualizer` repräsentieren soll (siehe Abschnitt 7.7). Die Aufgabe, aus den im Vergleich ermittelten Differenzen der Bäume, einen *edit-script* in Form eines XML-Dokuments zu erzeugen, übernimmt das Paket `XML-Generator`. Es erhält einen *edit-script* als Folge von Objekten und erzeugt daraus ein XML-Dokument. Durch Austausch dieses Pakets ist es möglich, alternative Formate für die *edit-scripts* zu erzeugen.

Die nachfolgenden Teile beschreiben die Verarbeitung von XML-Dokumenten und Erzeugung von Bäumen aus diesen (Abschnitt 7.2), die Handhabung der zu vergleichenden Bäume als bipartite Graphen (Abschnitt 7.3), die wahlweise Verwendung verschiedener Algorithmengruppen (Abschnitt 7.4) und die Erzeugung von *edit-scripts* (Abschnitt 7.5). Der Abschnitt 7.6 geht kurz auf einige bei der Entwicklung zu berücksichtigende Hindernisse und Richtlinien ein. Die von dem entwickelten Dienst verwirklichte Visualisierung der einzelnen Arbeitsschritte wird im Abschnitt 7.7 beschrieben. Der Abschnitt 7.8 stellt schließlich einige Konfigurationsmöglichkeiten des Dienstes dar.

7.2 Ereignisgesteuerte Verarbeitung von XML-Dokumenten

Dieser Abschnitt beschreibt die Komponente der für das Einlesen von XML-Dokumenten verwendeten Klassen.

Die Verwendung des *Parsers XML Parser for JAVA* [IBM98a] in Verbindung mit der Schnittstelle *SAX* (*Simple API for XML*)[Meg99, Edd99b] legt die in Abbildung 7.2 dargestellte Klassenarchitektur nahe, bei der die Erzeugung eines Baumes zur Repräsentation eines XML-Dokuments ereignisorientiert erfolgt: Einem Objekt der Klasse `Parser` wird durch die Methode `setDocumentHandler()` ein Objekt übergeben, welches die Schnittstelle `DocumentHandler` mit den entsprechenden Methoden `startDocument()`, `startElement()` etc. implementiert. Dieses übergebene Objekt ist eine Instanz einer selbst zu entwickelnden Klasse `XHandler`, deren überschriebene Methoden von dem Objekt der Klasse `Parser` aufgerufen werden können. Auf diese Weise ist die rekursive

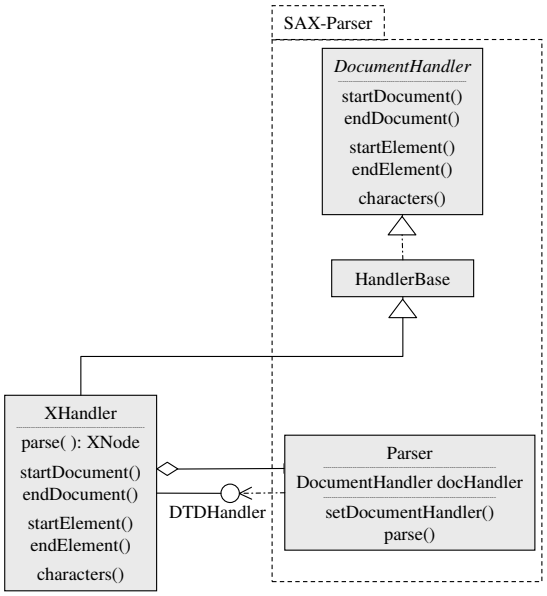

Abbildung 7.2: Klassendiagramm zur Darstellung der Beziehungen zwischen `DocumentHandler` und `Parser`.

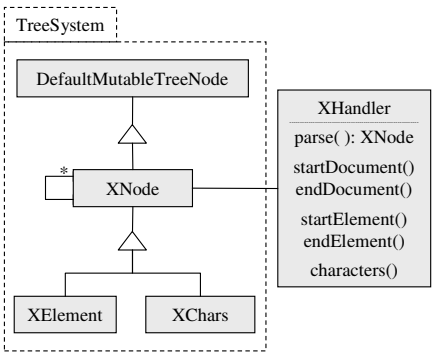

Abbildung 7.3: Klassendiagramm zur Darstellung der Beziehungen zwischen `DocumentHandler` und den Komponenten des zu erstellenden Baumes.

Erzeugung einer eigenen Baumstruktur gemäß Abbildung 7.3 möglich. Das Paket `SAX-Parser` faßt diejenigen Klassen und Schnittstellen zusammen, die von dem *XML Parser for JAVA* vorgegeben sind. Diese Vorgehensweise ist dem *Erbauer-Muster* sehr ähnlich [GHJV96, Edd99b]: Die Rolle des *Erbauers* wird von der Schnittstelle `DocumentHandler` und der Klasse `HandlerBase` übernommen. Die im Muster üblicherweise mit *Director* bezeichnete Klasse ist hier die Klasse `Parser`. Sie ruft entsprechende Methoden eines konkreten Erbauers (hier die Klasse `XHandler`) auf.

Der *Parser XML Parser for JAVA* bietet zusätzlich zu einer ereignisorientierten Verarbeitung von XML-Dokumenten, auch die des Erzeugens sogenannter *DOM-Bäume* (*DOM = Document Object Model*, [WWW00]), bei der jedoch nicht die Möglichkeit der Einflußnahme auf die Baumstruktur während des Vorgang des *Parsens* besteht, weshalb hier die Wahl auf das ereignisorientierte Verfahren fiel.

Der Aufruf der Methode `startElement()` verursacht die Erzeugung eines Elementknoten repräsentierenden Objektes der Klasse `XElement`. Dieses wird mit dem entsprechenden Elementnamen und Attributen (die der aufgerufenen Methode übergeben wurden) erzeugt und stellt das gerade „aktuelle“ Element-Objekt dar. Bei erneutem Aufruf der Methode `startElement()` wird erneut ein Element-Objekt erzeugt, dem „aktuellen“ Element-Objekt als Kind hinzugefügt und anstelle dessen seinerseits zum „aktuellen“ Element-Objekt gemacht. Bei Aufruf der Methode `characters()` wird ein Objekt der Klasse `XChars` erzeugt und dem „aktuellen“ Element-Objekt als Kind hinzugefügt. Bei Aufruf der Methode `endElement()` wird das Vater-Objekt des „aktuellen“ Element-Objektes wieder das aktuelle. Das in [GHJV96] beschriebene Kompositum-Muster stellt den allgemeinen Fall einer derartigen Struktur dar. Das Paket `TreeSystem` faßt diejenigen Klassen zusammen, die der Repräsentation der Baumknoten dienen.

7.3 Repräsentation und Bearbeitung bipartiter Graphen

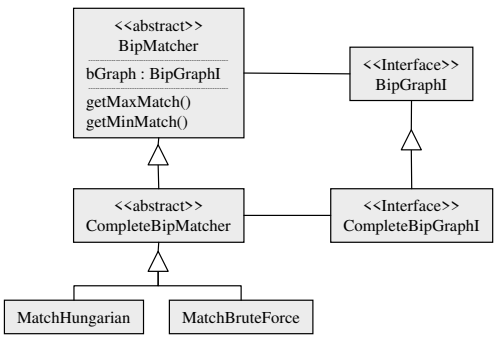

Abbildung 7.4: Klassendiagramm zur Darstellung der Beziehung zwischen den `BipMatcherI` implementierenden Klassen und den Klassen zur Repräsentation bipartiter Graphen.

In diesem Abschnitt wird die Komponente zur Repräsentation und Bearbeitung der bipartiten Graphen beschrieben, wie sie für die Lösung des Korrespondenzproblems und des Problems des Findens einer gewichtsminimalen Kantenüberdeckung benötigt werden.
Die Lösung des Korrespondenzproblems wird an mehreren Stellen des in Kapitel 6 beschriebenen Verfahren in unterschiedlichen Zusammenhängen benötigt. Das Klassendiagramm aus Abbildung 7.4 zeigt die Definition einer Schnittstelle zur Repräsentation bipartiter (beziehungsweise vollständiger bipartiter) Graphen. Die jeweiligen Zusammenhänge werden damit von der Lösung des Korrespondenzproblems entkoppelt. Die Eigenschaften eines bipartiten Graphen werden durch die Schnittstelle `BipGraphI` beschrieben, deren Verfeinerung `CompleteBipGraphI` die Eigenschaften

eines vollständigen bipartiten Graphen beschreibt. Die Erweiterung der Schnittstelle `BipGraphI` zur Schnittstelle `CompleteBipGraphI` stellt genaugenommen keine wirkliche Erweiterung dar (es kommen keine Methodenbezeichner hinzu), sondern soll lediglich dazu dienen, daß Methoden, die einen vollständigen bipartiten Graphen als Parameter erwarten, einen solchen erhalten. Während die abstrakte Klasse `BipMatcher` eine Basis für das Lösen des Korrespondenzproblems darstellt, beschreibt ihre Subklasse `CompleteBipMatcher` das Korrespondenzproblem eines vollständigen bipartiten Graphen. Sie überprüft den ihr übergebenen Graph daraufhin, ob es sich tatsächlich um einen vollständigen bipartiten Graphen handelt. Generell sind eine Reihe von Verfahren denkbar, die dieses Problem lösen, so daß jedes dieser Verfahren durch eine eigene Klasse repräsentiert werden kann. Das in Kapitel 6 beschriebene Verfahren verwendet den *Ungarischen Algorithmus* und damit die Klasse `MatchHungarian`. Die sich im Klassendiagramm befindende Klasse `MatchBruteForce` implementiert das Verfahren, das sämtliche Kombinationen durchprobiert, um schließlich die günstigste wählen zu können.

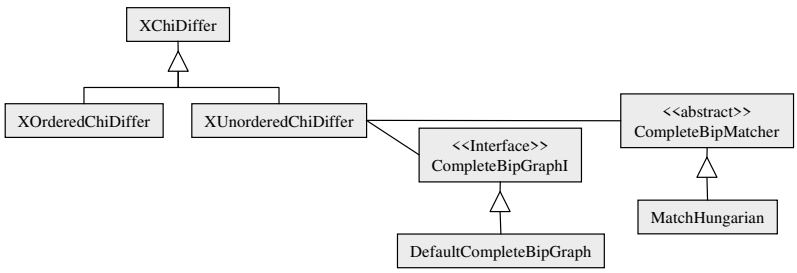

Abbildung 7.5: Klassendiagramm zur Darstellung der Verwendung der Klassenstruktur aus Abbildung 7.4 bei der Ermittlung des Nachfolgergewichts $chi(e)$ einer Kante e

Die Verwendung dieser Komponente wird in Abbildung 7.5 dargestellt: die Aufgabe eines Objektes der Klasse `XChiDiffer` ist die Erstellung einer Zuordnung der Kinder zweier Knoten. In Abhängigkeit davon, ob die zu vergleichenden Bäume als geordnete oder ungeordnete Bäume zu betrachten sind, ergeben sich zwei Verfeinerungen dieser Klasse. Bei dem Vergleich ungeordneter Bäume werden die Kinder der beiden oben genannten Knoten durch einen vollständigen bipartiten Graphen dargestellt. Eine Korrespondenz der Kinder wird durch ein Objekt der Klasse `CompleteBipMatcher` ermittelt. Die für den Vergleich geordneter Bäume verwendete Klasse `XOrderedChiDiffer` wird hier nicht beschrieben.

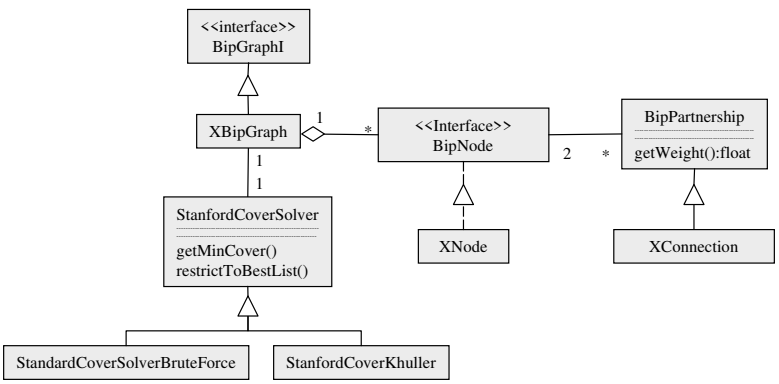

Abbildung 7.6: Klassendiagramm zur Darstellung der Beziehungen zwischen den Klassen zur Lösung des Kantenüberdeckungsproblems.

Abbildung 7.6 zeigt ein Klassendiagramm zur Darstellung der Beziehungen zwischen den Klassen zur Lösung des Kantenüberdeckungsproblems (`StanfordCoverSolver`), dem betreffenden bipartiten Graph (`XBipGraph`) sowie den Knoten (`BipNode`) und Kanten (`BipPartnership`), aus denen der bipartite Graph besteht. Die beiden miteinander zu vergleichenden Bäume werden durch Objekte der Klasse `XNode` (oder deren Subklassen) repräsentiert. Dadurch, daß die Klasse `XNode` die Schnittstelle `BipNode` und die Klasse `XConnection` die Schnittstelle `BipPartnership` implementieren, ist es möglich, daß Klassen, wie `XBipGraph` oder `StanfordCoverSolver` die Knoten und deren Verbindungen als `BipNode`- und `BipPartnership`-Objekte betrachten und dennoch das Gewicht eines `XConnection`-Objektes mit Hilfe der Methode `getWeight()` ermitteln können. Die Basisklasse derjenigen Klassen, die das Kantenüberdeckungsproblem[1] lösen sollen, ist mit `StanfordCoverSolver` bezeichnet, da dieses Problem in [CGM97] beschrieben wird. Die Subklassen `StanfordCoverSolverBruteForce` und `StanfordCoverKhuller`[2] implementieren jeweils ein Verfahren zur Lösung des Kantenüberdeckungsproblems, wobei `StanfordCoverSolverBruteForce` alle Kantenkombinationen durchprobiert, um schließlich die günstigste zu übernehmen und die Klasse `StanfordCoverKhuller` das in Abschnitt 4.3.2 beschriebene Verfahren anwendet, mit dem das Kantenüberdeckungsproblem auf ein Korrespondenzproblem reduziert wird.

7.4 Austauschbare Algorithmen zum Baumvergleich

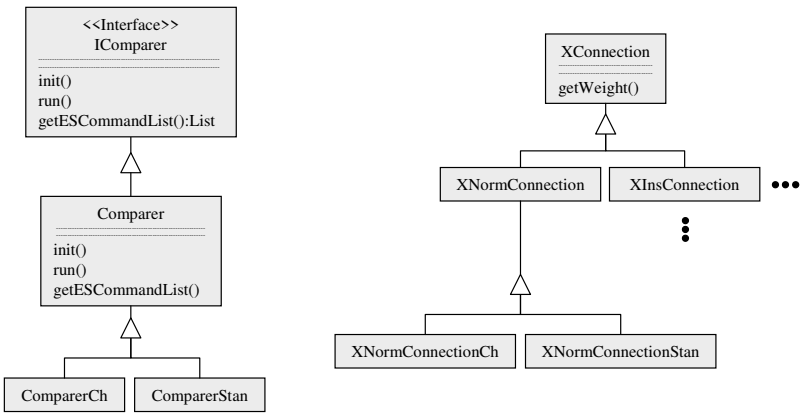

Abbildung 7.7: Klassendiagramm zur Repräsentation der Komponente, die eine wahlweise Verwendung verschiedener Algorithmengruppen ermöglicht

Das in Abbildung 7.7 dargestellte Klassendiagramm gibt einen groben Überblick über einen Ausschnitt der Komponente, die es ermöglicht, wahlweise verschiedene Algorithmengruppen für den Vergleich von Bäumen anzuwenden. Für jede der Algorithmengruppen gibt es eine Klasse, die von der Klasse `Comparer` erbt und damit die Schnittstelle `IComparer` implementiert. Die abstrakte Klasse `Comparer` legt die für alle ihre Subklassen gemeinsame Funktionalität fest, wie der prinzipielle Ablauf der Teilschritte. Auf der anderen Seite gibt es für jede von `XConnection` erbende Klasse jeweils für jede Algorithmengruppe eine Subklasse, die speziell auf das jeweilige Verfahren zugeschnitten ist. Die Klasse `XConnection` repräsentiert die Verbindung zweier durch Objekte der Klasse `XNode` repräsentierten Knoten der beiden Bäume. In dieser Klasse wird das allen Subklassen gemeinsame Verhalten festgelegt (beispielsweise Methoden zum Navigieren bzw. Auffinden der Verbindungspartner). Die Subklassen `XNormConnection`, `XInsConnection`, `XDelConnection`

[1] genauer: das Problem, eine gewichtsminimale Kantenüberdeckung eines gewichteten bipartiten Graphen zu finden.

[2] Das von Khuller und Raghavachari geschriebene Kapitel in [Ata99] beschreibt die von der Klasse verwendete Vorgehensweise.

und `XInsDelConnection` legen das Verhalten der jeweiligen Verbindungstypen fest, während deren jeweilige verfahrensspezifische Subklassen (z.B. `XNormConnectionCh`) genau festlegen, wie beispielsweise das Gewicht der Verbindung ermittelt wird. Das Verhalten der algorithmusspezifischen Klassen ist sehr unterschiedlich für die jeweiligen Verbindungstypen (`XNormConnection` vs. `XInsConnection` etc.), so daß die in *JAVA* ohnehin nicht verwendbare Mehrfachvererbung an dieser Stelle nicht hilfreich wäre.

7.5 Generierung des *edit-scripts*

Dieser Abschnitt beschreibt die Komponente, die für die Erzeugung des *edit-scripts* verantwortlich ist. Jedem `XConnection`-Objekt ist ein Objekt zugeordnet, das die Schnittstelle `IConnScriptor` implementiert (Abbildung 7.8). Letzteres übernimmt die Erzeugung einer entsprechenden Operation des *edit-scripts*. Für jeden Beschriftungstyp einer Verbindung `XConnection` (*"nil", "ins", "del", "mov", "cpy"* und *"glu"*, Abschnitt 6.5.2) gibt es jeweils eine Subklasse von `ConnScriptorCh`. Deren Erzeugung wird in Abhängigkeit der Beschriftung des jeweiligen `XConnection`-Objektes von einer Fabrik-Klasse `ConnScriptorFactoryCh` übernommen.

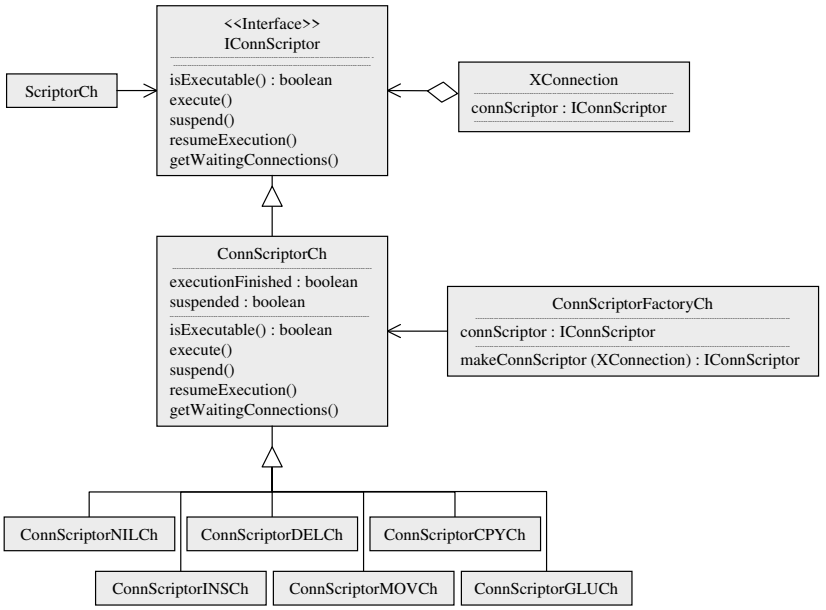

Abbildung 7.8: Klassendiagramm zur Generierung des *edit-scripts* I

Die Handhabung der `ConnScriptorXXXCh`-Objekte (`XXX` steht für `nil, ins, del, mov, cpy` oder `glu`) erfolgt über die Klasse `ScriptorCh`. Sie sucht die nächste, behandelbare Verbindung (`XConnection`-Objekt) und versucht, für diese Verbindung eine Operation des *edit-scripts* erstellen zu lassen (durch den zu der betreffenden Verbindung gehörenden `ConnScriptor`). Hierfür bietet ein `ConnScriptor` durch Implementierung der Schnittstelle `IConnScriptor` Methoden zum Testen auf Ausführbarkeit (`isExecutable()`), zum Ausführen (`execute()`), zum Zurückstellen und Wiederaufsetzen der Ausführung (`suspend()`, `resume()`) etc. an (siehe Abschnitt 6.5.3).

In dem in Abbildung 7.9 dargestellten Klassendiagramm wird die Beziehung zwischen der `ScriptorCh`-Klasse und den Klassen zur Repräsentation jeweils einer Operation des *edit-scripts* gezeigt. Ein Objekt der Klasse `ScriptorCh` verwaltet eine Liste von Objekten der Klasse `ESCommand`, die im einzelnen Objekte von `ESCommand` erbenden Klassen, wie `ESUpdateCommand` sind. Jede dieser erbenden Klassen überschreibt Methoden zum Ermitteln einer zu der jeweiligen Operation

passenden XML-Sequenz, wie sie im Anhang B angegeben sind. Ebenfalls von diesen Klassen implementiert werden Methoden, die die entsprechende Operation auf einen Baum (gegeben durch die Wurzel des Baumes) anwenden.

Ein *edit-script*, der die Differenz zweier Dokumente beschreibt, wird durch ein XML-Dokument (ein *edit-script*-Dokument) repräsentiert. Mit dem Ziel, nur mit Hilfe so eines Dokuments das Basisdokument schrittweise in das Zieldokument überführen zu können, muß aus dem *edit-script*-Dokument eine Folge von Objekten der Klasse `ESCommand` (bzw. deren Subklassen) erzeugt werden, die dann auf das Basisdokument angewendet werden können. Wie in Abbildung 7.9 gezeigt, enthält die Klasse `ESCommands` eine Methode `fromXNode`, die ein Objekt der Klasse `ESCommands` (bzw. einer ihrer Subklassen) auf Grundlage eines Objektes der Klasse `XNode` erstellt. Auf diese Weise lassen sich die notwendigen Operationen-Objekte erzeugen, die durch die Verwendung ihrer jeweiligen `apply`-Methoden den Basisbaum schrittweise in den Zielbaum überführen können.

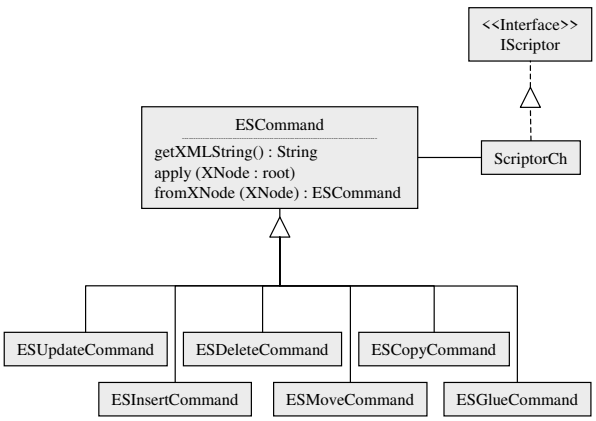

Abbildung 7.9: Klassendiagramm zur Generierung des *edit-scripts* II

7.6 Hindernisse und Richtlinien

Nachdem in den vorangegangenen Abschnitten die Architektur einiger Ausschnitte der Implementierung dargestellt wurden, wird in diesem Abschnitt kurz auf einige bei der Entwicklung zu überwindende Hindernisse sowie auf einige Richtlinien eingegangen, die es einzuhalten galt.

Der letztlich nicht erfolgreiche Versuch, das in [CGM97] beschriebene Verfahren *MH-Diff* zu implementieren führte schließlich einerseits zu dem Entwurf und der Implementierung eines eigenen Verfahrens und andererseits zu einer Architektur, die eine wahlweise Verwendung verschiedener Algorithmengruppen ermöglicht, wie es in Abschnitt 7.4 dargestellt wurde.

Bei der Erzeugung des *edit-scripts* aus der beschrifteten gewichtsminimalen Kantenüberdeckung ist die Reihenfolge der zu bearbeitenden Kanten zu beachten, da die Bearbeitung einiger Kanten die abgeschlossene Bearbeitung anderer Kanten voraussetzt (siehe Abschnitt 6.5.3). In der realisierten Klassenstruktur übernimmt für jede beschriftete Verbindung ein Objekt mit der Schnittstelle `IConnScriptor` die Erzeugung einer Operation des *edit-scripts* (siehe Abschnitt 7.5). Gemäß dem Beobachter-Muster [GHJV96] reihen sich wartende `IConnScriptor`-Objekte bei den `IConnScriptor`-Objekten in eine Warteschlange ein, auf die sie warten müssen, um nach deren vollständiger Bearbeitung benachrichtigt zu werden.

7.7 Visualisierung

Obgleich es sich bei der im Rahmen dieser Arbeit entwickelten Implementierung um einen Dienst handelt, der ohne graphische Benutzerschnittstelle auskommen kann, hat es sich als sinnvoll erwiesen, dennoch eine graphische Benutzerschnittstelle zu entwickeln. Sie dient der Kontrolle der im Rahmen der Konfigurierung vorgenommenen Einstellungen und deren Auswirkungen auf das Verfahren, der Überprüfung der Richtigkeit der Algorithmen, der Fehlersuche und -beseitigung bei der Entwicklung und schließlich auch der Demonstration des Dienstes.

Dieser Abschnitt beschreibt die von dem entwickelten Dienst zur Verfügung gestellten Visualisierungen der einzelnen Arbeitsschritte.

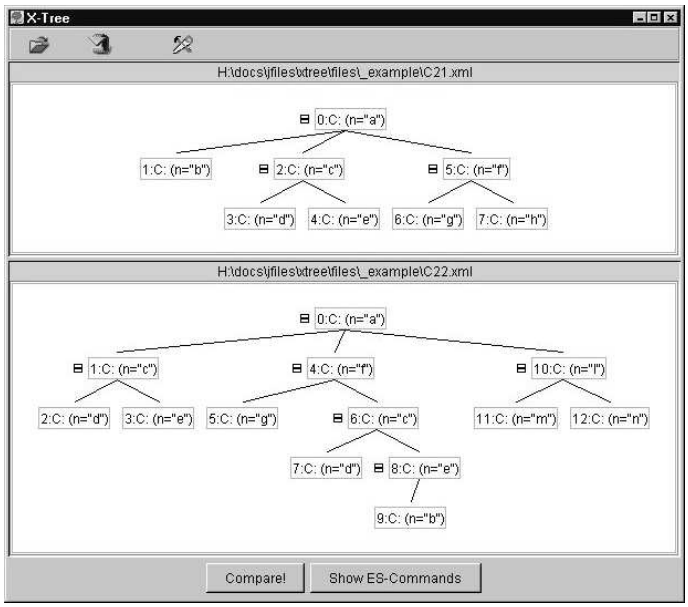

Abbildung 7.10: Der Dienst mit zwei miteinander zu vergleichenden Bäumen

Der Dienst läßt sich als Applikation „XTree“ starten und erscheint in der in Abbildung 7.10 dargestellten Form. Die Schaltflächen der oberen Schaltflächenleiste (*toolbar*) dienen dem Öffnen zweier miteinander zu vergleichenden XML-Dokumente, dem erneuten Einlesen eingelesener XML-Dokumente (*reload* - wenn diese beispielsweise von anderen Programmen modifiziert wurden) und dem Öffnen eines Konfigurationsdialogs.

Die miteinander zu vergleichenden XML-Dokumente werden graphisch als horizontal gezeichnete Bäume im Hauptteil der Applikation angezeigt. Am unteren Rand befinden sich schließlich zwei Schaltflächen zum Starten des Vergleichs und zur Darstellung der schrittweisen Abarbeitung des *edit-scripts*.

Nach Betätigung der Schaltfläche „compare!“ wird der Vergleich der Bäume gestartet und es öffnen sich drei weitere Fenster, die den während des Vergleichs verwendeten bipartiten Graphen in drei Situationen darstellen, die die Ergebnisse der Teilschritte des Verfahrens darstellen:

- der bipartite Graph, dessen Kanten jeweils mit denen aus den verschiedenen Komponenten zusammengesetzten Gewichten versehen sind (Abbildung 7.11)
- die gefundene gewichtsminimale Kantenüberdeckung, deren Kanten noch nicht beschriftet sind (ohne Abbildung)

- die gewichtsminimale Kantenüberdeckung, deren Kanten mit den in Abschnitt 6.5.2 beschriebenen Beschriftungen versehen sind (Abbildung 7.12)

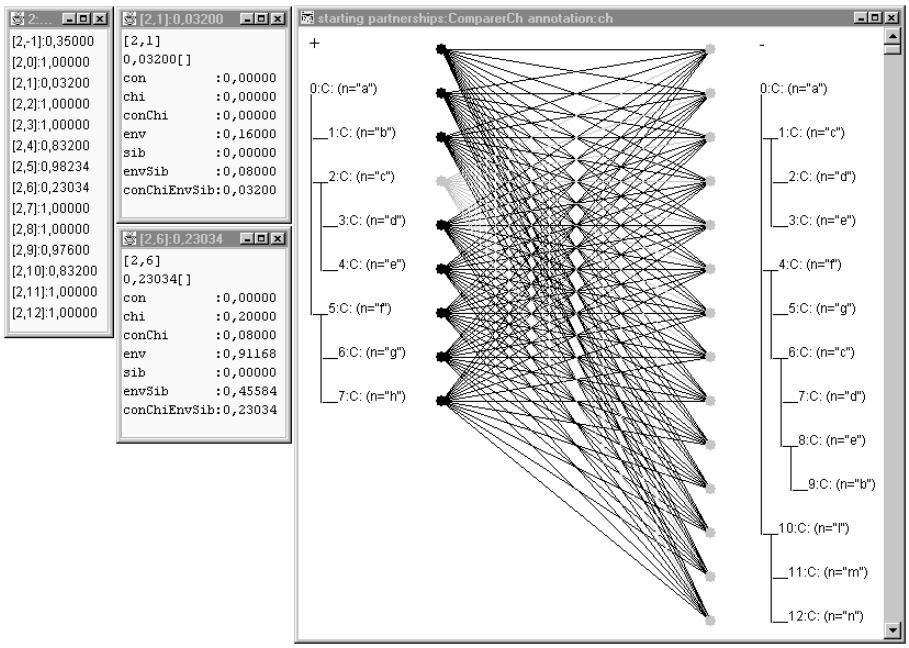

Abbildung 7.11: Ein von dem Dienst erzeugter vollständiger bipartiter Graph dessen Kanten jeweils ein Gewicht bestehend aus den vier Komponenten `con, chi, env, sib` und deren Kombinationsgewichten haben

Die Abbildungen 7.11 und 7.12 zeigen zusätzlich drei Fenster, von denen das linke jeweils sämtliche Verbindungen auflistet, die den Knoten mit der Nummer (2) des linken Baumes treffen. Ein solches Fenster wird geöffnet, indem mit der Maus der entsprechende Knoten in der Darstellung des bipartiten Graphen angeklickt wird. Zu jeder Verbindung werden die Knotennummern der Verbindungspartner, das Gesamtgewicht der Verbindung und (bei dem beschrifteten bipartiten Graphen) die Beschriftung dargestellt. Es lassen sich aus dieser Liste einzelne Verbindungen auswählen, die in gesonderten Fenstern genauer dargestellt werden. In beiden Abbildungen ist das für die Verbindungen, die den Knoten mit der Nummer (2) des linken Baumes mit den Knoten mit den Nummern (1) und (6) des rechten Baumes verbinden, gezeigt. Diese Fenster lassen sich öffnen, indem man mit der Maus in die Nähe der darzustellenden Kante klickt, oder in dem Fenster zur Darstellung aller Verbindungen eines Knoten auf eine der Zeilen klickt. Mit Hilfe dieser Fenster lassen sich die einzelnen Gewichtskomponenten der Verbindungen genau inspizieren.

Durch Betätigung der Schaltfläche „Show ES-Commands“ im Startfenster (Abbildung 7.10) wird ein Programmteil gestartet, der die schrittweise Anwendung des *edit-scripts* auf den Basisbaum darstellt (Abbildungen 7.13 bis 7.17). Hierzu öffnet sich ein Fenster, dessen oberer Teil den erreichten Bearbeitungszustand des Baumes wiedergibt, während sich im unteren Teil Schaltflächen zur Ausführung der jeweils nächsten Operation des *edit-scripts* und zum Neustart der Abarbeitung befinden. Unmittelbar über den Schaltflächen wird durch ein Textfeld die nächste auf den Baum anzuwendende Operation angekündigt. Im mittleren Abschnitt des Fensters wird bei bevorstehender *insert*-Operation der einzufügende Teilbaum dargestellt (siehe Abbildung 7.16).

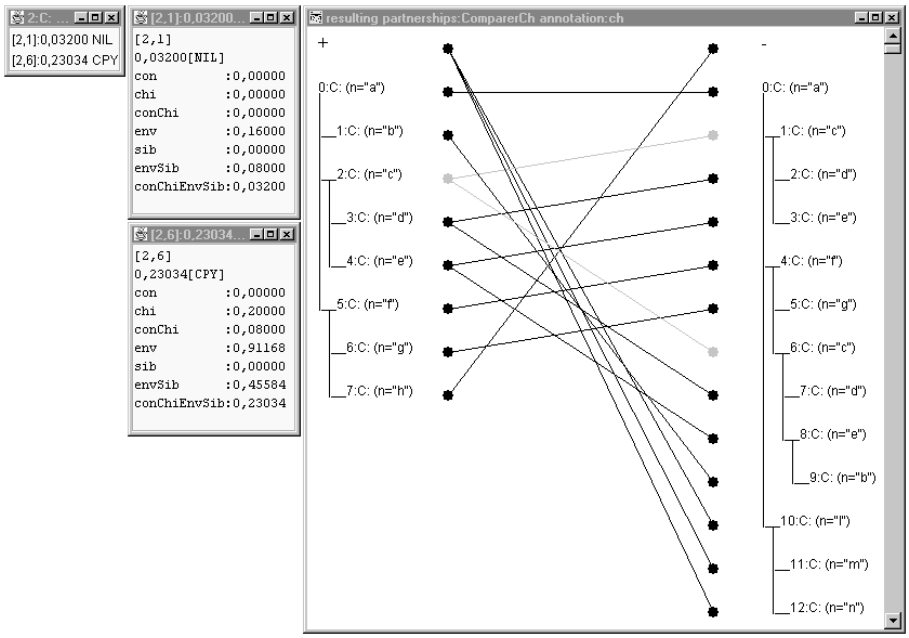

Abbildung 7.12: Die von dem Dienst erzeugte gewichtsminimale Kantenüberdeckung, deren Kanten zusätzlich beschriftet wurden

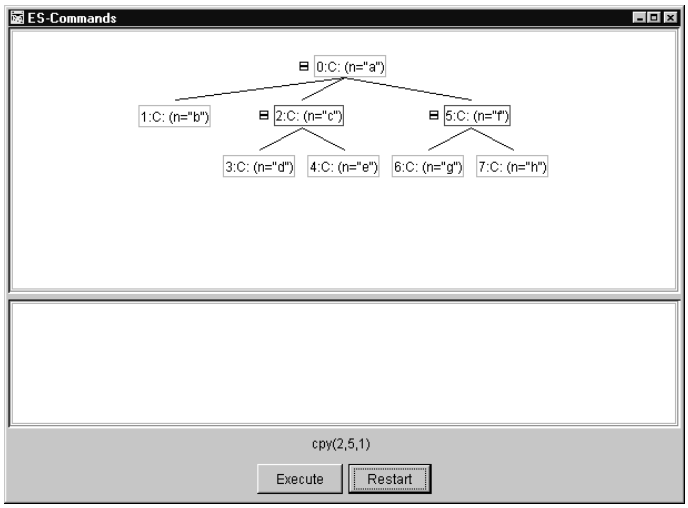

Abbildung 7.13: Visualisierung der schrittweisen Anwendung des erzeugten *edit-scripts*: *copy*-Operation

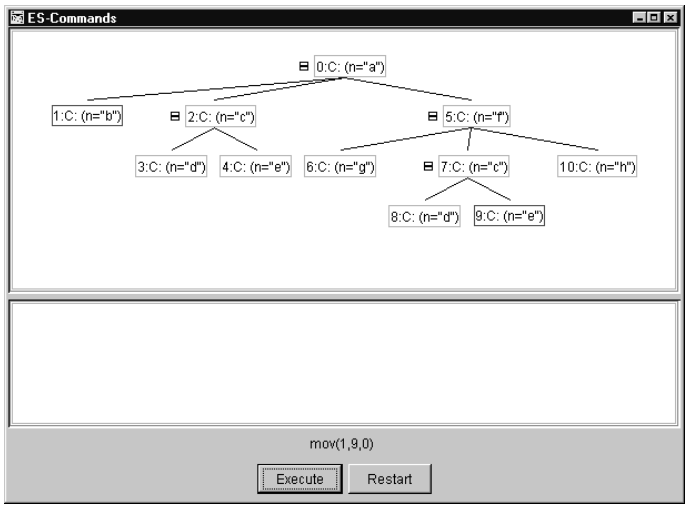

Abbildung 7.14: Visualisierung der schrittweisen Anwendung des erzeugten *edit-scripts*: *move*-Operation

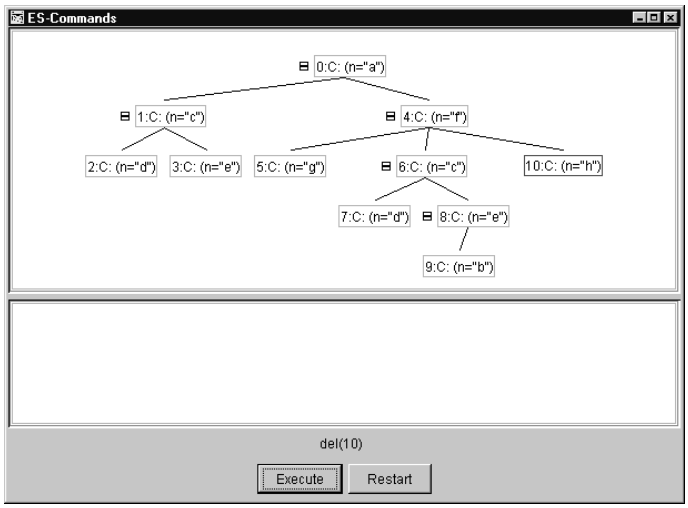

Abbildung 7.15: Visualisierung der schrittweisen Anwendung des erzeugten *edit-scripts*: *delete*-Operation

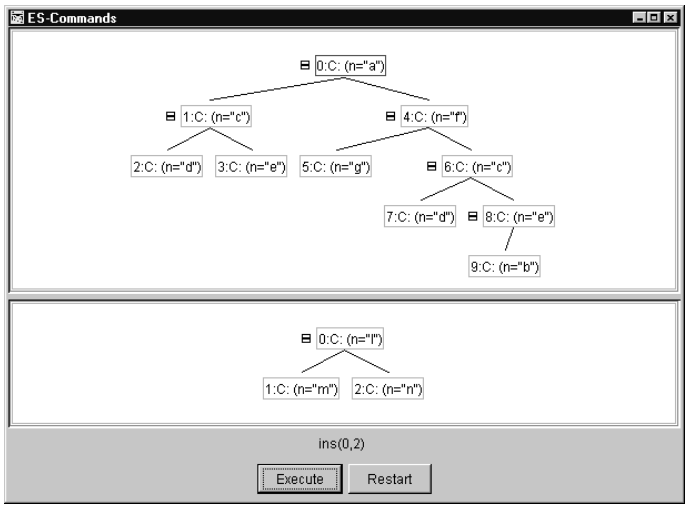

Abbildung 7.16: Visualisierung der schrittweisen Anwendung des erzeugten *edit-scripts*: *insert*-Operation

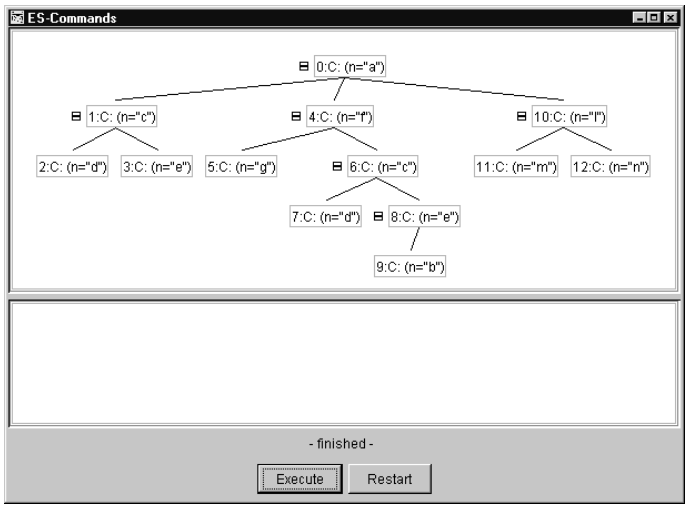

Abbildung 7.17: Visualisierung der schrittweisen Anwendung des erzeugten *edit-scripts*: nach der Ausführung der letzten Operation

7.8 Konfiguration des Dienstes

Der im Rahmen dieser Arbeit entstandene Dienst zum Vergleich von XML-Dokumenten läßt sich in Bezug auf die drei Kategorien *Administration*, *Visualisierung* und *Algorithmus* konfigurieren. Im Folgenden sollen diese drei Kategorien und deren Konfigurationsmöglichkeiten kurz beschrieben werden. Wie bei der Arbeit mit der Programmiersprache *JAVA* üblich, wird die jeweils aktuelle Konfiguration des Dienstes in Form sogenannter *properties* gehalten und in einer Datei gesichert. Die Modifikation der Einstellungen erfolgt mit Hilfe des in Abbildung 7.18 dargestellten Konfigurationsdialogs.

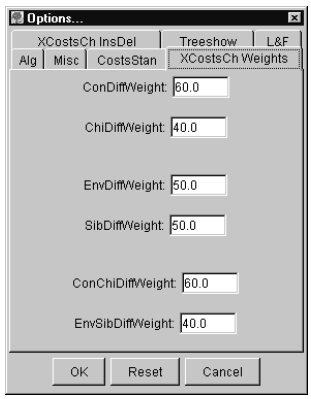

Abbildung 7.18: Konfigurationsdialog

Administrative Konfiguration: Um relative Dateizugriffe vornehmen zu können, benötigt der Dienst die Beschreibung des Verzeichnisses, in dem er sich selber befindet. Ebenso wird eine Beschreibung des Verzeichnisses benötigt, in dem die Ergebnisse eines Vergleichs in Form von XML-Dokumenten abgelegt werden sollen. Diese Einstellungen können über die administrative Konfiguration festgelegt werden.

Konfiguration der Visualisierung: Für die Darstellung der beiden miteinander zu vergleichenden Bäume verwendet das Programm Einträge darüber, ob Attribute der XML-Knoten dargestellt werden sollen, die Anzahl der anzuzeigenden Zeichen der Textkörper und Elementnamen. Für die Anzeige der bipartiten Graphen können Angaben über Positionen, Abstände und Knotendurchmesser der Knoten der Graphen gemacht werden. Darüberhinaus kann bestimmt werden, ob die Verbindungen des beschrifteten, gewichtsminimalen bipartiten Graphen ihren Beschriftungen entsprechend gefärbt werden sollen.

Konfiguration der Algorithmen: Die den Vergleichsalgorithmen zugrundeliegende Klassenarchitektur läßt den wahlweisen Einsatz von Algorithmen zu. Prinzipiell ist es möglich, jeweils für die verschiedenen Schritte des Verfahrens sowohl zwischen verschiedenen Algorithmengruppen zu wählen als auch innerhalb einer Algorithmengruppe spezifische Einstellungen vorzunehmen:

- Prinzipiell läßt sich zwischen mehreren Verfahren wählen, wobei jedes Verfahren durch eine Algorithmengruppe repräsentiert wird. Bis jetzt ist jedoch nur eines der Verfahren implementiert. Für dieses läßt sich einstellen, ob die miteinander zu vergleichenden Bäume als geordnet oder ungeordnet zu betrachten sind.
- Das im Abschnitt 6.5.1 beschriebene Verfahren sieht eine Zuordnung von Knoten der miteinander zu vergleichenden Bäume vor, die auf Grundlage der Ähnlichkeiten zwischen den Knoten ermittelt wird. Diese Ähnlichkeiten werden in Form von Gewichten ausgedrückt, die sich aus mehreren Komponenten zusammensetzen, deren jeweiliger Beitrag zum Gesamtgewicht konfigurierbar ist (siehe Abbildung 7.18).

- Bei der Zuordnung von Elementknoten wurden in Abschnitt 6.5.1 zwei Verfahren besprochen („analoges“ versus „digitales“ Verfahren), zwischen denen gewählt werden kann.
- Schließlich läßt sich auch die Ermittlung der Gewichte von *Plus*- und *Minus*-Kanten konfigurieren, indem die in Abschnitt 6.5.1 beschriebenen Mindestgewichte der *Plus*- und *Minus*-Kanten, sowie die zu diesen hinzukommenden Gewichtsteile definiert werden.

Kapitel 8

Zusammenfassung und Ausblick

Dieses abschließende Kapitel liefert eine Zusammenfassung und eine Beurteilung der im Rahmen dieser Arbeit vorgenommenen Entwürfe und Implementierungen, sowie einen Ausblick auf mögliche Erweiterungen und Verbesserungen der prototypischen Implementierung.

Die von Digitalen Bibliotheken zur Verfügung gestellten Informationsressourcen sind als dynamisch zu betrachten. Sie unterliegen Veränderungen, die auch die Metadaten betreffen. Dieses ist von den Betreibern einer Digitalen Bibliothek zur Erreichung des Ziels, den Informationskonsumenten aktuelle Informationen zur Verfügung zu stellen, zu berücksichtigen. Hierfür stellen Dienste zum Change Management ein geeignetes Mittel dar. Die Aufgabe dieser Systeme besteht neben der Erlangung bzw. Erzeugung möglichst aktueller Metadaten und der Benachrichtigung der an den Differenzen interessierten Personen und Systeme vor allem darin, stattgefundene Veränderungen der Metadaten zu erkennen und diese den Informationskonsumenten in geeigneter Form zur Verfügung zu stellen (vgl. Kapitel 2).

Es existieren bereits einige Dienste zu diesem Thema. Sie untersuchen Informationsbestände mit dem Ziel, deren Differenzen aufzuzeigen. Eine Klassifizierung dieser Dienste ist nach unterschiedlichen Kriterien möglich, die auch die unterschiedliche Eignung der Dienste zum Change Management Digitaler Bibliotheken begründen (vgl. Kapitel 3).

In Abhängigkeit der betrachteten Struktur der miteinander zu vergleichenden Informationsbestände ergeben sich auf der Ebene der Algorithmen die beiden Problemklassen des Vergleichs von Sequenzen und des Vergleichs beschrifteter Bäume. Letztere Klasse läßt sich weiter in den Vergleich geordneter und ungeordneter Bäume unterteilen. Für den Vergleich von Metadaten eignet sich das Modell der ungeordneten Bäume. Da es sich bei dem Vergleich ungeordneter Bäume um ein NP-vollständiges Problem handelt, bieten sich zur Lösung Heuristiken an. Das Ergebnis des Vergleichs von Informationsbeständen läßt sich durch sogenannte *edit-scripts* ausdrücken, die als Folgen von Änderungsoperationen, den einen der Informationsbestände schrittweise in den anderen umformen (vgl. Kapitel 4).

Mit dem Ziel, einen generischen Dienst für den Vergleich semistrukturierter Informationen zu erstellen, wurde ein Verfahren entwickelt, das in groben Zügen einem in Kapitel 4 beschriebenen Algorithmus entstammt, dessen Teilschritte jedoch durch eigene ersetzt. Die hierbei verwendete Schnittstelle verwendet XML sowohl für den Austausch der miteinander zu vergleichenden Metadaten als auch für die Repräsentation der *edit-scripts* (vgl. Kapitel 5 und 6).

Bei der Implementierung des entwickelten Verfahrens wurde besonderer Wert auf die Konfigurierbarkeit, die Erweiterbarkeit, die Wiederverwendbarkeit und die Nachvollziehbarkeit gelegt. Dabei ist ein generischer Dienst entstanden, der zwei XML-Dokumente miteinander vergleicht und deren Unterschiede in Form eines *edit-scripts* darstellt, der ebenfalls durch ein XML-Dokument reprä-

sentiert wird und durch Komponenten zur Visualisierung nachvollziehbar dargestellt wird (vgl. Kapitel 7). Mit Hilfe der Komponenten zur Visualisierung läßt sich die Arbeitsweise des Verfahrens und die Modifikation des Basisbestandes durch den *edit-script* schrittweise nachvollziehen.

8.1 Bewertung des entwickelten Systems

Auf der Grundlage der in der Einleitung angegebenen Zielsetzungen in Verbindung mit dem in Abschnitt 2.3 erarbeiteten Kriterienkatalog erfolgt in diesem Abschnitt eine Bewertung des entwickelten Dienstes.

Aus der Klassifizierung und Diskussion existierender Dienste und Verfahren ist der Entwurf des eigenen Verfahren mit anschließender Implementierung als ein Dienst für das Change Management hervorgegangen. Dieser vergleicht semistrukturierte Daten zweier Informationsbestände miteinander und ermittelt eine Folge von Änderungsoperationen, die beschreibt, wie der eine Informationsbestand aus dem anderen hervorgehen kann. Die Ergebnisse dieses Vergleichs werden durch *edit-scripts* repräsentiert, die aufgrund des ihnen zugrundeliegenden Operationen-Repertoires und ihrer Kompaktheit aussagekräftig sind.

Die Tatsache, daß es sich bei den miteinander zu vergleichenden Informationsbeständen um semistrukturierte Daten handelt, wird von dem entworfenen Dienst berücksichtigt. Die Informationsbestände werden als beschriftete Bäume, einer alternativen Repräsentation für semistrukturierte Daten, aufgefaßt, die miteinander verglichen werden. Die erstellten *edit-scripts* beziehen sich in ihren Operationen auf die Struktur der beschrifteten Bäume (Einfügen von Teilbäumen, Entfernen von Teilbäumen, Verschieben von Teilbäumen etc).

Der vorliegende Dienst ist als Bestandteil eines größeren Systems zum Change Management entwickelt, weshalb für die Ein- und Ausgabedaten des Dienstes ein standardisiertes Format (XML) verwendet wird. Die tatsächliche Benachrichtigung anderer Systeme oder Anwender muß durch die Verwendung weiterer Dienste erfolgen (etwa im Rahmen eines *Notification Frameworks*, wie es in einer am Arbeitsbereich Softwaresysteme aktuell erstellten Diplomarbeit beschrieben wird).

Durch die Verwendung von XML für den Austausch der Informationen sowohl für die miteinander zu vergleichenden Informationsbestände als auch für die von dem Verfahren ermittelten *edit-scripts* stellt der Dienst einen generischen Beitrag für das Change Management dar. Zur Generizität trägt zusätzlich die Tatsache bei, daß Komponenten des Dienstes austauschbar und durch eine Reihe von Parametern konfigurierbar sind.

Die von dem Dienst erzeugten *edit-scripts* stellen dar, wie eines der XML-Dokumente durch schrittweise Anwendung des *edit-scripts* aus dem anderen hervorgehen kann. Die dabei entstehenden Zwischenversionen stellen in Bezug auf die Dokumententypdefinition, die den ursprünglichen Dokumenten zugrundeliegt, nicht unbedingt gültige Dokumente dar. Andererseits entstehen auch bei manuellen Modifikationen zwischenzeitlich ungültige Dokumente. Entscheidend ist, daß die ursprünglich verglichenen Dokumente der Dokumententypdefinition entsprechen.

Trotz umfangreicher Implementierung hat das entwickelte System Prototypcharakter. Im Sinne des objekt- und prototyping orientierten Life-Cycle-Modells [PB93] sind noch einige Zyklen nötig, bis der Dienst tatsächlich eine ausreichende Produktreife erzielt. Das Kriterium der *Performance* wird von dem entwickelten Dienst noch nicht optimal erfüllt, obwohl bei der Auswahl der Algorithmen und deren Implementierung bereits auf Effizienz Wert gelegt wurde.

Mit Hilfe der Visualisierungskomponenten läßt sich die Arbeitsweise des Dienstes und der von ihm implementierten Algorithmen gut nachvollziehen. Ferner erleichtern sie die Konfigurierbarkeit und ermöglichen eine Demonstration des Dienstes.

8.2 Ausblick

Dieser Abschnitt zeigt zunächst einige weiterführende Verwendungen der in dieser Arbeit gewonnenen Erkenntnisse und des entwickelten Dienstes auf und bearbeitet anschließend einige Möglichkeiten zur Erweiterung.

8.2.1 Weiterführende Arbeiten

Bei der Arbeit mehrerer Personen an einem Informationsbestand besteht die Notwendigkeit, eine Koordination der parallel stattfindenden Arbeiten vorzunehmen, um unerwünschte Effekte parallelen Arbeitens zu vermeiden. Wird dabei aus Gründen der Autonomie der beteiligten Personen und der Vermeidung von Blockierungen auf striktes Sperren verzichtet, entsteht das Problem, parallel veränderte Informationsbestände zusammenzuführen. Da dieser Vorgang aus Gründen der Qualitätssicherung nicht automatisch vorgenommen werden soll, wird im Rahmen einer Studienarbeit am Arbeitsbereich Softwaresysteme ein Werkzeug entwickelt, das den halbautomatischen Prozeß des Zusammenführens unterschiedlicher Versionen unterstützt.

Der im Rahmen dieser vorliegenden Arbeit entwickelte Dienst stellt für ein derartiges System eine wichtige Grundlage dar. Mit Hilfe zweier *edit-scripts*, welche die Unterschiede der beiden parallel entwickelten Versionen A_1 und A_2 jeweils zu dem ursprünglichen Informationsbestand A beschreiben, lassen sich die jeweiligen Änderungen schrittweise annehmen oder zurückweisen. Das Ergebnis dieses Prozesses ist dann eine Version A', die Änderungen aus beiden A_1 und A_2 in Bezug auf die ursprüngliche Version A enthält.

Weitere Anwendungen des Dienstes werden im Abschnitt 1.2.2 erwähnt.

8.2.2 Erweiterungen

Für den realisierten Dienst sind eine Reihe von Erweiterungen denkbar, von denen hier exemplarisch fünf diskutiert werden.

Die Veränderungen von Dokumenten erfolgt häufig dokumententypspezifisch. So wird bei der Erstellung eines umfangreicheren Textdokuments zunächst ein Titel entworfen, eine Gliederung erarbeitet, die einzelnen Kapitel und deren Unterkapitel mit Text versehen und erst am Ende eine Zusammenfassung erstellt. Derart typische Vorgehensweisen lassen sich von Diensten für das Change Management berücksichtigen, was der vorliegende Dienst jedoch nicht tut. Hierzu sind neben den für XML-Dokumenten typischen Dokumententypdefinitionen entsprechende Angaben über typische Veränderungsvorgänge notwendig.

Im Abschnitt 6.5.1 wird dargestellt, daß Elemente mit unterschiedlichen Elementnamen als zu unterschiedlich betrachtet werden, um einander zugeordnet werden zu können (bei der Erstellung des induzierten Graphen fehlen die entsprechenden Kanten). Durch die Angabe einer symmetrischen Relation läßt sich ausdrücken, wie ähnlich sich Elemente mit unterschiedlichen Elementnamen sein können. Am Beispiel der Auszeichnungssprache HTML sind sich die Elemente mit den Namen `<H1>`, `<H2>` etc. ähnlich genug, um einander zugeordnet werden zu können. Eine solche Relation läßt sich bei der Erstellung des induzierten Graphen berücksichtigen.

Das Operationen-Repertoire, welches der Erzeugung der *edit-scripts* zugrunde liegt, beschränkt sich hier auf die sechs Operationen *update, insert, delete, move, copy* und *glue*. Zusätzlich zu diesen sind insbesondere bei der Betrachtung von XML-Dokumenten weitere Operationen denkbar:

***split* / *merge*:** Für einige Anwendungen ist die Verwendung von Operationen sinnvoll, die das Aufteilen eines Elements oder Textkörpers in mehrere erlaubt. Umgekehrt läßt sich das Verschmelzen mehrerer Elemente und/oder Textkörper zu einem durch die Operation

merge modellieren. Diese beiden Operationen sind im Sinne der in Abschnitt 6.3 angestellten Überlegungen zur Umkehrbarkeit eines *edit-scripts* zueinander invers.

Wird bei der Verwendung der Auszeichnungssprache HTML ein Textkörper `<p>...x...y...z...</p>` durch eine Auszeichnung beispielsweise mit dem *Tag* `<b>` in die Struktur `<p>...x...<b>y</b>...z...</p>` modifiziert, ließe sich dieses durch eine *split*-Operation darstellen.

***replace*:** Der Vorgang des *Ersetzens* eines Elements (oder Textkörpers) durch ein anderes laßt sich durch die Verwendung einer *replace*-Operation darstellen, wenn die den beiden Dokumenten zugrundeliegende Dokumententypdefinition beispielsweise die alternative Verwendung der betreffenden Elemente vorsieht. Im Sinne der in Abschnitt 6.3 angestellten Überlegungen zur Umkehrbarkeit eines *edit-scripts* ist diese Operation zu sich selbst invers.

Bei den im Abschnitt 6.3 angestellten Überlegungen zur Umkehrbarkeit eines *edit-scripts* wird beschrieben, daß die *edit-scripts* nur bedingt umkehrbar sind. Durch die Angabe weiterer Informationen bei einigen Operationen (wie beispielsweise die einzelnen von einer *delete*-Operation entfernten Elemente) läßt sich ein *edit-scripts* während der Bearbeitung umkehren.

Das im Rahmen dieser Arbeit entwickelte Verfahren und der implementierte Dienst konzentrieren sich bei dem Vergleich semistrukturierter Daten stark auf deren Struktur. Zur Repräsentation der Daten werden beschriftete Bäume verwendet, deren Knoten durch die Inhalte von XML-Elementen (Elementnamen und Attribute) beziehungsweise von Textkörpern beschriftet werden. Der Inhalt letztgenannter Textkörper wird zwar für das Auffinden einer Zuordnung der Knoten genutzt, jedoch bei der Erzeugung der *edit-scripts* nicht differenziert berücksichtigt. Wird durch das Verfahren festgestellt, daß ein Textkörper A_t des Informationsbestandes A dem ihm zugeordneten Textkörper B_t des Informationsbestandes B nicht gleich ist, folgt daraus eine *update*-Operation, die den Inhalt des neuen Textkörpers setzt. Dabei wird die Ähnlichkeit der Textkörper nicht berücksichtigt, was durch die Anwendung eines der in Abschnitt 4.2 erwähnten Verfahren möglich ist.

Anhang A

Planarisierungsproblem in bipartiten Graphen

Dieser Abschnitt beschreibt einen Algorithmus, der im Rahmen dieser Arbeit entwickelt wurde und benutzt wird, um eine reihenfolgenorientierte Zuordnung der Kinder des Knoten $m \in T_A$ zu den Kindern des Knoten $n \in T_B$ zu erhalten. Dieses läßt sich auch beschreiben als das Problem, eine optimale Korrespondenz zu finden, die einen planaren Graphen darstellt.

Gegeben sei ein bipartiter Graph $G = (V_L, V_R, E, W)$ mit zwei disjunkten Knotentupeln $V_L = (v_{l1},\ v_{l2}\ ...v_{lm})$ und $V_R = (v_{r1},\ v_{r2}\ ...v_{rn})$, einer Kantenmenge E mit $e = (v_1, v_2) \in E \Rightarrow v_1 \in V_L\ \wedge\ v_2 \in V_R$ und einer Gewichtsfunktion $W : E \to \mathbb{R}_0^+$. Sei $E_U \subseteq \mathcal{P}(E)$ folgendermaßen definiert:

$$E_U = \{X \in \mathcal{P}(E) \mid \forall (v_p, v_q) \in X \neg\exists (v_{p'}, v_{q'}) \in X : (p' < p) \wedge (q' \geq q) \vee (p' > p) \wedge (q' \leq q)\}$$

E_U enthält also genau die Teilmengen der Kantenmengen, bei denen jedem Knoten aus V_R höchstens ein Knoten aus V_L zugeordnet ist und es zu keinen Kantenüberschneidungen kommt.
Gesucht wird nun eine Teilmenge aus E_U mit maximalem Gewicht:

$$E_{Umax} \in E_U : \forall E' \in E_U w(E_{Umax}) \geq W(E')$$

Zur Ermittlung von E_{Umax} berechnet man für jede Kante $e = (v_p, v_q) \in E$ die Nutzen e_n und die Kosten e_k, wobei sich die Nutzen aus dem Gewicht von e und die Kosten als Summe der Gewichte der von e geschnittenen Kanten sowie der Summe der anderen v_q berührenden Kanten errechnen:

$$e_n = W(e)$$

$$e_k = \sum_{(v_{p'}, v_{q'}) \in E, (p' < p) \wedge (q' \geq q)} w(v_{p'}, v_{q'}) + \sum_{(v_{p'}, v_{q'}) \in E, (p' > p) \wedge (q' \leq q)} w(v_{p'}, v_{q'})$$

Die Kanten werden dann primär ihrem Nettonutzen $e_{nk} = e_n - e_k$ und sekundär der Anzahl der geschnittenen anderen Kanten entsprechend sortiert. Dieser Reihenfolge entsprechend prüft man für jede einzelne Kante, ob sie bereits blockiert ist oder nicht. Wenn sie noch frei ist, wird sie in eine Liste der letztlich bestehenden Kanten übernommen und alle von ihr blockierten Kanten werden als solches markiert.

Da die zu bevorzugenden Kanten in der eigentlichen Aufgabenstellung durch ein niedriges Gewicht ausgezeichnet sind, ist die Gewichtung der Kanten zu invertieren. Dieses läßt sich recht einfach erreichen, indem man die Summe $W_{Total} = \sum_{e \in E} W(e)$ aller Kanten ermittelt und jeder Kante e

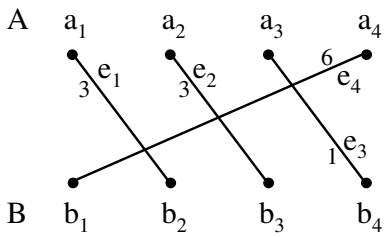

Abbildung A.1: Gegenbeispiel für das entwickelte Verfahren zur Ermittlung einer gewichtsmaximalen Zuordnung ohne Kantenüberschneidung in einem bipartiten Graph

das Gewicht $W(e) := W_{Total} - W(e)$ gibt.

Dieses Verfahren ermittelt erfahrungsgemäß recht gute aber nicht immer optimale Ergebnisse, wie das in Abbildung A.1 dargestellte Gegenbeispiel zeigt: mit der dort angegebenen Kantengewichtung würde die Kante e_4 ausgewählt werden, da für sie $e_{nk} = 6 - (3 + 3 + 1) = -1$ gilt. Die Nettonutzen der übrigen Kanten betragen -3 für die Kanten e_1 und e_2 und -5 für die Kante e_3. Dennoch wäre eine Lösung mit den drei Kanten e_1, e_2 und e_3 mit einem Gesamtgewicht von 7 besser, als die Lösung mit der Kante e_4 mit einem Gewicht von 6.

Bessere Lösungen dieses Problems finden sich in [MW98].

Anhang B

DTD des *edit-scripts* und *edit-script*-Beispiele in XML

In Abschnitt 6.4.1 wurde die Repräsentation der Änderungen durch XML beschrieben. Für diese Repräsentation wurde eine DTD (*Document-Type-Definition*) entworfen. Sie wird im Abschnitt B.1 dieses Anhangs wiedergegeben. Im Abschnitt B.2 wird für jede Operation des Operationen-Repertoires exemplarisch ein entsprechender Ausschnitt eines *edit-script*-XML-Dokuments angegeben.

B.1 DTD des *edit-scripts*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT es (date?, time?, fnames?, comment?
              (upd|ins|del|mov|cpy|glu)*)>

<!ELEMENT date EMPTY> <!ATTLIST date
    year  CDATA #REQUIRED
    month CDATA #REQUIRED
    day   CDATA #REQUIRED>

<!ELEMENT time EMPTY> <!ATTLIST time
    hour   CDATA #REQUIRED
    minute CDATA #REQUIRED
    second CDATA #REQUIRED>

<!ELEMENT fnames EMPTY> <!ATTLIST fnames
    fname1 CDATA #REQUIRED
    fname2 CDATA #REQUIRED>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT upd (charcontent|attribute*)>
<!ATTLIST upd
    nodenr CDATA #REQUIRED>

<!ELEMENT charcontent (#PCDATA)>
```

```
<!ELEMENT ins (node)>
<!ATTLIST ins
    targetparent CDATA #REQUIRED
    childpos     CDATA #IMPLIED>

<!ELEMENT del EMPTY>
<!ATTLIST del
    nodenr CDATA #REQUIRED>

<!ELEMENT mov EMPTY>
<!ATTLIST mov
    nodenr       CDATA #REQUIRED
    targetparent CDATA #REQUIRED
    childpos     CDATA #IMPLIED>

<!ELEMENT cpy EMPTY>
<!ATTLIST cpy
    nodenr       CDATA #REQUIRED
    targetparent CDATA #REQUIRED
    childpos     CDATA #IMPLIED>

<!ELEMENT glu EMPTY>
<!ATTLIST glu
    nodenr     CDATA #REQUIRED
    originalnr CDATA #REQUIRED>

<!ELEMENT node (elnode|data)>

<!ELEMENT elnode (attribute*, (node*))>
<!ATTLIST elnode
    name CDATA #REQUIRED>

<!ELEMENT data (#PCDATA)>

<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
    attname  CDATA #REQUIRED
    attvalue CDATA #REQUIRED
    atttype  CDATA #REQUIRED>
```

B.2 Beispiele für *edit-scripts* in XML

Für jede Operation des *edit-scripts* wird ein kurzes Beispiel-XML-Dokument angegeben, das der oben dargestellten DTD genügt:

B.2.1 *update*-Operation

```
<?xml version="1.0"?>
<!DOCTYPE es SYSTEM "es.dtd">

<es>
  <upd nodenr="1">
```

```
    <attribute attname="n" attvalue="neu" atttype="CDATA" />
  </upd>
</es>
```

B.2.2 *insert*-Operation

```
<?xml version="1.0"?>
<!DOCTYPE es SYSTEM "es.dtd">

<es>
  <ins targetparent="1" childpos="1">
    <node>
      <elnode name="D">
        <attribute attname="n" attvalue="m" atttype="CDATA" />
        <node>
          <elnode name="D">
            <attribute attname="n" attvalue="n" atttype="CDATA" />
          </elnode>
        </node>
        <node>
          <elnode name="D">
            <attribute attname="n" attvalue="o" atttype="CDATA" />
          </elnode>
        </node>
      </elnode>
    </node>
  </ins>
</es>
```

B.2.3 *delete*-Operation

```
<?xml version="1.0"?>
<!DOCTYPE es SYSTEM "es.dtd">

<es>
  <del nodenr="1"/>
</es>
```

B.2.4 *move*-Operation

```
<?xml version="1.0"?>
<!DOCTYPE es SYSTEM "es.dtd">

<es>
  <mov nodenr="1" targetparent="4" childpos="1"/>
</es>
```

B.2.5 *copy*-Operation

```
<?xml version="1.0"?>
<!DOCTYPE es SYSTEM "es.dtd">

<es>
  <cpy nodenr="1" targetparent="4" childpos="1"/>
```

```
</es>
```

B.2.6 *glue*-Operation

```
<?xml version="1.0"?>
<!DOCTYPE es SYSTEM "es.dtd">

<es>
  <glu nodenr="6" originalnr="1"/>
</es>
```

Anhang C

Die CD-Rom zur Arbeit

Diesem schriftlichen Teil der Arbeit ist eine CD-Rom beigelegt, auf der sich unter anderem der im Rahmen dieser Arbeit implementierte Dienst befindet. Nachfolgend wird der Inhalt der sich auf dieser CD-Rom befindenden Datei `readme.txt` wiedergegeben, in der der Inhalt des Datenträgers sowie die Installation und Ausführung des Dienstes beschrieben werden.

```
------------------------------------------------------------------------

README.TXT                                              Hamburg, 24.02.2000
------------------------------------------------------------------------

Der Inhalt dieser CD-Rom gehört konzeptuell zu einer Ende Februar
fertiggestellten Diplomarbeit:

  ------------------------------------------------------------
  "Change Management für Metadaten in Digitalen Bibliotheken"

  <abstract>
  Die von Digitalen Bibliotheken zur Verfügung gestellten
  Informationsressourcen unterliegen Änderungen, von
  denen auch deren Metadaten betroffen sind. Die
  Handhabung dieser Veränderungen erfordert den Einsatz
  eines Systems zum Change Management, welches das
  Stattfinden von Änderungen erkennt, über relevante
  Änderungen informiert und entsprechende Reaktionen
  erlaubt.

  Ausgangspunkt dieser Arbeit ist das Change Management
  für semistrukturierte Daten, wie man sie für die
  Darstellung von Metadaten verwenden kann. Sie lassen
  sich durch beschriftete Bäume repräsentieren.

  In der Arbeit wird ein Verfahren entwickelt, welches
  zwei beschriftete Bäume miteinander vergleicht und
  deren Differenzen in Form einer Folge von
  Änderungsoperationen ermittelt. Derartigen edit-scripts
  liegt ein Operationen-Repertoire zugrunde, welches das
  Ändern, Einfügen, Entfernen, Verschieben, Kopieren und
  Einschmelzen von Teilbäumen umfaßt, wodurch die
  Änderungsfolgen eine hohe Aussagekraft erhalten.
```

```
  Das entwickelte Verfahren wird von einem
  konfigurierbaren Dienst implementiert, dessen
  Informationsaustausch in Bezug auf die zu
  vergleichenden Metadaten und die erzeugten edit-scripts
  auf der Grundlage von XML (extensible markup language)
  erfolgt, wodurch weite Anwendungsbereiche für den
  Dienst und eine Integration des Dienstes in andere
  Anwendungen ermöglicht werden.

  ------------------------------------------------------------

1. Inhalt der CD-Rom

2. Voraussetzungen

3. Installation und Start der Anwendung (Microsoft Windows NT/9x)

  ------------------------------------------------------------

1. Inhalt der CD-Rom
--------------------

Im Rahmen der Arbeit wurde ein Dienst entwickelt, der es ermöglicht,
XML-Dokumente miteinander zu vergleichen. Diese CD enthält die zur
Ausführung des Dienstes notwendigen Dateien.

Auf dem Datenträger befinden sich vier Verzeichnisse:

- dipl
  Der schriftliche Teil der Arbeit befindet sich in Form einer
  Postscript-Datei in diesem Verzeichnis. Zusätzlich wird sie
  über die Adresse http://www.sts.tu-harburg.de/papers/entry.html
  erreichbar sein.

- xtree
  Dieses Verzeichnis enthält den implementierten Dienst, die hierfür
  notwendigen implementierten packages, sowie die bereits
  extrahierten Bibliotheken zum Parsen von XML-Dokumenten.


  ------------------------------------------------------------

2. Voraussetzungen
--------------------

Zur Ausführung des Dienstes ist die Java Laufzeitumgebung der Version 2
oder höher sowie der "XML Parser for JAVA" von IBM, Version 2.0.11, 1999
(http://www.alphaWorks.ibm.com/tech/xml4j) und die SAX-Bibliothek
(http://www.megginson.com/SAX/) notwendig.

  ------------------------------------------------------------

3. Installation und Start der Anwendung (Microsoft Windows NT/9x)
```

```
-----------------------------------------------------------------

Das Verzeichnis 'xtree' muß von der CD auf die Festplatte kopiert werden.
In dem Verzeichnis befindet sich eine MS-DOS Batch-Datei 'XTree.bat', die
den Aufruf der Anwendung vornimmt. In dieser Datei müssen zwei Pfade gesetzt
werden:

----
...
REM ---------------- Set the JDK1.2 directory
set JDK=c:\jdk1.2.2

REM ---------------- Set the XTree directory
set XTREEDIR=c:\ch.spreen\diplom-rom\xtree\xtree
...
----

Der erste Pfad bezeichnet das Verzeichnis, in dem sich JAVA befinden sollte.
Der zweite Pfad bezeichnet das Verzeichnis, in dem sich (u.a.) die Datei
'XTree.class' befindet. Diese Pfade sind entsprechend den Umgebungen
anzupassen.

Die Bibliotheken des XML-Parsers und der SAX-Schnittstelle müssen von dem
in der Zeile
----
REM ---------------- Set the classpath for java
set CLASSPATH=%JDK%\lib;%XTREEDIR%\xml4j_2_0_11.jar;%XTREEDIR%\sax.jar;%XTREEDIR%\..
----
gesetzten "classpath" erfaßt werden.

Durch Ausführung der Datei 'XTree.bat' (von einem beliebigen Verzeichnis aus)
wird die Anwendung schließlich gestartet.

-----------------------------------------------------------------------

Christoph Spreen                      ch.spreen@sun01.sts.tu-harburg.de

AB Softwaresysteme (STS) TU Hamburg-Harburg   Tel: (+49)40-42878-2116
Harburger Schlossstr. 20, D-21079 Hamburg

              "http://www.sts.tu-harburg.de/~ch.spreen"
-----------------------------------------------------------------------
```

Anhang D

Glossar

#childpos: #childpos ist ein Parameter für Operationen, die einen Knoten oder Teilbaum in einen bestehenden Baum einsetzen. *#childpos* bezeichnet die Position innerhalb der Kinder des Zielvaters, die der einzusetzende Knoten oder die Wurzel des einzusetzenden Baumes einnehmen soll.

#node: Allgemein bedeutet ein vor die Bezeichnung eines Knoten gestelltes Zeichen „#“, daß es sich hierbei um die Nummer des betreffenden Knoten handeln soll.

Änderungsoperation: Eine einzelne Änderungsoperation ist Element eines Operationen-Repertoires: $op \in OPR = \{opr_1, ..., opr_n\}$

Arbeitsbestand: Für die sequentielle Abarbeitung eines *edit-scripts* wird eine Kopie des Basisbestands erstellt, die schrittweise modifiziert wird.

Basisbestand, Zielbestand: Die beiden zu vergleichenden Informationsbestände werden *Basis-* und *Zielbestand* genannt, um sie benennen und unterscheiden zu können. Analog dazu ergeben sich auch die Begriffe *Basis-* und *Ziel**baum*** und *Basis-* und *Ziel**sequenz*** etc.

Baum: Ein Graph $T = (V, E)$ heißt *Baum*, wenn er zyklenfrei und zusammenhängend ist. Meist wird ein bestimmter Knoten $w \in V$ als *Wurzel* bezeichnet. Da T zyklenfrei und zusammenhängend ist, gibt es für zwei Knoten $v_1, v_2 \in V$ genau einen Weg, der diese Knoten verbindet. Für jeden Knoten $v \in V, v \neq w$ gibt es einen v auf dem Weg zur Wurzel nächstgelegenen Knoten $p(v)$, der als *Vater* von v bezeichnet wird. Alle anderen benachbarten Knoten von v bilden die Menge seiner *Kinder* $c(v)$. Für einen Knoten $v \in V$ bezeichnet $p^i(v)$ denjenigen Knoten, der auf dem Weg von v zur Wurzel w liegt, und durch einen Weg der Länge i von v getrennt ist. Demnach bezeichnet beispielsweise $p^0(v)$ den Knoten v selbst und $p^2(v)$ den Vater des Vaters von v. Ein Knoten $v \in V$ wird als *Blatt* bezeichnet, wenn v keine Kinder hat. Alle Knoten, die weder Blatt noch Wurzel sind, werden *innere Knoten* genannt. Häufig betrachtet man *beschriftete* Bäume: sei L eine Menge von Bezeichnern, dann gibt man zusätzlich zu den Mengen der Knoten und Kanten eine Funktion $l : V \rightarrow L$ als *Beschriftung* an, die jedem Knoten einen Bezeichner der Menge L zuordnet[Jun90, Aig84, Hal89].

Beschneidungsregeln: Das in [CGM97] beschriebene Verfahren ermittelt eine Zuordnung der Knoten der miteinander zu vergleichenden Bäume auf Grundlage von Kosten, die eine Verbindung zweier Knoten nach sich zieht. Das dort beschriebene Verfahren sieht vor, zunächst jeden Knoten des einen Baumes mit jedem Knoten des anderen Baumes zu verbinden und anschließend iterativ mit Hilfe einiger Regeln (den *Beschneidungsregeln* oder engl. *pruning-rules*) überflüssige Verbindungen wieder zu lösen.

Bipartiter Graph: Ein bipartiter Graph $G_{bip} = (V_1, V_2, E)$ ist ein Graph mit zwei Knotenmengen V_1 und V_2 und einer Kantenmenge, die nur Kanten enthält, die Knoten aus V_1 mit Knoten aus V_2 verbinden [Jun90].

Edit-Graph: Die Umformung einer Sequenz A in eine Sequenz B läßt sich mit Hilfe eines sogenannten *edit-Graphen* veranschaulichen, wie er in Abbildung 4.4 dargestellt ist. Ein *edit-Graph* ist ein gerichteter Graph mit einem Knoten für jedes Paar $(x, y) \in [0..|A|] \times [0..|B|]$, wobei $|A|$ und $|B|$ die Länge der jeweiligen Sequenz angibt. Von jedem Knoten führt je eine Kante zu seinem rechten Nachbarn (soweit ein solcher existiert) und eine Kante zu seinem unteren Nachbarn (soweit ein solcher existiert). Eine Kante, die von dem Knoten $(2, 2)$ zu dem Knoten $(2, 3)$ verläuft, repräsentiert das Einfügen des Elementes b_3 in der Sequenz B, analog dazu entspricht eine Kante von $(4, 5)$ nach $(5, 5)$ dem Entfernen des Elementes a_5 der Sequenz A. Zusätzlich erhält der Graph diagonale Kanten für je zwei äquivalente Elemente beider Sequenzen: eine diagonale Kante $(x, y) \rightarrow (x + 1, y + 1)$ existiert genau dann, wenn $a_{x+1} \equiv b_{y+1}$ gilt.

***edit-script*:** Eine Folge von Änderungsoperationen $es = (op_1, ..., op_n)$ mit $op_i \in OPR$. Die Länge eines *edit-scripts* es ist $|es|$. Eine Menge mehrerer *edit-scripts* ist $ES = \{es_1, ..., es_n\}$.

Elementknoten: Bei der Interpretation von XML-Dokumenten als Bäume werden zwei Typen von Knoten unterschieden: *Elementknoten* und *Textkörperknoten*. Elementknoten haben einen eindeutigen *Elementnamen*, eine Menge von *Attributen* und möglicherweise untergeordnete Knoten (Kinder). Sie repräsentieren XML-Elemente eines XML-Dokumentes.

Gemeinsame Untersequenz: Sind $A = (a_1, ..., a_m), B = (b_1, ..., b_n)$ und $C = (c_1, ..., c_o)$ Sequenzen, dann ist die Sequenz C eine *Untersequenz* der Sequenz A, wenn sie durch Entfernen einzelner Elemente aus A hergestellt werden kann. C ist *gemeinsame Untersequenz* der Sequenzen A und B, wenn sie eine Untersequenz sowohl von A, als auch von B ist. Die *längste gemeinsame Untersequenz - LCS* ergibt sich als diejenige gemeinsame Untersequenz, die gegenüber allen existierenden gemeinsamen Untersequenzen die größte Länge aufweist [WMMM90].

Gewichtetes Kantenüberdeckungsproblem: Gegeben sei ein Graph $G = (V, E, W)$, wobei $W : E \rightarrow \mathbb{R}$ die Kanten gewichtet. Das *gewichtete Kantenüberdeckungsproblem* besteht darin, aus der Menge aller Kantenüberdeckungen $\mathcal{K}$ diejenige Kantenüberdeckung $\mathcal{K}_{\Updownarrow\backslash\backslash}$ zu finden, deren Summe der Kantengewichte minimal ist.

Gewichtetes Korrespondenzproblem: Gegeben sei ein Graph $G = (V, E, W)$, wobei $W : E \rightarrow \mathbb{R}$ die Kanten gewichtet. Das gewichtete Korrespondenzproblem besteht darin, eine Menge $E' \subseteq E$ zu finden, so daß $G' = (V, E')$ eine Korrespondenz darstellt und die Summe der Kantengewichte von E' möglichst groß ist.

Gewichtsminimale Kantenüberdeckung: Gegeben sei ein Graph $G = (V, E, W)$, wobei $W : E \rightarrow \mathbb{R}$ die Kanten gewichtet. Eine *gewichtsminimale Kantenüberdeckung* ist eine Kantenüberdeckung $\mathcal{K}* \subseteq E$, deren Summe der Kantengewichte verglichen mit der aller anderen Kantenüberdeckungen minimal ist: $\mathcal{K}*$ ist Kantenüberdeckung von G und $W(\mathcal{K}* = min\{W(\mathcal{K}) : \mathcal{K}$ ist Kantenüberdeckung von $G\}$.

Graph: Ein Paar $G = (V, E)$, bestehend aus einer nicht leeren Menge von *Knoten* V (*vertices*) und einer Menge von *Kanten* $E \subseteq V \times V$ (*edges*) wird als *Graph* bezeichnet. Gibt es zu einem Knoten $v_1 \in V$ einen Knoten $v_2 \in V$, so daß $e = (v_1; v_2) \in E$, dann sagt man, daß v_1 und v_2 mit e *inzident* sind und daß v_1 und v_2 *adjazent* sind (insbesondere kann es sich bei den Knoten v_1 und v_2 um den selben Knoten handeln, der durch die Kante e mit sich selber verbunden ist.) [Jun90]. Ist ein Knoten v_1 mit einer Kante e inzident, so wird in dieser Arbeit auch geschrieben, daß v_1 von e *getroffen* wird.

Informationsart: Gewissermaßen die Typen von Informationsbeständen. Informationsarten können sich in ihrer Struktur (Syntax), aber auch nur in ihrer Bedeutung (Semantik) unterscheiden.

Informationsbestand: Ausprägung einer Informationsart. Sie repräsentiert demnach beispielsweise eine Sequenz, einen Baum oder ein anderes Objekt.

Inhaltsgewicht: Das *Inhaltsgewicht* einer Kante $e = (m; n)$ ist eines der vier Gewichtsteile (Inhaltsgewicht, Nachfolgergewicht, Ordnungsgewicht und Umgebungsgewicht), aus denen sich das Gesamtgewicht einer Kante zur Ermittlung einer gewichtsminimalen Kantenüberdeckung zusammensetzt. Das Inhaltsgewicht bezieht sich nur auf die *Inhalte* der betroffenen Knoten m und n (siehe Abschnitt 6.5.1).

Kantenüberdeckung: Gegeben sei ein ungerichteter Graph $G = (V, E)$. Eine Kantenmenge $\mathcal{K} \subseteq E$ nennt man Kantenüberdeckung von G, genau dann, wenn gilt: $\forall v_i \in V\ \exists v_j \in V$: $(v_i, v_j) \in \mathcal{K}$. Eine Kantenüberdeckung ist *minimal*, wenn sie durch das Entfernen einer beliebigen Kante ihre Eigenschaft als Kantenüberdeckung verlieren würde.

Kind-Position: die Kindposition ist eine Zahl, die angibt, an welcher Position innerhalb aller Kinder eines Knoten ein neu einzufügender Teilbaum eingesetzt werden soll.

Korrespondenz: Eine Korrespondenz in einem Graphen G ist eine Menge K von Kanten, von denen keine zwei einen Endpunkt gemeinsam haben [Jun90] (jeder Knoten wird von höchstens einer Kante getroffen).

Korrespondenzproblem: Gegeben sei ein Graph $G = (V, E)$, das Korrespondenzproblem besteht darin, eine möglichst große Teilmenge $E' \subseteq E$ zu finden, so daß $G' = (V, E')$ eine Korrespondenz darstellt. Zusätzlich werden hier gewichtete Kanten vorausgesetzt ($G = (V, E, W)$) , so daß das Problem nicht mehr darin besteht, eine möglichst große Menge $E' \subseteq E$ zu erhalten, sondern eine Menge E', deren Summe der Kantengewichte möglichst groß ist. Das Korrespondenzproblem wird auch oft als *matching-* oder Heiratsproblem bezeichnet [Hal89].

***La-Diff*:** Das in [CRGMW96] beschriebene Verfahren zum Vergleich geordneter beschrifteter Bäume wird mit *La-Diff* bezeichnet.

Längste gemeinsame Untersequenz: Sind $A = (a_1, ..., a_m)$, $B = (b_1, \ ..., \ b_n)$ und $C = (c_1, \ ..., \ c_o)$ Sequenzen, dann ist die Sequenz C eine *Untersequenz* der Sequenz A, wenn sie durch Entfernen einzelner Elemente aus A hergestellt werden kann. C ist *gemeinsame Untersequenz* der Sequenzen A und B, wenn sie eine Untersequenz sowohl von A, als auch von B ist. Die *längste gemeinsame Untersequenz - LCS* ergibt sich als diejenige gemeinsame Untersequenz, die gegenüber allen existierenden gemeinsamen Untersequenzen die größte Länge aufweist [WMMM90].

***Levenshtein-Abstand*:** Zur Bewertung von *edit-scripts* bei Sequenzen ist die als *Levenshtein-Abstand* bezeichnete Länge eines *edit-scripts* von besonderem Interesse [WMMM90].

***MH-Diff*:** Das in [CGM97] beschriebene Verfahren zum Vergleich ungeordneter beschrifteter Bäume wird mit *MH-Diff* bezeichnet.

***Minus*-Knoten:** Ist einem Knoten aus dem Baum $m \in T_A$ in der gewichtsminimalen Kantenüberdeckung $\mathcal{K}$ kein Knoten des Baumes T_B zugeordnet, so ist er mit dem speziellen, konzeptionell zum Baum T_B gehörenden *Minus*-Knoten verbunden, um deutlich zu machen, daß er im Verlauf der Abarbeitung des *edit-scripts* entfernt wird.

***Minus*-Kante:** Diejenigen Kanten, die einen Knoten $m \in T_A$ mit dem *Minus*-Knoten verbinden, werden *Minus*-Kanten genannt.

Nachfolgergewicht: Das *Nachfolgergewicht* einer Kante $e = (m; n)$ ist eines der vier Gewichtsteile (Inhaltsgewicht, Nachfolgergewicht, Ordnungsgewicht und Umgebungsgewicht), aus denen sich das Gesamtgewicht einer Kante zur Ermittlung einer gewichtsminimalen Kantenüberdeckung zusammensetzt. Das Nachfolgergewicht bezieht sich nur auf die Kinder der betroffenen Knoten m und n (siehe Abschnitt 6.5.1).

Operationen-Repertoire: Das Operationen-Repertoire, das in dem hier entwickelten Verfahren verwendet wird umfaßt die folgenden Operationen (das in [CGM97] verwendete Operationen-Repertoire umfaßt die gleichen Operationen, jedoch mit einer etwas anderen Syntax und Semantik)

- op_{upd}: *update*-Operation
- op_{ins}: *insert*-Operation
- op_{del}: *delete*-Operation
- op_{mov}: *move*-Operation
- op_{cpy}: *copy*-Operation
- op_{glu}: *glue*-Operation

Ordnungsgewicht: Das *Ordnungsgewicht* einer Kante $e = (m; n)$ ist eines der vier Gewichtsteile (Inhaltsgewicht, Nachfolgergewicht, Ordnungsgewicht und Umgebungsgewicht), aus denen sich das Gesamtgewicht einer Kante zur Ermittlung einer gewichtsminimalen Kantenüberdeckung zusammensetzt. Das Ordnungsgewicht bezieht sich nur auf die Position der betreffenden Knoten m und n bezüglich ihrer Geschwister (siehe Abschnitt 6.5.1).

Planarer Graph: Ein Planarer Graph $G = (V, E)$ ist ein Graph, der sich ohne überflüssige Schnittpunkte in der Ebene zeichnen läßt [Jun90].

***Plus*-Knoten:** Ist einem Knoten aus dem Baum $n \in T_B$ in der gewichtsminimalen Kantenüberdeckung $\mathcal{K}$ kein Knoten des Baumes T_A zugeordnet, so ist er mit dem speziellen, konzeptionell zum Baum T_A gehörenden *Plus*-Knoten verbunden, um deutlich zu machen, daß er im Verlauf der Abarbeitung des *edit-scripts* eingefügt wird.

***Plus*-Kante:** Diejenigen Kanten, die einen Knoten $n \in T_B$ mit dem *Plus*-Knoten verbinden, werden *Plus*-Kanten genannt.

Primärpartner: der Primärpartner eines Knoten $n \in T_B$ ist derjenige Knoten, mit dem n verbunden ist. Wird n von mehreren Kanten getroffen, so gibt es keinen Primärpartner von n.

SAX: *SAX (Simple API for XML)* [Meg99, Edd99b] ist eine standardisierte Schnittstelle für das ereignisgesteuerte *Parsen* von XML-Dokumenten. Der *Parser XML Parser for JAVA*[IBM98a] verwendet diese Schnittstelle.

targetparent: Parameter für Operationen, mit dem der Zielvater gemeint ist. Bezieht man sich auf die Knotennummer von *targetparent*, schreibt man *#targetparent*.

Textkörperknoten: Bei der Interpretation von XML-Dokumenten als Bäume werden zwei Typen von Knoten unterschieden: *Elementknoten* und *Textkörperknoten*. Der Inhalt eines Textkörperknoten besteht nur aus einer Zeichenkette. Diese Knoten haben insbesondere keinen Namen, keine Attribute und keine untergeordnete Knoten. Ferner besitzen Textkörperknoten keine Textkörperknoten als unmittelbaren Nachbarn. Ein Textkörperknoten repräsentiert einen Textkörper eines XML-Dokuments.

Umgebungsgewicht: Das *Umgebungsgewicht* einer Kante $e = (m; n)$ ist eines der vier Gewichtsteile (Inhaltsgewicht, Nachfolgergewicht, Ordnungsgewicht und Umgebungsgewicht), aus denen sich das Gesamtgewicht einer Kante zur Ermittlung einer gewichtsminimalen Kantenüberdeckung zusammensetzt. Das Umgebungsgewicht bezieht sich auf die gesamte *Umgebungen* der betroffenen Knoten m und n (siehe Abschnitt 6.5.1).

UML (*unified modelling language*): Die *unified modelling language* UML ist eine Modellierungssprache und damit eine (überwiedgend graphische) Notation, die dazu verwendet wird, Entwürfe auszudrücken. In vielerlei Hinsicht ist eine Modellierungssprache der wichtigste Teil einer Methode. Sie ist zumindest der essentielle Teil zum Kommunizieren [FS97].

***Ungarischer Algorithmus*:** Der *Ungarische Algorithmus* ist ein Algorithmus, mit dem sich eine optimale Korrespondenz in einem gewichteten bipartiten Graphen mit der Komplexität $O(n^3)$ berechnen läßt.

Untersequenz: Sind $A = (a_1, ..., a_m), B = (b_1, ..., b_n)$ und $C = (c_1, ..., c_o)$ Sequenzen, dann ist die Sequenz C eine *Untersequenz* der Sequenz A, wenn sie durch Entfernen einzelner Elemente aus A hergestellt werden kann. C ist *gemeinsame Untersequenz* der Sequenzen A und B, wenn sie eine Untersequenz sowohl von A, als auch von B ist. Die *längste gemeinsame Untersequenz - LCS* ergibt sich als diejenige gemeinsame Untersequenz, die gegenüber allen existierenden gemeinsamen Untersequenzen die größte Länge aufweist [WMMM90].

Vollständiger bipartiter Graph: Ein bipartiter Graph $G = (V_1, V_2, E)$ ist vollständig, wenn Jeder Knoten der Menge V_1 mit jedem Knoten der Menge V_2 verbunden ist: $\forall(v_1; v2) \in V_1 \times V_2 \exists e \in E$.

XML (*extensible markup language*): Das WORLD WIDE WEB CONSORTIUM (W3C) erarbeitete mit XML (*Extensible Markup Language*) einen Standard, mit dem sich eine Web-Auszeichnungssprache anwendungsspezifisch definieren lässt. XML ist eine standardisierte Sprache zur Notation der Syntax von Auszeichnungssprachen. Somit ist XML eine *Metagrammatik* für kontextfreie Grammatiken - also die Grammatik einer Sprache mit der sich die Regeln von Grammatiken notieren lassen [Tol99] (siehe Abbildung 5.1).

Zielvater: der Zielvater ist der Knoten im Baum T_A der der neue Vater eines einzusetzenden Teilbaumes werden soll.

Literaturverzeichnis

[Aig84] Martin Aigner. *Graphentheorie*. B. G. Teubner, Stuttgart, 1984.

[Ata99] Mikhail J. Atallah, editor. *Algorithms and theory of computation handbook*. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1999.

[Bar98] Andreas Barth. *Digital libraries in computer science: the MeDoc approach*, volume 1392 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1998.

[Ber98] R. Berndt, editor. *Unternehmen im Wandel - Change Management*. Springer-Verlag, 1998.

[Bra99] Neil Bradley. *The XML companion*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[CGM97] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 26–37, New York, May 13–15 1997. ACM Press.

[Cha97] Chawathe, Sudarshan S. The C3 Project at Stanford, 1997. `http://www-db.stanford.edu/c3/c3.html`.

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2), 1996.

[DBCK96] Fred Douglis, Thomas Ball, Yih-Farn Chen, and Eleftherios Koutsofios. WebGUIDE: querying and navigating changes in Web repositories. *Computer Networks and ISDN Systems*, 28(7–11):1335–1344, May 1996.

[Dev] Tympani Development. Tympani - information management & offline web search agents. `www.tympani.com`.

[Eck98] Bruce Eckel. *Thinking in Java*. SunSoft Press Java series. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1998.

[Edd99a] Edd Dumbill et al. XML.com, 1999. `http://www.xml.com/pub/Guide/XML-Parsers`.

[Edd99b] Edd Dumbill et al. Xml.com - the sax-based approach, 1999. `http://www.xml.com/pub/1999/11/cplus/sax.html`.

[ELW98] Robert Eckstein, Marc Loy, and Dave Wood. *Java Swing*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1998.

[FS97] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Modeling Object Language*. Object Technology Series. Addison-Wesley, 1997.

[GHJV96] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996.

[GNU84] GNU Project. GNU's Not Unix!, 1984. `http://www.gnu.org/home.html`.

[Hal89] Rudolf Halin. *Graphentheorie*. Wissenschaftliche Buchgesellschaft, Darmstadt, 1989.

[Hil98] Ed Hill. Webpluck - gathering data from the web, 1998. `http://strobe.weeg.uiowa.edu/~edhill/public/webpluck/article.html`.

[HSW96] Darren R. Hardy, Michael F. Schwartz, and Duane Wessels. Harvest user's manual, January 1996. `http://harvest.transarc.com`.

[HU79] J. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Language, and Computation*. Addison–Wesley, Reading, MA, 1979.

[IBM98a] IBM-Alphaworks. XML Parser for Java, 1998. `http://alphaworks.ibm.com/formula/xml`.

[IBM98b] IBM-Alphaworks. XML TreeDiff, 1998. `http://alphaworks.ibm.com/formula/xml`.

[IBM99] IBM-Alphaworks. XML Diff and Merge Tool, 1999. `http://alphaworks.ibm.com/formula/xml`.

[IM98] P. Ion and R. Miner. Mathematical Markup Language (MathML) 1.0 Specification. W3C Recommendation., Apr 7, 1998. `http://www.w3.org/TR/REC-MathML`.

[Jun90] D. Jungnickel. *Graphen, Netzwerke und Algorithmen*. BI Wissenschaftsverlag, 1990. 2. überarbeitete und erweitete Auflage.

[Kos94a] Martijn Koster. Guidelines for robot writers, 1994. `http://web.nexor.co.uk/users/mak/doc/robots/guidelines.html`.

[Kos94b] Martijn Koster. A standard for robot exclusion, 1994. `http://web.nexor.co.uk/users/mak/doc/robots/norobots.html`.

[Kos95] Martijn Koster. Robots in the web: threat or treat?, 1995. `http://web.nexor.co.uk/users/mak/doc/robots/threat-or-treat.html`.

[Meg99] Megginson Technologies. *SAX 1.0: The Simple API for XML*, 1999. `http://www.megginson.com/SAX/`.

[MS0] How to detect and compare file differences. `http://support.microsoft.com/support/kb/articles/Q119/1/10.asp`.

[MS98] Stefan Middendorf and Reiner Singer. *Java Programmierhandbuch und Referenz Erweiterungen*. dpunkt Verlag, Ringstraße 19, 69115 Heidelberg, Germany, April 1998.

[MW98] Mutzel and Weiskircher. Two-layer planarization in graph drawing. In *ISAAC: 9th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms), Organized by Special Interest Group on Algorithms (SIGAL) of the Information Processing Society of Japan (IPSJ) and the Technical Group on Theoretical Foundation of Computing of the Institute of Electronics, Information and Communication Engineers (IEICE))*, 1998.

[Mye86] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–256, 1986.

[NCK+92] C. M. Neuwirth, R. Chandhok, D. S. Kaufer, P. Erion, J. Morris, and D. Miller. Flexible diff-ing in a collaborative writing system. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, Collaborative writing, pages 147–154, Toronto, Ontario, 1992. ACM Press.

[NHM+96] C. Niederée, C. Hattendorf, S. Müßig, M. Warnke, and J.W. Schmidt. Warburg electronic library: Eine digitale Bibliothek für die Politische Ikonographie. *unihh forschung*, XXXI, 1996.

[Oes97] Bernd Oestereich. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*. Oldenbourg, Muenchen, 3 edition, 1997.

[PB93] G. Pomberger and G. Blaschek. *Software Engineering. Prototyping und objektorientierte Software-Entwicklung*. Carl Hanser Verlag, Muenchen, 1993.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Prentice Hall, 1982.

[RBP+94] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Objektorientiertes Modellieren und Entwerfen*. Carl Hanser Verlag, Muenchen, 1994.

[Sch92] Schöning,Uwe. *Theoretische Informatik kurz gefasst*. BI Wissenschaftsverlag, Universität Ulm, 3 edition, 1992.

[Sch97] Schmidt, J.W. and Warnke, M. Die *Warburg Electronic Library*, 1997. `http://www.sts.tu-harburg.de/projects/WEL`.

[SSNM97] J.W. Schmidt, G. Schröder, C. Niederée, and F. Matthes. Linguistic and architectural requirements for personalized digital libraries. *International Journal of Digital Libraries*, 1(1), 1997.

[Sun95] Sun Microsystems. *The Java Language Specification*, release 1.0 alpha 3 edition, 1995. `http://java.sun.com/1.0alpha3/doc/javaspec/javaspec.ps`.

[Sun00] Sun Microsystems. *Java Standard Edition Platform Documentation*, 2000. `http://java.sun.com/docs/javaspec`.

[SW99] M. Seeboerger-Weichselbaum. *XML*. bhv-Verlag GmbH, 1999.

[SWZS94] Shasha, Wang, Zhang, and Shih. Exact and approximate algorithms for unordered tree matching. *IEEETSMC: IEEE Transactions on Systems, Man, and Cybernetics*, 24, 1994.

[SZ90] Dennis Shasha and Kaizhong Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, 11(4):581–621, December 1990.

[Tec] NetMind Technologies. Personalized solutions. `http://www.netmind.com`.

[TEK99] TEK-TOOLS. KAWA - The Simple Yet Powerful IDE, 1999. `http://tek-tools.com/kawa`.

[Ten96] Roy Tennant. The berkeley digital library sunSITE. Technical Report february96-tennant, D-Lib Magazine, February 15, 1996. `http://www.dlib.org/dlib/february96/ucb/02tennant.html`.

[Tol99] R. Tolksdorf. XML und darauf basierende Standards: Die neuen Auszeichnungssprachen des Web. *Informatik Spektrum*, 22(6), 1999.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 1974.

[WMMM90] S. Wu, U. Manber, G. Myers, and W. Miller. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35:317–323, 1990.

[WWW00] WWW Consortium (W3C). Document Object Model (DOM), 2000. `http://www.w3.org/DOM/Activity.html`.

[ZD96] Justin Zobel and Philip Dart. Phonetic string matching: Lessons from information retrieval. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Multimedia IR, pages 166–172, 1996.

[ZS89] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18:1245–1262, 1989.

[ZSS92] Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42(3):133–139, May 1992.

[ZWS96] Zhang, Wang, and Shasha. On the editing distance between undirected acyclic graphs. *IJFCS: International Journal of Foundations of Computer Science*, 7, 1996.

Erklärung

„Ich versichere, daß ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und mich nicht anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht."

Hamburg, d. 29. Februar 2000

(Christoph Spreen)