Thomas Lettow

Versionskontrolle für ein graph-orientiertes Datenbankmanagementsystem

Diplomarbeit

vorgelegt am
Lehrstuhl für Informatik 3
Universitätsprofessor Dr.-Ing. M. Nagl
RWTH Aachen

Betreuer:
Dipl.-Inform. Boris Böhlen
und
Dipl.-Inform. Erhard Schultchen

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 8. Februar 2006 Thomas Lettow

Zusammenfassung

Während der Evolution eines Softwaresystems finden Veränderungen in allen Phasen der Entwicklung statt. Versionskontrolle ermöglicht es, Zustände der Entwicklung zu speichern und bei Bedarf wiederherzustellen. Diese Fähigkeit wird im Kontext heutiger Softwareentwicklung als notwendig angesehen. Durch Versionskontrolle wird die evolutionäre Veränderung eines Systems dokumentiert und der Entwicklungsprozess unterstützt. Ein Entwickler kann seine durchgeführten Änderungen rückgängig machen und die Entwicklung basierend auf einem historischen Zustand fortsetzen. Die Betrachtung ist dabei nicht auf die Änderungen an einzelnen Dokumenten beschränkt. Da Beziehungen zwischen Dokumenten bestehen, ist es wichtig, dokumentübergreifende Gesamtzustände als Konfigurationen zu verwalten. Versionskontrolle bildet die Grundlage für eine Vielzahl weitergehender Funktionen. So kann sie die Existenz paralleler Entwicklungszweige ermöglichen und die Arbeit mehrerer Entwickler koordinieren.

Werden unterschiedliche Dokumentarten von separaten Werkzeugen bearbeitet, ist die Verwaltung von Beziehungen zwischen den Dokumenten schwierig. Deshalb verwenden integrierte Softwareentwicklungsumgebungen eine gemeinsame Datenbasis für die Verwaltung aller entstehender Dokumente. Das DRAGOS-Projekt stellt ein Datenbankmanagementsystem zur Verfügung, das an die Anforderungen solcher Entwicklungsumgebungen angepasst ist. Durch einen modularen Aufbau können Erweiterungen der Funktionalität integriert werden. Der graph-orientierte Ansatz von DRAGOS ist dafür geeignet, sowohl die Struktur unterschiedlicher Dokumente als auch ihre Beziehungen untereinander in ein gemeinsames Datenmodell abzubilden. Als Grundlage für Entwicklungswerkzeuge sollte DRAGOS Mechanismen zur Versionskontrolle anbieten.

Die vorliegende Arbeit zeigt, welche Anforderungen eine Versionkontrolle für DRAGOS erfüllen muss und welche Konzepte geeignet sind, um diese Anforderungen zu erfüllen. Das Ergebnis ist eine funktionsfähige Versionierungserweiterung, die Basis-Mechanismen zur Versionskontrolle in das graph-orientierte Datenmodell integriert. Auf Basis von DRAGOS entwickelte Werkzeuge können diese Mechanismen an ihre anwendungsspezifischen Bedürfnisse anpassen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Versionskontrolle wird bei der Softwareentwicklung nicht nur für die klassische Versions- und Konfigurationsverwaltung eingesetzt. Das Speichern und Wiederherstellen von Zuständen stellt die Basis für die Realisierung vieler weitergehender Funktionen dar, die von Entwicklungswerkzeugen angeboten werden.

Bei der Softwareentwicklung entsteht eine Vielzahl von Dokumenten, die von verschiedenen Werkzeugen erzeugt und bearbeitet werden. Obwohl sich die Dokumente in Aufbau und Struktur teilweise stark unterscheiden, sind sie dennoch durch Beziehungen und Abhängigkeiten miteinander verbunden. Um sowohl die unterschiedlichen Arten von Dokumenten als auch die Struktur ihrer Beziehungen untereinander in einem gemeinsamen Datenmodell erfassen zu können, bietet sich die Verwendung von Graphen an. Das DRAGOS Projekt [GRA] stellt ein allgemeines Graphmodell zur Speicherung von Dokumenten zur Verfügung.

Da die Dokumente ständiger Veränderung unterliegen, ist es sinnvoll, Versionskontrollmechanismen auf niedriger Ebene in das gemeinsame Datenmodell zu integrieren, was Inhalt dieser Diplomarbeit ist.

1.1 DRAGOS und Versionskontrolle

Das Ziel von DRAGOS ist die Entwicklung eines Datenbankmanagementsystems, das an die Anforderungen integrierter Softwareentwicklungsumgebungen angepasst ist. Die Dokumente beliebiger Werkzeuge können mit Hilfe von DRAGOS in einer gemeinsamen Datenbasis verwaltet werden. Das dazu bereitgestellte Datenmodell basiert auf attributierten und getypten hierarchischen Graphen. Solche Graphen sind dafür geeignet, hochstrukturierte Dokumente abzubilden und feingranulare Beziehungen zu verwalten, die sowohl zwischen den Dokumenten als auch innerhalb der Dokumente existieren können. Auf Basis des universellen Graphmodells können anwendungsspezifische Datenmodelle entwickelt werden.

Als Datenbankmanagementsystem sorgt DRAGOS für die persistente Speicherung der Graphstrukturen und stellt eine eigene Transaktionsverwaltung bereit, welche die Konsistenz der Daten sicherstellt. Neben der Transaktionsverwaltung kann eine Anwendung umfangreiche Möglichkeiten zur Ereignisbehandlung nutzen. Auf Ereignisse kann nach definierten Regeln reagiert werden. Zu diesem Zweck steht der Anwendung eine Regelmaschine zur Verfügung.

Der standardmäßige Funktionsumfang von DRAGOS lässt sich über das Einbinden von Erweiterungen um zusätzliche Funktionalität ergänzen. Hierzu zählen beispielsweise die Möglichkeit zum automatischen Transaktions-Commit und dynamische Attributauswertung.

Mechanismen zur Änderungs- und Versionsverwaltung sind standardmäßig nicht in DRAGOS integriert. Zu jedem Zeitpunkt existiert nur ein Zustand der gespeicherten Daten. Von einer Anwendung getätigte Änderungen überschreiben diesen Zustand. Es ist nicht möglich, verschiedene Zustände der Daten in Form von Versionen zu speichern und bei Bedarf wiederherzustellen. Bei der Softwareentwicklung wird diese Funktionalität jedoch benötigt. So werden häufig externe Versionskontrollsysteme wie beispielsweise das populäre CVS [Mor96] für die langfristige Verwaltung von Versionen eingesetzt. Gleichzeitig wird darüber die Arbeit mehrerer Benutzer an einer gemeinsamen Datenbasis ermöglicht.

Das Speichern und Wiederherstellen von vergangenen Zuständen bildet jedoch auch die Grundlage für weitere Funktionen, die Softwareentwicklungswerkzeuge zur Verfügung stellen. So ist die kurzfristige Verwaltung von Änderungen die Grundlage für die Realisierung von Undo/Redo-Funktionen. Da DRAGOS die Datenbasis unterschiedlicher Werkzeuge verwaltet, ist die Integration von Versionskontrollmechanismen wünschenswert.

1.2 Ziele der Arbeit

Das Ziel dieser Diplomarbeit ist die Erweiterung des DRAGOS-Projekts um Versionskontrollfunktionen. Diese sollen es ermöglichen, beliebige Zustände von DRAGOS-Graphstrukturen in Form von Versionen zu verwalten. Da DRAGOS die Grundlage für ein breites Spektrum von Anwendungen darstellt, ist es von entscheidender Bedeutung, zunächst die Anforderungen an eine Versionierungserweiterung zu erkennen und zu formulieren. Die bereitgestellten Mechanismen sollten so allgemein ausgelegt sein, dass jede Anwendung sie an

ihre individuellen Befürfnisse anpassen kann. So sollten sie zur reinen Änderungs- und Versionsverwaltung ebenso einsetzbar sein wie als Grundlage für spätere Erweiterungen, die Undo/Redo-Funktionen anbieten oder Mehrbenutzerbetrieb unterstützen. Letztgenannte Funktionen werden in dieser Diplomarbeit nicht implementiert. Sie werden jedoch als zukünftige Erweiterungsmöglichkeiten in die Anforderungsüberlegungen mit einbezogen.

Die Funktionalität des bereits erwähnten Versionskontrollsystems CVS zeigt, wie gemeinsames Arbeiten mehrerer Entwickler an einer gemeinsamen Datenbasis auf der Basis von Versionskontrolle ermöglicht werden kann. Dazu wird ein *Check-In/Check-Out*-Mechanismus [BK91] realisiert: Durch Check-Out-Operationen auf einem Server-Repository werden Versionen der Daten erzeugt, die dann im lokalen Repository verändert werden können. Hierbei ist zu bedenken, dass für diese Art von Mehrbenutzerbetrieb zusätzliche Funktionen benötigt werden, die über die Basis-Versionsverwaltungsfunktionalität hinausgehen. So baut die erwähnte Check-Out-Operation darauf auf, Teile der Datenbasis replizieren zu können, um identische Kopien in den unterschiedlichen Repositories anlegen zu können. Um die lokalen Änderungen anschließend per Check-In-Operation öffentlich zu machen, ist zudem die Möglichkeit zum Zusammenführen von Versionen erforderlich. Eine sogenannte *Merge*-Operation muss Konflikte zwischen Änderungen erkennen und möglicherweise sogar semantisch korrekt lösen können.

Die Umsetzung von Mehrbenutzerfunktionalität übersteigt in ihrer Komplexität den Umfang dieser Arbeit. Ziel ist es vielmehr, die Grundlagen für zukünftige Weiterentwicklungen zu schaffen. So sollen Konzepte zum Zusammenführen von Versionen entwickelt und erste Ansätze zur Replikation von Graphen realisiert werden.

In der Literatur existieren nur wenige Ansätze, die sich mit der Realisierung von Versionskontrolle für graphbasierte Daten beschäftigen. Neben GRAS3, dem Vorgängerprojekt von DRAGOS, zählt das an der TH Braunschweig entwickelte CoObRA zu den wichtigsten vergleichbaren Projekten. GRAS3 und CoObRA sollen deshalb in dieser Arbeit besondere Beachtung finden.

1.3 Gliederung der Arbeit

Im folgenden Kapitel 2 wird zunächst das DRAGOS Projekt vorgestellt. Dabei wird das zugrundeliegende Graphmodell sowie die Architektur und Erweiterbarkeit des Systems erläutert. Darauf aufbauend formuliert Kapitel 3, welche

Anforderungen eine Versionierungserweiterung für DRAGOS erfüllen muss. Diese werden anhand von Anwendungsszenarien hergeleitet. Kapitel 4 liefert daraufhin einen Überblick über den aktuellen Stand der Technik in Bezug auf Versionskontrollsysteme. Hierbei werden besonders die bereits erwähnten Projekte GRAS und CoObRA betrachtet.

Die Realisierung der Versionierungserweiterung wird in Kapitel 5 ausführlich dargestellt. Darin folgt auf eine Erläuterung der erarbeiteten Konzepte eine Darlegung der wichtigsten Implementierungsdetails. Die erarbeiteten Ergebnisse fasst Kapitel 6 zusammen. Dabei wird zunächst die realisierte Versionskontrolle mit den in GRAS3 und CoObRA implementierten Mechanismen verglichen. Den Abschluss der Arbeit bildet ein Ausblick auf zukünftige Erweiterungen, deren Entwicklung auf Basis der Versionierung denkbar ist.

2 DRAGOS

DRAGOS [Böh04] ist die Abkürzung für *Database Repository for Applications using Graph Oriented Storage*. Die Bezeichnung beinhaltet das Ziel des Projektes, ein auf Graphstrukturen basierendes Datenbankmanagementsystem (DBMS) zur Verfügung zu stellen. Es ist das jüngste Projekt aus der GRAS (*GRAph Storage*) Familie, zu der die Projekte RGRAS [KSW95] und GRAS3 [Bau99] gehören. DRAGOS ist eine grundsätzliche Neuentwicklung, welche die Nachteile der Vorgänger beseitigt und damit das Spektrum der Anwendungsmöglichkeiten vergrößert. Die Spezifizierung des eingesetzten Graphmodells wurde von Aspekten der Graph Exchange Language (GXL) [GXL03] beeinflusst, weshalb der Arbeitstitel des Projekts zunächst *Gras/GXL* lautete. Inzwischen ist aber erkannt worden, dass das GXL-Graphmodell einige Probleme bereitet, wenn es als Datenmodell für die persistente Datenbankspeicherung eingesetzt werden soll.

Dieses Kapitel erläutert die für diese Arbeit wichtigen Aspekte von DRAGOS und ist wie folgt gegliedert: In Abschnitt 2.1 wird zunächst die Motivation für die Entwicklung von DRAGOS sowie sein Einsatzgebiet in der Softwareentwicklung vorgestellt. Abschnitt 2.2 erläutert das zu Grunde liegende Graphmodell bevor in Abschnitt 2.3 die Architektur des Systems beschrieben wird. Das Konzept, nach dem Erweiterungen für DRAGOS realisiert werden, ist Thema von Abschnitt 2.5.

2.1 Motivation und Einsatzgebiet

DRAGOS' Haupteinsatzgebiete liegen im Bereich von integrierten Softwareentwicklungsumgebungen. An den verschiedenen Phasen der Entwicklung von Softwaresystemen ist eine große Anzahl von unterschiedlichen Werkzeugen beteiligt, die unterschiedliche Arten von Dokumenten bearbeiten. So entstehen bei der Dokumentation sowie der Anforderungsspezifikation ausformulierte Texte,

die mit einem Textverarbeitungsprogramm erstellt werden, während der Entwurfsprozess Architektur-Diagramme hervorbringt, die von graphischen Editoren bearbeitet werden.

Innerhalb der einzelnen Arbeitsbereiche stehen bereits komfortable Entwicklungswerkzeuge zu Verfügung, die auch im Stande sind, Querbezüge zwischen Dokumenten der gleichen Art zu verwalten. Im Bereich der objekt-orientierten Programmierung ist dies Grundlage für Refactoring-Operationen [Opd92]. So kann beispielsweise die Änderung eines Methodennamens klassenübergreifend propagiert werden, wenn die Entwicklungsumgebungen alle Referenzen auf diese Methode verwaltet. Gleichzeitig können so Konsistenzprobleme erkannt werden, die zum Beispiel entstehen, wenn die Signatur einer Methode geändert wird.

Bei der Entwicklung integrierter Softwareentwicklungsumgebungen werden die einzelnen Arbeitsbereiche in einer gemeinsamen Umgebung zusammengeführt. Dabei soll auch die Verwaltung von Querbezügen zwischen Dokumenten unterschiedlicher Struktur möglich sein. Zum Beispiel trägt eine programmierte Klasse zur Implementierung eines Teilsystems bei, das in einem Entwurfsdiagramm spezifiziert ist. Die Verwaltung dieser Beziehung kann zum Beispiel genutzt werden, um Inkonsistenzen zu erkennen, die durch Rückgriffe in die Entwurfsphase entstehen. Wird in Folge eines zu spät erkannten Fehlers das Entwurfsdiagramm nachträglich geändert, sollte die Entwicklungsumgebung die Konsequenzen solcher Rückgriffe erkennen können. Querbezüge existieren auch auf feingranularer Ebene. Zum Beispiel muss sich eine Methode, die im Entwurf definiert wird, auch in der Implementierung wiederfinden lassen.

Voraussetzung hierfür ist die Datenintegration der verschiedenen Werkzeuge. Diese kann erfolgen, indem alle Dokumente in einer gemeinsamen Datenbasis gespeichert werden. Es lassen sich zwei Anforderungen für eine solche Datenbasis formulieren: Zum einen muss sie dafür geeignet sein, die unterschiedliche Komplexität aller Arten von auftretenden Dokumenten abgebilden zu können. Zum anderen muss sie auch Querbezüge zwischen beliebigen Dokumenten auf feingranularer Ebene ermöglichen. DRAGOS findet bereits Einsatz im Graphersetzungssystem PROGRES (PROgrammed Graph REwriting Systems) [Sch91], das zur Spezifikation graphbasierter Werkzeuge benutzt wird. Die Codegenerierung von PROGRES wurde so angepasst, dass die spezifizierten Systeme DRAGOS zur Graphenspeicherung verwenden. Darüber hinaus ist DRAGOS bislang nicht in der Praxis eingesetzt worden. Mit dem Vorgängerprojekt GRAS wurden jedoch schon umfangreiche Erfahrungen im Praxiseinsatz gesammelt. GRAS wurde für den Einsatz in der integrierten Softwareentwicklungsumgebung IPSEN [Nag96] entwickelt. Die Erfahrungen zeigen, dass eine graphori-

entierte Datenbank geeignet ist, um die Anforderungen von Softwareentwicklungswerkzeugen zu erfüllen.

2.2 Graphmodell und Graphschema

Graphen sind ein geeignetes Mittel, um die Struktur beliebiger Dokumente abbilden zu können. Die Bedeutung des Begriffes *Graph* unterscheidet sich in den Auslegungen unterschiedlicher graphbasierter Anwendungen teilweise erheblich. So benutzt eine Anwendung einfache Konstrukte aus Knoten und Kanten, während eine andere hierarchisch geschachtelte Graphen und n-äre Beziehungen zwischen Knoten zulässt. Die Eigenschaften der verwendeten Graphen werden formal in einem *Graphmodell* beschrieben. Darüber hinaus erlaubt die Definition eines *Graphschemas* die Formalisierung weiterer Einschränkungen des Graphmodells.

Neben einer mathematischen Notation gibt es verschiedene Möglichkeiten für die graphische Spezifikation von Graphmodell und Graphschema. Diese erleichtern das intuitive Verständnis komplexer Modelle. So werden in den folgenden Abschnitten UML-Klassendiagramme verwendet, um Graphmodell und Graphschema von DRAGOS zu veranschaulichen.

2.2.1 Graphmodell

DRAGOS verfolgt den Ansatz, ein möglichst allgemein gehaltenes Graphmodell bereitzustellen, das leicht an die Anforderungen unterschiedlicher Applikationen angepasst werden kann. Es ist in Form eines UML-Klassendiagrammes in Abbildung 2.1 dargestellt.

In einem Graphpool, der durch die Klasse `GraphPool` realisiert ist, werden beliebig viele Graphen verwaltet. Diese werden durch ihre Rolle identifiziert, die innerhalb des Graphpools eindeutig sein muss. Die Klasse `Graph` erbt von der Oberklasse `GraphEntity`. Diese definiert die Eigenschaften, die für alle Graphelemente zutreffen, nämlich Typisierbarkeit, Attributierbarkeit und Meta-Attributierbarkeit.

Typisierbarkeit bedeutet, dass jedes Graphelement einen Typ besitzt. Dies wird durch eine Referenz auf eine Graphelementklasse (`GraphEntityClass`) dargestellt. Graphelementklassen werden im Graphschema definiert, das im nächsten Abschnitt näher erklärt wird. Im Folgenden werden Graphelemente auch

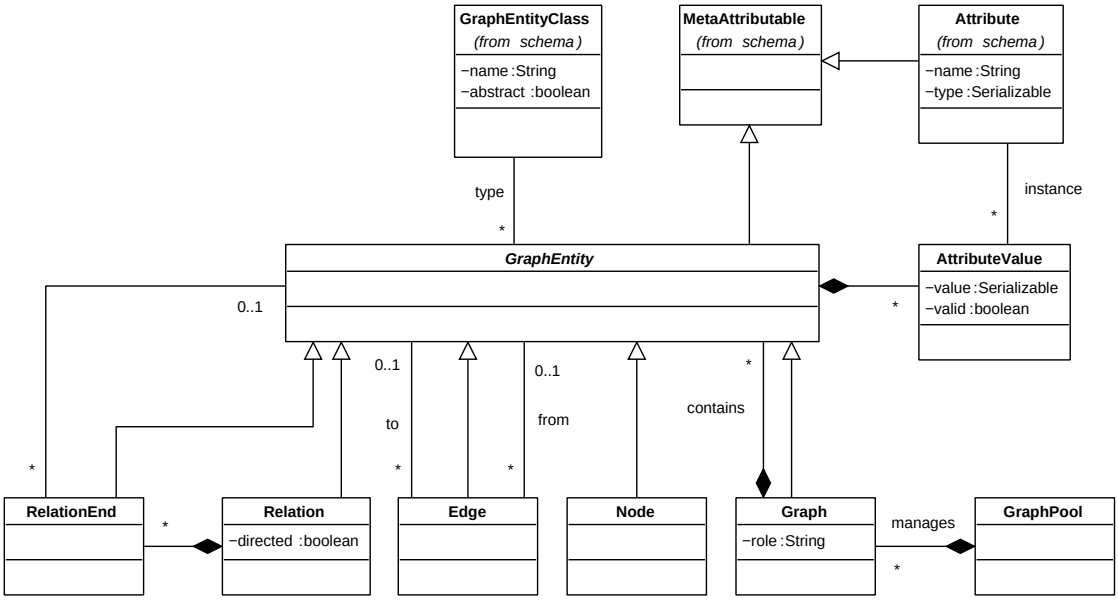

Abbildung 2.1: Das DRAGOS Graphmodell

als *Instanzen* ihrer Graphelementklasse bezeichnet. Untypsierte Graphmodelle sind indirekt realisierbar, indem bei der Erzeugung der Graphelemente implizit Graphelementklassen erzeugt werden, die von der Anwendungssicht verborgen bleiben.

Ebenfalls im Graphschema definiert werden die Attribute, die einem Graphelement zugeordnet sind. Um zusätzliche Informationen in einem Graphelement zu speichern, können diese Attribute mit Werten belegt werden. Attributwerte können sowohl einzelne Instanzen als auch Kollektionen beliebiger serialisierbarer Java-Klassen sein. Die Identifizierung von Attributen erfolgt über die im Graphschema definierten Instanzen der Klasse `Attribute`.

Erweiterungen von DRAGOS, auf die im Abschnitt 2.5 näher eingegangen wird, und Anwendungsgraphmodelle müssen häufig zusätzliche Verwaltungsinformationen für Graphelemente speichern. Diese sind nur für die jeweilige Erweiterungs- bzw. Graphmodellschicht von Bedeutung und sollten von der Anwendungssicht verborgen sein. Aus diesem Grund wurden sogenannte *Meta-Attribute* eingeführt, die wie Attribute mit beliebigen serialisierbaren Werten belegt werden können. Im Unterschied zu Attributen werden sie jedoch nicht im Graphschema definiert. Sie stehen nicht nur für jedes Graphelement sondern darüber hinaus auch für Attribute zur Verfügung. Diese Eigenschaft kann genutzt werden, um beispielsweise Layout-Informationen zu speichern.

Graphen enthalten eine beliebige Anzahl von Graphelementen. Hierzu zählen

Knoten (`Node`), Kanten (`Edge`) und Relationen (`Relation`). Zwischen Graphen und den in ihnen enthaltenen Graphelemente besteht eine Vater-Kind-Beziehung. In Folge dessen besitzen alle Graphelemente bis auf die direkt im Graphpool erzeugten Graphen einen eindeutigen Vater-Graphen (`parent`). Graphen können auch andere Graphen enthalten, wodurch die Modellierung hierarchischer Graphen ermöglicht wird. Die enthaltenen Kind-Graphen werden auch als *Untergraphen* ihres Vater-Graphen bezeichnet.

Relationen besitzen belieblig viele Relationsenden mit denen sie Graphelemente referenzieren. Sie könnten demnach auch als Hyperkanten [EKMR99] bezeichnet werden. Die Klasse `RelationEnd` ist eine Unterklasse von `GraphEntity`, wodurch Relationsenden die gleichen Eigenschaften haben wie alle anderen Graphelemente. So ist es auch möglich, mit Relationsenden andere Relationsenden zu referenzieren. Darüber hinaus können Relationsenden gerichtet sein, wobei die Richtung entweder ein- oder auslaufend sein kann. Relationen, die kein gerichtetes Relationsende besitzen, sind ungerichtet. Welche Art von Graphelementen von einem Relationsende referenziert werden kann, wird im Graphschema definiert.

Kanten sind in DRAGOS durch eine eigene Klasse repräsentiert, obwohl die Möglichkeit besteht, sie als Relationen mit jeweils zwei Relationsenden zu modellieren. Dies liegt zum einen darin begründet, dass viele Graphmodelle Kanten explizit definieren. Zum anderen wird so der Aufwand vermieden, der durch die zusätzliche Definition von zwei Relationsenden entsteht. Kanten besitzen immer zwei Referenzen auf Graphelemente, von denen die eine *Quelle* (source) und die andere *Ziel* (target) genannt wird. Eine Kante kann gerichtet oder ungerichtet sein. Diese Eigenschaft wird im Graphschema definiert. Eine gerichtete Kante hat eine Richtung, die von der Quelle zum Ziel führt. Im Schema wird getrennt definiert, welche Graphelemente als Quelle und Ziel in Frage kommen. Für ungerichtete Kanten wird diese Unterscheidung nicht vorgenommen.

Kanten und Relationsenden können Graphelemente referenzieren, die in einem vom Vater-Graphen verschiedenen Graphen enthalten sind. Derartige Verbindung werden als *graphübergreifend* bezeichnet. Wie aus den in Abbildung 2.1 dargestellten Kardinalitäten ersichtlich ist, sind hängende Kanten nicht ausgeschlossen. Sie stellen jedoch eine Inkonsistenz dar, die nach Transaktionsende behoben sein muss. Um der Anwendung möglichst wenig Einschränkungen aufzuerlegen, ist es aber sinnvoll, Zwischenzustände mit hängenden Kanten zuzulassen.

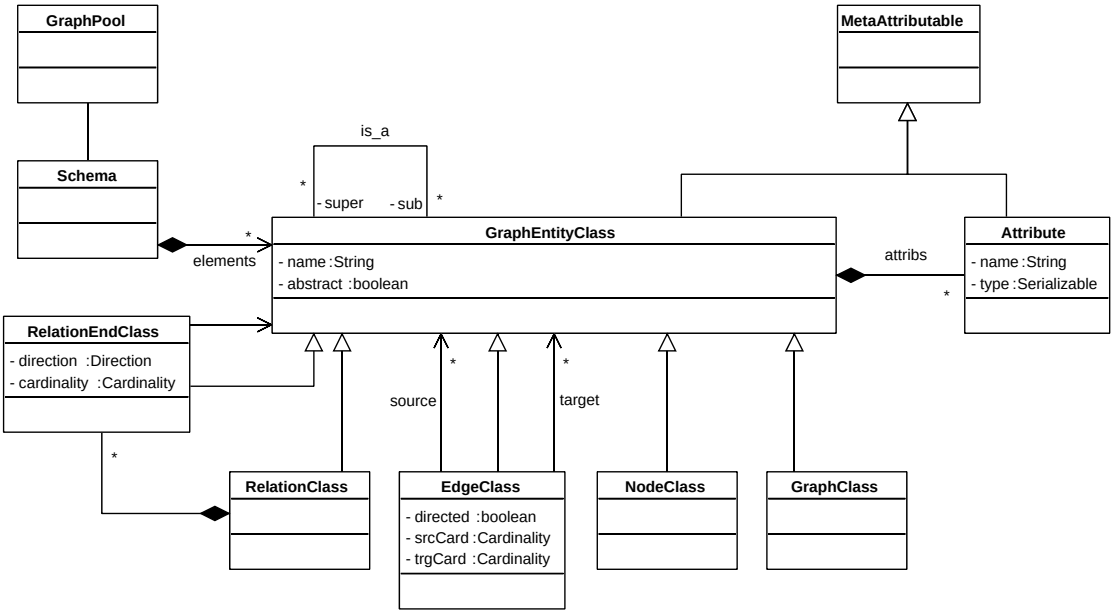

Abbildung 2.2: Das DRAGOS Graphschema

2.2.2 Graphschema

Alle Graphelemente sind getypt und somit Instanzen von Graphelementklassen. Diese Klassen werden im Graphschema definiert, das im Folgenden auch kurz *Schema* genannt wird. Nach der Definition wird das Schema als unveränderlich betrachtet und ist deshalb nicht Gegenstand der Versionskontrolle. DRAGOS könnte jedoch später aufbauend auf der Versionierungserweiterung um Aspekte der Schema-Evolution erweitert werden.

Das Klassendiagramm des DRAGOS-Graphschemas ist in Abbildung 2.2 dargestellt. Jeder Graphpool besitzt genau ein Graphschema (`Schema`). Jede darin definierte Graphelementklasse kann über einen eindeutigen Namen identifiziert werden. Um Informationen in Attributen von Graphelementen speichern zu können, müssen diese Attribute zuvor für die entsprechenden Graphelementklassen deklariert worden sein. Ein Attribut besitzt sowohl einen Bezeichner als auch einen Typ, der die Art von Daten bestimmt, die das Attribut speichern kann. Instanzen der betreffenden Klasse können dem deklarierten Attribut einen Wert des definierten Typs zuweisen. Graphelementklassen und Attribute sind meta-attributierbar (`MetaAttributable`). Diese Eigenschaft macht das Speichern von Verwaltungsinformationen in Meta-Attributen möglich, wodurch zum Beispiel Erweiterungen und Anwendungsgraphmodelle die Funktionalität des Schemas erweitern beziehungsweise spezialisieren können.

Für Kanten und Relationsenden werden bei der Klassendefinition noch weitere Eigenschaften festgelegt, die sich darauf beziehen, welche Klassen von Graphelemente ihre Instanzen referenzieren dürfen. Zudem wird für Kantenklassen durch die *Kardinalität* definiert, wie groß die Menge der Instanzen sein darf, die ein Graphelement gleichzeitig referenzieren. Die Kardinalität besteht dabei aus zwei Werten, von denen einer die untere Grenze und der andere die obere Grenze der Mengengröße festlegt. Für gerichtete Kanten wird die Klasse und die Kardinalität von Quelle und Ziel getrennt festgelegt, während bei ungerichteten Kanten diese Unterscheidung entfällt. Für Klassen von Relationsenden wird ebenfalls eine Kardinalität definiert, die festlegt, wie viele ihrer Instanzen in einer Relation enthalten sein dürfen.

DRAGOS bietet rudimentäre Unterstützung von Vererbung zwischen Graphelementklassen, die verschiedene Applikationen an ihre individuellen Bedürfnisse anpassen können. Graphelementklassen können belieblig viele Oberklassen besitzen, wodurch Mehrfachvererbung möglich ist. Es existieren keine Einschränkungen bezüglich der Art der Unterklassen, wodurch zum Beispiel eine Kantenklasse von einer Graphklasse erben kann. Zyklische Vererbung ist jedoch nicht erlaubt. Standardmäßig realisiert ist die von objektorientierten Programmiersprachen gewohnte Eigenschaft, dass Attribute, die für eine Graphelementklasse deklariert sind, auch in ihren Unterklassen gültig sind. Da Attribute über Instanzen der Klasse `Attribute` identifiziert werden, müssen ihre Namen nur in der Graphelementklasse selbst, nicht jedoch in der Vererbungshierarchie eindeutig sein. Die Anfrage-Methoden des Graphmodells, die sich auf bestimmte Graphelementklassen beziehen, sind so ausgelegt, dass sie wahlweise Unterklassen der gegebenen Graphelementklasse mitberücksichtigen. Auf diese Weise kann die Anwendung beispielsweise alle Knoten eines Graphen finden, die eine bestimmte Knotenklasse oder eine ihrer Unterklassen instanziieren.

Graphelementklassen können als *abstrakt* definiert werden. In diesem Falle können sie nicht von Graphelementen instanziiert werden. Sie können jedoch instanziierbare Unterklassen besitzen, und so zur Modellierung von gemeinsamen Eigenschaften verwendet werden.

2.3 Architektur

Die Systemarchitektur von DRAGOS ist in Abbildung 2.3 in Form eines Schichtenmodells dargestellt.

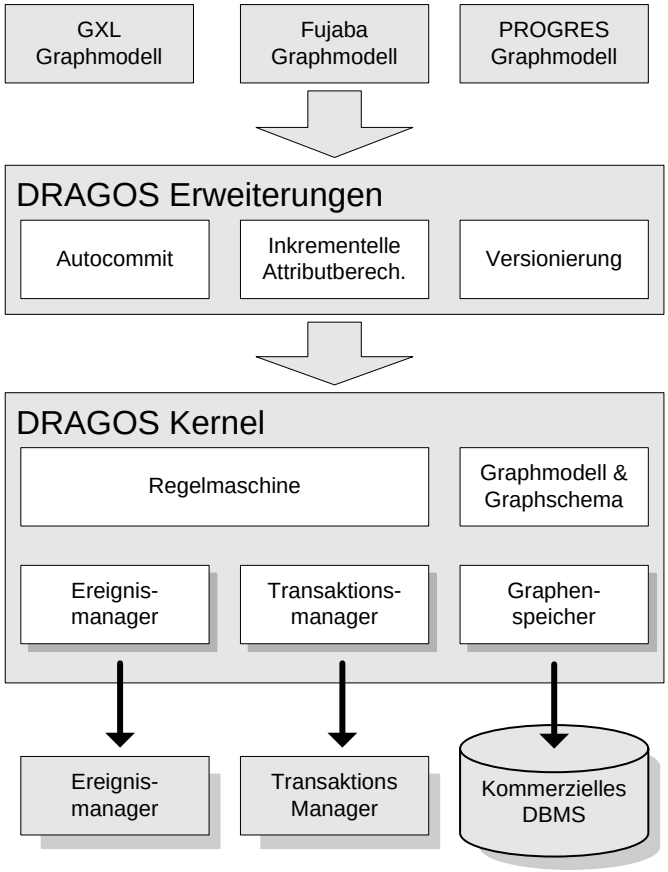

Abbildung 2.3: Schichtenmodell der DRAGOS-Systemarchitektur

Das bereits erläuterte Graphmodell sowie das Graphschema gehören zum Kernel des Systems. Darin befinden sich auch die allgemeinen Teile von Transaktions- und Ereignismanagement. Graphveränderungen stellen Ereignisse dar, auf welche die Anwendung reagieren kann. Eine Regelmaschine ermöglicht zudem die Definition von Regeln, nach denen automatisch auf bestimmte Ereignisse reagiert wird.

Während die Vorgängerprojekte GRAS3 und RGRAS auch die gesamte Datenbank-Funktionalität implementieren, nutzt DRAGOS fremde Datenbanksysteme zur Speicherung. Graphmodell, Graphschema und spezielle Teile des Transaktions- und Ereignismanagement sind deshalb im Kernel nur als Sammlung von Java-Schnittstellen vorhanden. Deren unterschiedliche Realisierungen für die externen Datenspeicher sind in der unter dem Kernel befindlichen Schicht zu finden. Neben einer In-Memory-Implentierung existieren bislang unter anderem Anbindungen an das objekt-relationale DBMS PostgreSQL [Mom00] so-

wie das Java Data Objects (JDO) [Roo02] Rahmenwerk.

Erweiterungen des DRAGOS-Graphmodells werden aufbauend auf dem Kernel realisiert. Sie werden im nachfolgenden Kapitel näher erläutert. Applikationsgraphmodelle können Erweiterungen einbinden, um zusätzliche Funktionalität nutzen zu können. Als Beispiele für Applikationsgraphmodelle sind in Abbildung 2.3 die Graphmodelle von PROGRES, FUJABA und GXL erwähnt, jedoch sind auch beliebige andere Modelle realisierbar.

2.4 Einsatz von DRAGOS

Eine Anwendung nutzt DRAGOS, indem sie einen Graphpool zur Speicherung ihrer Anwendungsdaten verwendet. Vor dem Zugriff muss der Graphpool zunächst geöffnet werden. Beim Öffnen eines Graphpools wird die Verbindung zum verwendeten externen Datenbankbanksystem hergestellt und eventuell benötigte Initialisierungsvorgänge gestartet. Beim Vorgänger GRAS3 mussten auch einzelne Graphen geöffnet werden, um mit ihnen arbeiten zu können. Dies ist in DRAGOS nicht nötig. Das Schließen eines Graphpools sorgt für die Ausführung datenbankspezifischer Vorgänge vor dem Trennen der Verbindung zum Datenbanksystem. Unter anderem werden alle eventuell noch laufenden Transaktionen abgeschlossen.

Nach jedem Transaktions-Commit wird überprüft, ob für alle im Zuge der Transaktion veränderten Teile des Graphpools Konsistenz gewährleistet ist. Die Prüfungen, die dabei vorgenommen werden, können durch Bereitstellung neuer `GraphPoolChecker`-Klassen erweitert werden. Standardmäßig wird geprüft, ob hängende Kanten oder Relationsenden existieren und ob alle Kardinalitätsbedingungen erfüllt werden.

2.5 Erweiterungen

Das Graphmodell von DRAGOS enthält die Basisfunktionalität zum Erzeugen und Verändern von Graphstrukturen sowie zur Formulierung von Anfragen. Alle Funktionen, die über diese Basisfunktionalität hinaus gehen, wurden bewusst nicht in den Kernel integriert, wodurch dieser sehr schlank bleibt. Das Erweiterungskonzept ermöglicht es, das Standard-Graphmodell um zusätzliche Funktionalität zu ergänzen. So existiert zum aktuellen Zeitpunkt bereits

eine *Autocommit*-Erweiterung, die für die Applikation das Transaktionsmanagement übernimmt, indem sie für alle Graphveränderungen automatisch neue Transaktionen erzeugt. Eine Erweiterung für dynamische Attributauswertung ermöglicht es, Attributwerte zur Laufzeit anhand von Berechnungsvorschriften inkrementell aus anderen Attributwerten abzuleiten.

Ein Vorteil des Erweiterungskonzepts ist, dass Applikationen nur die Erweiterungen in den Kernel einbinden müssen, die sie auch verwenden möchten. Dynamische Attribute werden beispielsweise nicht von allen Anwendungen benötigt, genauso wie auch Anwendungen denkbar sind, die das Transaktionsmanagement selbst implementieren und auf die Autocommit-Fähigkeit verzichten können. Da jede zusätzliche Funktionalität auch zusätzliche Ressourcen in Anspruch nimmt, stellen flexibel einzubindende Erweiterungen einen wesentlichen Beitrag zur Optimierung des Ressourcenverbrauchs dar.

Der zweite Vorteil dieses Konzeptes ist, dass es Raum für beliebige zukünftige Erweiterungen lässt. So können Funktionen, an denen durch neue Anwendungsgebiete Bedarf entsteht, zu einem späteren Zeitpunkt ins Graphmodell aufgenommen werden. Darüber hinaus wird durch die strikte Funktionstrennung zwischen dem Kernel und den einzelnen Erweiterungen eine klar strukturierte Architektur mit leicht wartbaren Teilsystemen geschaffen.

2.5.1 Realisierung von Erweiterungen

Erweiterungen von DRAGOS werden realisiert, indem alle Klassen des Graphmodells sowie des Graphschemas implementiert und um zusätzliche Funktionen beziehungsweise Methoden ergänzt werden. Die Realisierung der Erweiterung muss so ausgelegt sein, dass sie andere Erweiterungen kapseln kann. Durch dieses *Wrapper*-Konzept lassen sich Erweiterungen beliebig miteinander kombinieren und transparent in das Graphmodell einbinden. Das Konzept ist in Abbildung 2.4 anhand eines Beispiels veranschaulicht. Darin ist ein Anwendungsgraphmodell dargestellt, dass die Funktionalität der Erweiterungen zur Versionierung sowie zur inkrementellen Attributauswertung nutzt.

Die Reihenfolge, in der die Erweiterungen eingebunden werden, ist dabei von wesentlicher Bedeutung. Nur die erste eingebundene Erweiterung greift direkt auf die Funktionalität der Graphmodell-Implementierung zu. Alle zusätzlichen Erweiterungen bauen aufeinander auf und nutzen ausschließlich entweder ihre eigene Funktionalität oder die der direkt darunterliegenden Erweiterung. Im

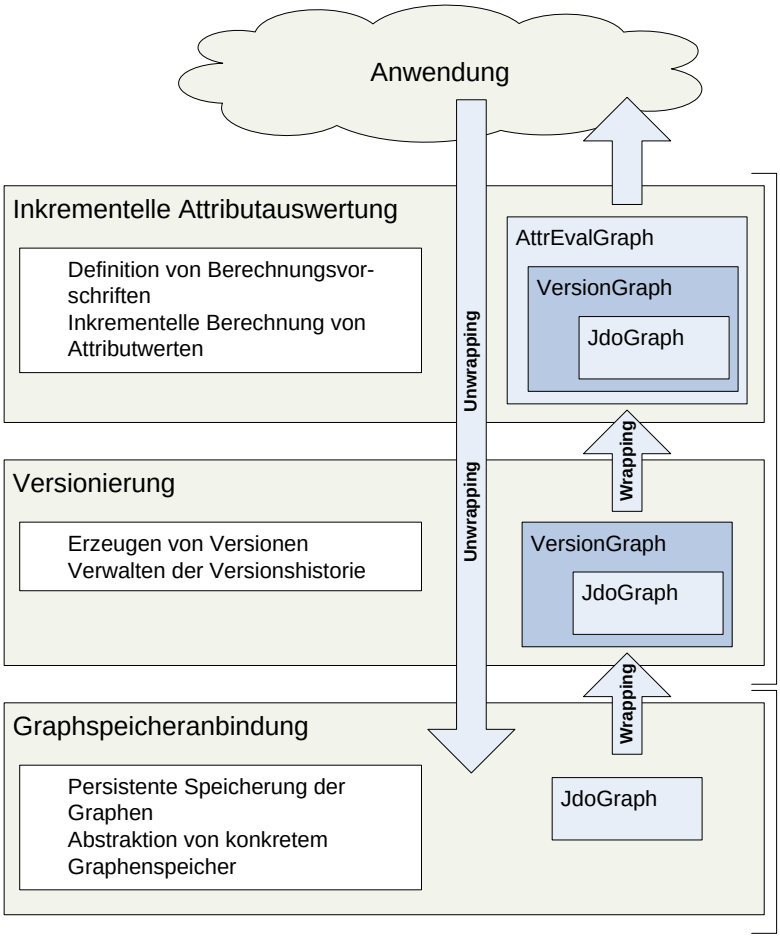

Abbildung 2.4: Integration von DRAGOS-Erweiterungen

Beispiel wären also Änderungen der Berechnungsvorschriften, welche die inkrementellen Attributauswertung bestimmt, automatisch Gegenstand der Versionierung.

Die Anwendung arbeitet mit in Erweiterungen gekapselten Graphmodell- und Graphschema-Objekten. Sie kann Methoden aus allen Erweiterungsschichten verwenden, indem sie Objekte schichtweise entpackt (*unwrapping*), bis die benötigte Erweiterungsklasse gefunden wird. Eine Anfrage an eine beliebige Schicht wird so bis zur Speicherungsschicht durchgeleitet. Zurückgegebene Objekte dieser Speicherungsschicht werden wiederum in der festgelegten Reihenfolge in Objekte der verwendeten Erweiterungen verpackt (*wrapping*). Im Beispiel ist dies für einen Graphen aus der JDO-Graphmodell-Impelementierung dargestellt. Das in der Speicherschicht enthaltene `JdoGraph`-Objekt wird durch den Wrapping-Prozess zunächst von einem `VersionGraph`-Objekt gekapselt,

welches dann wiederum von einem `AttrEvalGraph`-Objekt gekapselt wird.

DRAGOS benötigt bestimmte Informationen über eine zu nutzende Erweiterung, die mittels einer *Erweiterungsbeschreibung* zur Verfügung gestellt werden. Diese besteht aus folgenden Teilen:

1. **Wrapper-Objekt**: Jede Erweiterung stellt ein Wrapper-Objekt bereit. Seine einzige Aufgabe besteht darin, beliebige Graphelement-Objekte in die entsprechenden erweiterungsspezifischen Objekte zu verpacken. Bei der Registrierung einer Erweiterung wird ihr Wrapper-Objekt in die Wrapper-Liste des Graphpools eingefügt. Beim Zurückgeben von Graphelementen wird diese Wrapper-Liste der Reihe nach abgearbeitet.

2. **Erweiterungsspezifischer GraphPoolChecker**: Falls auf Ebene der Erweiterung unterschiedliche Anforderungen an die Datenkonsistenz bestehen, kann diese durch spezifische Prüfvorgänge sichergestellt werden. Dabei werden die Konsistenzprüfungen tieferliegender Erweiterungen oder die Standard-Prüfungen des Graphmodells nicht ersetzt.

3. **Erweiterungsspezifischer GraphPool**: Falls die Erweiterung es erfordert, dass die Funktionalität der `GraphPool`-Klasse erweitert wird, kann das Graphpool-Objekt ähnlich wie Graphelemente durch ein erweitertes Objekt gekapselt werden.

Üblicherweise müssen Erweiterungen Zusatzinformationen persitent speichern, was durch die Nutzung von Meta-Attributen erfolgen kann. Darüber hinaus ist es auch möglich, zusätzliche Graphstrukturen anzulegen. Um diese vor der Anwendung und anderen Erweiterungen zu verbergen, lassen sich die Klassen `GraphPool` und `Schema` um Filterfunktionen ergänzen. Dadurch wird es möglich, Graphen sowie Graphelementklassen aus Anfrageergebnissen herauszufiltern und damit unsichtbar zu machen. Die Identifikation der zu filternden Elemente erfolgt anhand von erweiterungsspezifisch reservierten Bezeichnern.

Vorteil der Integration von Erweiterungen nach dem Wrapper-Prinzip ist, dass die Standard-Graphmodellschnittstelle transparent erhalten bleibt, unabhängig davon, wie viele Erweiterungen hinzugefügt werden. Idealerweise kann also bei der Arbeit mit den Graphstrukturen das Vorhandensein von Erweiterungen außer Acht gelassen werden bis tatsächlich explizit erweiterte Funktionalität benötigt wird.

2.6 Zusammenfassung

Sowohl das Graphmodell als auch das Graphschema von DRAGOS ist so allgemein gehalten, dass ein breites Spektrum von Applikationsgraphmodellen darauf abgebildet werden kann. Die Universalität wird auch im modularen Aufbau des Systems deutlich. Durch das Erweiterungskonzept kann jede zusätzlich gewünschte Funktionalität als Modul realisiert und bei Bedarf ins Graphmodell integriert werden. Das nächste Kapitel beschäftigt sich damit, welche Anforderungen eine Erweiterung erfüllen muss, die das Graphmodell um Versionskontrollfunktionen erweitert.

3 Anforderungen

Um die Anforderungen an eine Versionskontrolle für DRAGOS formulieren zu können, werden im nächsten Abschnitt zunächst einige Begriffe geklärt. Danach wird anhand eines typischen DRAGOS-Anwendungsszenarios erläutert, wie Versionskontrolle im Graphmodell eingesetzt werden kann. Abschnitt 3.3 betrachtet weitergehende Anforderungen, die geplante zukünftige Entwicklungen an die Versionierungsfunktionalität stellen. Das Ergebnis wird im darauf folgenden Abschnitt 3.4 als Anforderungen an ein zu realisierendes Versionsmodell zusammengefasst.

3.1 Versionskontrolle Allgemein

Die in dieser Arbeit verwendete Begriffe sind angelehnt an die in [CW98] verwendete Terminologie. Der Begriff *Versionskontrolle* bezeichnet im Allgemeinen die Fähigkeit, Zustände von Objekten speichern und bei Bedarf wiederherstellen zu können. Die Objekte, die Gegenstand der Versionskontrolle sind, werden im Folgenden als *versionierte Objekte* bezeichnet. Eine *Version* stellt einen gespeicherten Zustand eines versionierten Objektes dar. Jedem versionierten Objekt ist folglich ein *Versionsraum* zugeordnet, in dem die Versionen des Objektes enthalten sind. Werden Änderungen an einer Version vorgenommen und wird anschließend der neue Zustand gespeichert, entsteht eine Ableitungsbeziehung zwischen den Versionen: Die neu entstandene Version ist *abgeleitet* von der Version, die als Ausgangszustand für die Veränderung diente. Abgesehen von der Ursprungsversion hat demnach jede Version mindestens eine Vorgängerversion. Der Unterschied zwischen einer Version und einer Vorgängerversion wird als *Delta* bezeichnet. Werden Versionen als Knoten und die Ableitungsbeziehungen als gerichtete Kanten betrachtet, so bilden die Versionen eines Versionsraums einen gerichteten, azyklischen Graphen (DAG). Dieser wird als *Versionsgraph* bezeichnet.

Die Betrachtung von Versionsräumen als Graphen setzt voraus, dass Versionen als explizit abgespeicherte Zustände existieren. Dieser Ansatz wird in [CW98]

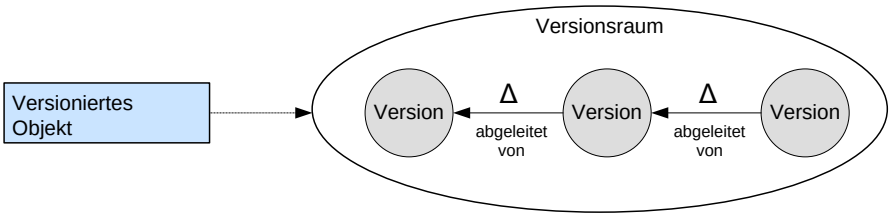

Abbildung 3.1: Versioniertes Objekt und Versionsraum

als *extensionale Versionierung* (extensional versioning) bezeichnet. Im Gegensatz dazu werden bei der *intensionalen Versionierung* (intensional versioning) Versionen anhand eines beschreibenden Ausdrucks bei Bedarf konstruiert. Im folgenden wird Versionierung nur auf Basis expliziter Versionen betrachtet.

Abbildung 3.1 zeigt ein versioniertes Objekt, dessen Versionsraum drei Versionen beinhaltet, die linear voneinander abgeleitet sind. Zwischen je zwei Versionen existierende Unterschiede sind als Deltas (Δ) an den Ableitungskanten eingezeichnet.

Es existiert eine Vielzahl von verschiedenen Ansätzen, Versionskontrolle für spezielle Datenmodelle zu realisieren. Die verschiedenen Modelle zur Versionierung lassen sich anhand einiger wichtiger Eigenschaften charakterisieren.

Versionierte Objekte Versionsverwaltungssysteme unterscheiden sich in der Granularität der Versionierung. Die Art von versionierten Objekten legt fest, auf welcher Ebene der Kompositionshierarchie Versionierung angeboten wird. So existieren Modelle, die getrennte Versionsräume für einzelne Komponenten auf feingranularer Ebene verwalten, und solche, die nur die Komposition im Ganzen versionieren.

Versionsidentifikation Versionsverwaltungssysteme vergeben automatisch eindeutige Bezeichner an neu erstellte Versionen, die oftmals zusätzlich durch ausdrucksstärkere Beschreibungen ergänzt werden können.

Struktur des Versionsraumes Die unterschiedlichen Ansätze in Bezug auf die Struktur des Versionsraumes spiegeln sich in den Einschränkungen der Versionsgraphen wieder. Abbildung 3.2 zeigt Beispiele für die verschiedenen Konzepte. Dürfen Versionen nur einen Nachfolger besitzen, überschreibt eine Änderung an einer Version alle existierenden Nachfolger-Versionen. Die Versionsgraphen sind in diesem Fall nur linear verkettete Listen, wie in Graph (A)

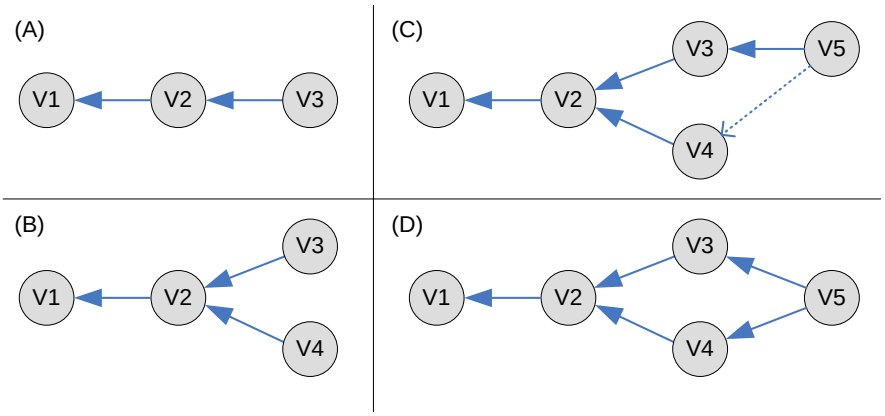

Abbildung 3.2: Struktur des Versionsraums

dargestellt ist. Werden hingegen mehrere Nachfolger, also *Varianten* von Versionen, zugelassen, nehmen die Versionsgraphen die Form von Bäumen an. In Graph (B) sind zwei Versionen von Version *V2* abgeleitet und bilden deshalb Varianten.

Einige Versionsmodelle erlauben das Zusammenführen von Varianten durch Merge-Operationen. In Bezug auf die Struktur des Versionsraum existieren dabei zwei verschiedene Ansätze. In Graph (C) wurden die existierenden Varianten *V3* und *V4* zusammengeführt. Im Versionsraum wird jedoch nur eine einzige Vorgängerbeziehung der so entstandenen Version *V5* verwaltet, wodurch die Baumstruktur des Versionsgraphen erhalten bleibt.

In der in Graph (D) dargestellten Situation werden nach einer Merge-Operation Vorgängerbeziehungen zu beiden beteiligten Versionen verwaltet. Für jede Version kann so nachvollzogen werden, ob sie durch das Zusammenführen von Versionen entstanden ist. Versionsgraphen sind folglich nicht mehr auf Baumstrukturen beschränkt sondern bilden zykelfreie Graphen.

Beziehungen zwischen Versionsräumen Existieren mehrere versionierte Objekte und bestehen Beziehungen zwischen ihnen, müssen eventuell auch Beziehungen zwischen den Versionsräumen berücksichtigt werden. Besteht beispielsweise eine Kompostionsbeziehung, könnte die Erzeugung einer neuen Version des Eltern-Objektes auch eine neue Version des Kind-Objektes erfordern. Diese Situation stellt eine Beziehung zwischen den Versionsräumen beider Objekte dar.

Versionierungsstrategie Mit Bezug auf die hier betrachtete extensionale Versionierung umfasst dieser Begriff das Schema, nach dem die Erzeugung und Wiederherstellung von Versionen angestoßen wird. Manuelle Versionserzeugung wird explizit durch den Benutzer initiiert, wohingegen automatische Versionserzeugung anhand von Regeln geschieht. Auch die Versionswiederherstellung kann explizit vom Benutzer gewollt oder auch im Rahmen von systeminterne Abläufen automatisch erfolgen.

Die aufgeführten Charakterisierungsmerkmale von Versionierungsmodellen sollen im folgenden anhand eines typischen DRAGOS-Anwendungsszenarios evaluiert werden.

3.2 Anwendungsszenario

DRAGOS wird hauptsächlich für die Verwaltung komplexer Dokumente eingesetzt. Die Struktur eines Dokumentes wird dabei auf sehr feingranularer Ebene in der Datenbank abgebildet. Es besteht eine Vielzahl von Möglichkeiten, dies in Form von Graphen zu modellieren. Bei Dokumenten, die durch eine Kompositionshierarchie strukturiert sind, bietet es sich an, diese mit Hilfe von hierarchischen Graphen abzubilden.

Als Anwendungsbeispiel wird ein Szenario betrachtet, in dem das zu verwaltende Dokument ein Entwurfsdiagramm für ein Softwaresystem ist. Dieses Entwurfsdiagramm könnte in der Datenbank als Graph repräsentiert sein, der auf oberster Ebene, also direkt im Graphpool, angelegt ist. Das Entwurfsdiagramm enthält eine Menge von Teilsystemen, die als Untergraphen dieses Graphen erzeugt wurden. Diese enthalten auf einer dritten Ebene Untergraphen, die Module repräsentieren. Die für einen Modul definierten Methoden könnten als Knoten in diesem Graph enthalten sein. Eine andere Möglichkeit wäre, sie ebenfalls als Graphen zu modellieren. Die Struktur würde dadurch weiter verfeinert, so dass beispielsweise die einzelnen Parameter der Methoden-Signatur durch Knoten repräsentiert werden können. Dadurch entsteht die Möglichkeit, eine Beziehung zwischen dem im Entwurf spezifizierten Methoden-Parameter und seiner Realisierung in der implementierten Klasse als Kante abzubilden.

Abbildung 3.3 stellt ein so modelliertes Entwurfsdiagramm dar. Die linke Seite zeigt das Diagramm so, wie es im Werkzeug angelegt sein könnte. Es enthält zwei Teilsysteme, die wiederum jeweils einen Modul enthalten. Die rechte Seite der Grafik zeigt die Abbildung des Entwurfsdiagramms in das DRAGOS-Graphmodell nach obiger Beschreibung. Die mit *enthält* beschrifteten Pfeile sind

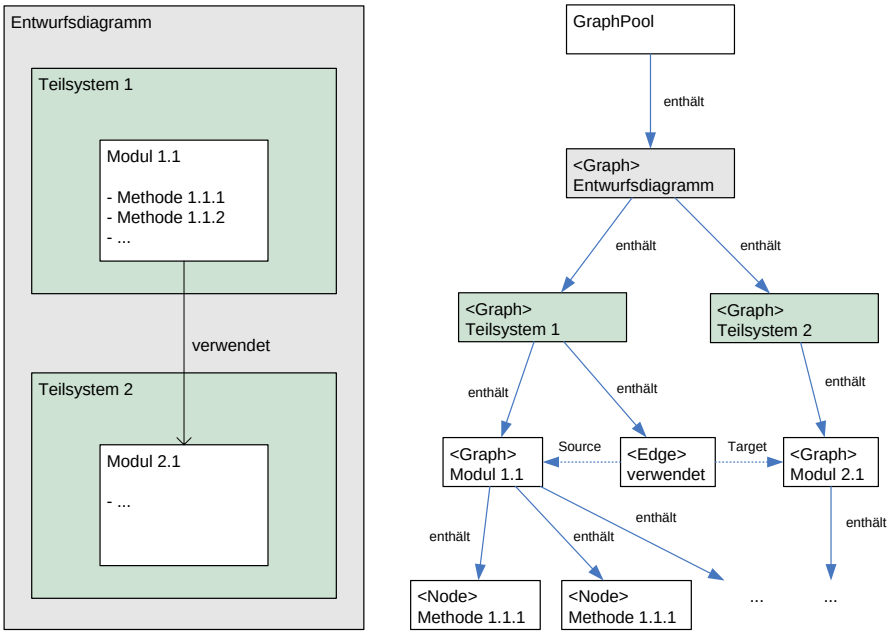

Abbildung 3.3: Abbildung eines Entwurfsdiagramms in DRAGOS

dabei keine DRAGOS-Kanten sondern veranschaulichen die Kompositionsbeziehung zwischen dem Graphpool und den Graphen auf oberster Ebene beziehungsweise beliebigen Graphen und den in ihnen enthaltenen Graphelementen. Die explizit vorhandene *verwendet*-Kante ist als im Graph *Teilsystem 1* enthaltenes Graphelement dargestellt. Quell- und Ziel-Verbindungen der Kante sind durch Pfeile dargestellt, die mit *Source* und *Target* beschriftet sind. Um die Komplexität des Beispiels zu beschränken, sind Methoden nicht als Graphen sondern als Knoten modelliert.

Versionskontrolle kann in diesem Szenario aus verschiedenen Gründen zum Einsatz kommen. Kurzfristige Änderungsverwaltung wird von vielen Entwicklungswerkzeugen in Form einer linearen Undo-/Redo-Funktion angeboten, die es erlaubt, einzelne Änderungen rückgängig zu machen. Ein solches *Undo* kann wiederum durch ein anschließendes *Redo* rückgängig gemacht werden. Würde im Beispiel *Teilsystem 2* aus dem Diagramm gelöscht werden, hätte ein anschließendes Undo zur Folge, dass die im Bild dargestellte Situation rekonstruiert wird. Dies beinhaltet, dass sowohl der enthaltene *Modul 2.1*, der ebenfalls von der Löschaktion betroffen war, als auch dessen *verwendet*-Beziehung zu *Modul 1.1* wiederhergestellt wird.

Aus Benutzersicht ist die Möglichkeit zur manuellen Rekonstruktion von explizit abgespeicherten Zuständen der Datenbasis erforderlich. Durch eine sol-

che langfristige Änderungsverwaltung können beliebig viele Versionen einer Graphstruktur persistent in der Datenbank gespeichert sein. Durch die Wahl ausdrucksstarker Bezeichner wird eine spätere Identifikation vergangener Zustände erleichtert. Wenn sich eine Folge getätigter Änderungen als fehlerhaft erweist, kann ein solcher Zustand wiederhergestellt und die Entwicklung an diesem Punkt wieder aufgenommen werden. Hierbei ist die Existenz paralleler Entwicklungszweige sinnvoll. In Bezug auf das Beispiel könnten unterschiedliche Entwicklungszweige für zwei vom Softwaresystem zu unterstützende Plattformen nötig sein, wenn im Entwurf plattformspezifische Einschränkungen berücksichtigt werden müssen.

In diesem Zusammenhang ist auch die Möglichkeit zum Zusammenführen zweier Versionen wünschenswert. Durch eine *Merge*-Operation wird ein Zustand erzeugt, der aus beiden Versionen abgeleitet ist. Die Semantik einer derartigen Operation ist von der Anwendung abhängig. Konflikte, die dabei auftreten, können oftmals sogar nur durch Benutzerinteraktion beseitigt werden. Hierbei ist es von besonderer Bedeutung, dass Unterschiede zwischen zwei Versionen erkennbar gemacht werden können. Dies impliziert, dass das Delta einer Version zum Vorgänger bei Bedarf zur Verfügung stehen muss, um der Anwendung Informationen über die getätigten Änderungen bereitzustellen.

Von einer Versionswiederherstellung sollte nicht zwingend der Zustand des gesamten Graphpools betroffen sein. Vielmehr ist es wünschenswert, Versionen einzelner Teile des Graphpools verwalten zu können, um beispielsweise nur die getätigten Änderungen an einem bestimmten Graph rückgängig zu machen. Undo/Redo-Funktionen eines Werkzeuges müssen sich beispielsweise nur auf das gerade geöffnete Dokument beziehen. Auch bei der langfristigen Änderungsverwaltung hat dies Vorteile. Im Beispiel könnte ein Zustand des Graphpools hergestellt werden, in dem der aktuellste Entwicklungsstand von *Teilsystem 1* mit einem vergangenen Entwicklungsstand von *Teilsystem 2* kombiniert wird.

Eine solche Konstellationen von Versionen unterschiedlicher Teile des Graphpools wird als *Konfiguration* bezeichnet. Konfigurationen werden von der Anwendung definiert, um konsistente Gesamtzustände verwalten zu können. Der Konsistenzbegriff ist dabei anwendungsspezifisch belegt. Die an einen Kunden ausgelieferte Konfiguration eines Softwaresystems besteht beispielsweise aus einer bestimmten Version jeder enthaltenen Klasse. Die Definition von Konfigurationen sollte von der Versionskontrolle unterstützt werden.

3.3 Weitere Anwendungsmöglichkeiten

Die Versionskontrolle auf Datenbankebene unterstützt nicht nur den Arbeitsprozess eines Entwicklers. Die Möglichkeit zur Verwaltung historischer Zustände kann auch als Grundlage für eine Reihe von erweiterten Funktionen dienen, die im Folgenden vorgestellt werden. Ihre spätere Realisierung soll bei der Konzeption der Versionskontrolle berücksichtigt werden.

DRAGOS soll in der Codegenerierung des Graphersetzungssystems PROGRES Anwendung finden. PROGRES stellt eine Spezifikationssprache bereit, mit deren Hilfe graphbasierte Werkzeuge spezifiziert werden können. Aus einer Spezifikation kann lauffähiger Code generiert werden. Wie PROGRES selbst, so verwendet auch ein generiertes Werkzeug eine GRAS3-Anbindung zur Speicherung der Graphstrukturen. In Zukunft soll die Codegenerierung von PROGRES so angepasst werden, dass stattdessen eine DRAGOS-Anbindung genutzt werden kann. Da DRAGOS in Java implementiert ist, wird auf diese Weise die Plattformunabhängigkeit für die spezifizierten Werkzeuge erreicht. Deren Funktionalität wird in der PROGRES-Spezifikation durch Graphersetzungsregeln ausgedrückt. Bei der Ausführung einer solchen Regel kommt ein nicht-deterministischer Mechanismus zum Einsatz, der Backtracking verwendet, um eine erfolgreiche Transformation durchzuführen. Die Backtracking-Funktionalität kann mit Hilfe von Versionsverwaltungsfunktionen realisiert werden. Folglich sollte die DRAGOS-Versionskontrolle alle Eigenschaften besitzen, die nötig sind, um diesen Mechanismus nachzubilden.

Eine weitere Anwendung für den Einsatz von Versionskontrolle wird deutlich, wenn die Betrachtungsweise über die des Einbenutzerbetriebs hinaus erweitert wird. DRAGOS verfügt bislang über keine Mechanismen, die verteiltes Arbeiten mehrerer Entwickler unterstützen. Klassische Systeme für das Softwarekonfigurationsmanagement (Software Configuration Management, SCM) wie beispielsweise CVS realisieren Verteilung über lokale Repositories, die gemeinsam auf ein Server-Repository zugreifen. Mittels *Check-Out*-Operation werden die Daten des Servers in ein lokales Repository übertragen. Um dies mit DRAGOS realisieren zu können, ist die Fähigkeit zur Replikation von Änderungen an Graphstrukturen erforderlich. Durch Erzeugung von Arbeitskopien der zu bearbeitenden Dokumente wird die Realisierung optimistischer Transaktionen [Vos99] ermöglicht. Diese erlauben es, auf das Sperren von Dokumenten zu verzichten. Für die Replikation von Graphstrukturen können die in der Versionsverwaltung gespeicherten Delta-Informationen benutzt werden.

Eine *Check-In*-Operation übermittelt die geänderten Daten vom lokalen Repository zum Server, um sie allen Entwicklern zugänglich zu machen. Da der

im Server-Repository gespeicherte Zustand zwischenzeitlich von anderen Entwicklern verändert sein kann, entspricht ein Check-In-Vorgang dem Zusammenführen zweier Versionen. Einzelne Änderungen am lokalen Zustand können dabei im Konflikt mit dem aktuellen Zustand des Servers stehen. Da die Semantik einer Änderung anwendungsspezifisch festgelegt wird, ist die Beseitigung dieser Konflikte nur durch die Anwendung möglich. Ihr müssen alle notwendigen Informationen zur Verfügung stehen, um die Konfliktsituation auswerten zu können. Die Deltas müssen demnach aussagekräftige Informationen über einzelne Änderungen beinhalten.

3.4 Anforderungen an das Versionierungsmodell

Die Versionskontrollfunktionen von DRAGOS werden als Erweiterung der Datenbankfunktionalität realisiert und sollen demnach sowohl von Anwendungen als auch von anderen Erweiterungen genutzt werden können. Aus diesem Grund sind nur solche Änderungen zu betrachten, die für die persistente Speicherung vorgesehen sind. Die Unterscheidung zwischen kurzfristiger und langfristiger Änderungsverwaltung ist folglich nicht notwendig. Sowohl die gespeicherten Zustände für ein kurzfristiges Undo/Redo als auch explizit als Versionen gespeicherte Zustände können in einem einheitlichen Versionsraum existieren. Das Ziel der Arbeit besteht darin, einen allgemeinen Mechanismus bereitzustellen, mit dem beliebige Anwendungsfunktionen realisiert werden können, welche die Verwaltung historischer Zustände und Änderungen voraussetzen. Der Undo/Redo-Mechanismus ist zum Beispiel eine solche Funktionalität, die später auf Basis der Versionskontrolle realisiert werden soll.

Eine wichtige Anforderung ergibt sich aus der Eigenschaft der transparenten Integrierbarkeit, die jede DRAGOS-Erweiterung besitzen muss. Durch die Nutzung der Versionskontrolle dürfen dem Graphmodell keine Einschränkungen auferlegt werden. Dies impliziert, dass eine Anwendung oder eine zusätzlich genutzte Erweiterung die integrierte Versionierungsfunktionalität ignorieren kann, wenn sie nicht explizit benötigt wird. Zu jedem Zeitpunkt muss also ein einziger konsistenter Zustand des Graphpools existieren.

Die in Abschnitt 3.1 identifizierten Charakterisierungsmerkmale für Versionierungssmodelle werden im Folgenden aufgegriffen, um die Anforderungen an die Versionskontrolle für DRAGOS zu spezifizieren.

Versionierte Objekte Wie aus der Szenario-Analyse hervorgeht, ist eine wichtige Forderung an die Versionskontrolle, Graphen in unterschiedlichen Versionen verwaltet zu können. Einzelne Graphen sollten dabei unabhängig voneinander in einen gewünschten gespeicherten Zustand versetzt werden können. Demzufolge muss jeder Graph einen eigenen Versionsraum besitzen. Diese Eigenschaft auf alle Graphelemente auszuweiten bringt dabei keine Vorteile, die den höheren Ressourcenverbrauch bezüglich Speicherbedarf und Rechenzeit rechtfertigen würden. Dies liegt darin begründet, dass unter den verschiedenen Arten von Graphelementen Graphen eine besondere Rolle einnehmen. Dadurch, dass sie andere Graphelemente enthalten können, sind sie das wichtigste Mittel zur Strukturierung von Informationen. Ihr von der Versionskontrolle zu speichernder Zustand ist nicht nur bestimmt durch Attributwerte, sondern auch durch den Zustand jedes enthaltenen Graphelementes. Der Zustand von Knoten, Kanten und Relationen wird demzufolge implizit ebenfalls von der Versionskontrolle erfasst. Sollte ihre explizite Versionierung in speziellen Anwendungsfällen dennoch erforderlich sein, lässt sich dieses Verhalten durch Einsatz von Graphen nachbilden. Knoten lassen sich zum Beispiel als leere Graphen modellieren. In Ausnahmefällen könnten Anwendungen es erfordern, Versionen von Kanten oder Relationen zu erzeugen. Diese Möglichkeit ließe sich mit Hilfe von versionierten Graphen emulieren, die genau eine Kante beziehungsweise eine Relation enthalten.

Eine weitere Anforderung in diesem Zusammenhang ist es, ein Konstrukt zur Definition von Konfigurationen zur Verfügung zu stellen. Eine Konfiguration erlaubt es, Versionen von beliebigen Graphen zusammenzufassen. Bei Bedarf sollten die gespeicherten Versionen der betreffenden Graphen wiederhergestellt werden können.

Versionsidentifikation Jede Version eines Graphen muss einen im Versionsraum eindeutigen Bezeichner besitzen. Darüber hinaus sollte das Anwendungsgraphmodell beziehungsweise der Benutzer selbstdefinierte Bezeichner für explizit erzeugte Versionen verwenden können. Dabei sollten keine Einschränkungen bezüglich des Typs des verwendeten Bezeichners bestehen. Eine einfache Zeichenkette wäre genauso denkbar wie ein komplexes Objekt, das Informationen über die Version beinhaltet. Aus technischer Sicht ist für die Speicherung jedoch die Serialisierbarkeit des Objektes gefordert.

Struktur des Versionsraumes Um parallele Entwicklungszweige zu unterstützen, sind baumartige Versionsgraphen erforderlich. Für die Realisierung eines baumartigen Undo/Redo-Mechanismus müssen bestimmte Pfade im Ver-

sionsgraphen definierbar sein, durch die der aktuelle Zweig für einen Redo-Schritt identifiziert werden kann. Auch das Zusammenführen von Versionen soll möglich sein. In Abschnitt 3.1 wurden zwei unterschiedliche Ansätze vorgestellt, die durch die Graphen (C) und (D) in Abbildung 3.2 repräsentiert sind. Das Zusammenführen von Arbeitskopien mit einem Entwicklungszweig erfordert nicht, dass mehrere Vorgängerbeziehungen der resultierenden Version existieren. Die Einschränkung der Struktur der Versionsgraphen auf Bäume ist demnach nicht aufzuheben.

Von großer Bedeutung ist zudem der Umfang der Versionsdeltas. Die in ihnen gespeicherten Informationen sollten Anwendungen in verwertbarer Form zur Verfügung stehen und um spezifische Inhalte ergänzt werden können. Deltas stellen die Grundlage für die Replikation von Graphstrukturen und einzelnen Änderungen dar. Mehrbenutzerbetrieb und Arbeit mit verteilten Repositories sind auf diese Replikationsfähigkeit angewiesen.

Beziehungen zwischen Versionsräumen Bei der Versionierung von Graphen ist die Kompositionsbeziehung zu den Untergraphen zu berücksichtigen, da diese einen eigenen Versionsraum besitzen. So müsste die Wiederherstellung einer vergangenen Version eines Graphen rekursiv die Wiederherstellung der Zustände der enthaltenen Graphen zur Folge haben. In Abbildung 3.4 ist dies anhand eines Beispiels demonstriert. Zu dem Zeitpunkt als Version *V2* von *Graph 1* gespeichert wurde, lag der enthaltene *Graph 2* in der Version *V2* vor. Diese Version wurde anschließend verworfen und stattdessen die Variante *V3* erzeugt. Der in ihr gespeicherte Zustand ist auch Teil der dritten Version des Obergraphen. Wird dessen vergangene Version *V2* wiederhergestellt, muss anschließend auch *Graph 2* in *V2* vorliegen.

Quelle und Ziel einer Kante sowie die Referenz eines Relationsendes müssen von der Versionierung erfasst werden. Durch die Erzeugung graphübergreifender Kanten und Relationen entstehen weitere Beziehungen zwischen Versionsräumen, die berücksichtigt werden müssen. Wird beispielsweise das Löschen einer graphübergreifenden Kante durch eine Versionswiederherstellung rückgängig gemacht, müssen zur Konsistenzwahrung ihre Quell- und Ziel-Verbindungen rekonstruiert werden. Dazu muss gewährleistet sein, dass die referenzierten Graphelemente nicht inzwischen gelöscht wurden beziehungsweise keine Kardinalitätsverletzungen auftreten. Die Versionskontrolle muss die Anwendung bei der Verhinderung solcher Inkonsistenzen unterstützen.

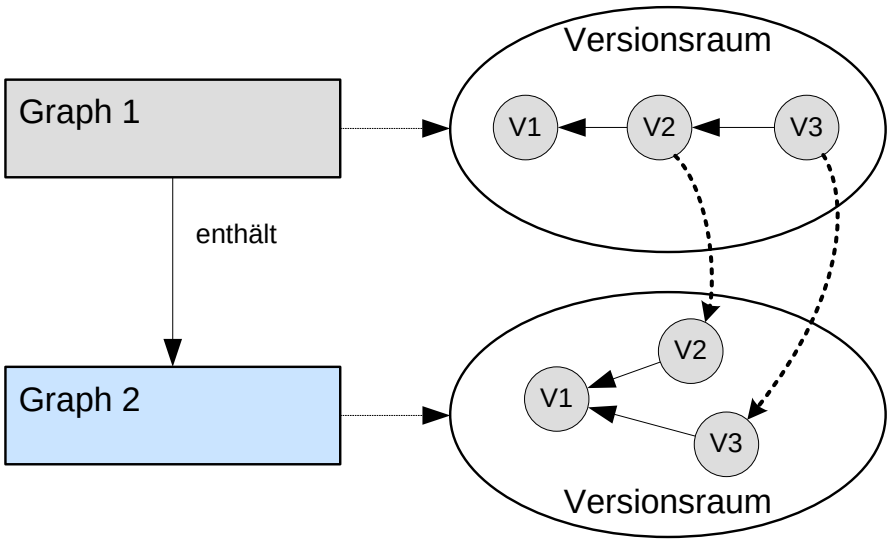

Abbildung 3.4: Beziehungen zwischen Versionsräumen

Versionierungsstrategie Da die Versionskontrollfunktionen für ein sehr weites Spektrum von Anwendungen ausgelegt sein sollen, ist die Festlegung einer Versionierungsstrategie nicht angebracht. So sind Anwendungen denkbar, die automatische Versionierung erfordern, wie auch solche, die nur die Möglichkeit zur manuellen Versionserzeugung benötigen. Ebenso sind Werkzeuge denkbar, in denen beide Möglichkeiten gleichzeitig Anwendung finden. Der Backtracking-Mechanismus von PROGRES beispielsweise benötigt ein feingranulares Undo/Redo. Dies setzt die automatische Versionserzeugung nach jeder atomaren Graphveränderung innerhalb einer Transaktion voraus. Für ein Undo/Redo auf gröberer Ebene könnte auch die Versionserzeugung nach dem Ende jeder Transaktion erfolgen. Überflüssige Versionserzeugung erhöht Speicher- und Rechenzeitbedarf und sollte deshalb vermieden werden. Die Realisierung der Versionierungsstrategie ist also der Anwendung zu überlassen.

3.5 Zusammenfassung

Die Anforderungen an die Versionskontrolle bestehen darin, ein Versionierungsmodell zu realisieren, dass sich transparent in das Graphmodell integrieren lässt. Wie das gesamte DRAGOS-Projekt, so soll auch seine Versionierungserweiterung einen sehr allgemein gehaltenen Ansatz verfolgen, der als Grundlage für spezielle Funktionen möglichst vieler Anwendungen dienen kann. Die

grundlegende Funktionalität soll darin bestehen, Zustände von Graphen speichern und wiederherstellen zu können. Den in den Versionsdeltas gespeicherten Informationen kommt dabei eine besonders wichtige Bedeutung zu. Indem jedem Graph ein eigener Versionsraum zugeordnet wird, wird die Datenbank um eine historische Dimension erweitert. Diese kann sowohl für die Realisierung von baumartigen Undo/Redo-Mechanismen als auch für die spätere Erweiterung von DRAGOS um Mehrbenutzerfähigkeit genutzt werden. Auch Anwendungsgraphmodelle, die Funktionen wie Backtracking benötigen, können auf der Versionskontrolle aufbauen.

4 Stand der Technik

Für die langfristige Änderungsverwaltung werden im Bereich der Softwareentwicklung häufig kassische Konfigurationsverwaltungssysteme wie CVS [Ber90] oder Subversion [CS02] eingesetzt, die Versionskontrolle und gemeinsame Arbeit unterstützen. Diese Systeme versionieren Textdateien, die in einer Verzeichnisstruktur enthalten sind. Darüber hinaus werden keine Beziehungen zwischen den Dokumenten berücksichtigt. Auch ihr Inhalt besitzt, abgesehen von der Einteilung in einzelne Zeilen, keine strukturellen Eigenschaften. Indem die Zeilen zeichenweise verglichen werden, werden bei der Versionserzeugung Änderungen erkannt und in Form von Delta-Informationen gespeichert. Diese Art der Versionsverwaltung ist nicht anwendbar für feingranular strukturierte Dokumente, wie sie von DRAGOS verwaltet werden. Da zudem keine Anforderungen der kurzfristigen Änderungsverwaltung erfüllt werden, können solche Systeme nicht als Vergleichssysteme dienen.

Langfristige Änderungsverwaltung findet sich auch in Datenbanksystemen, die speziell für den Einsatz in Softwareentwicklungsumgebungen entwickelt wurden. Einige dieser Systeme sind in der Übersicht über Versionsmodelle für die Konfigurationsverwaltung aufgeführt, die in [CW98] zu finden ist.

Realisierungen kurzfristiger Änderungsverwaltung sind häufig in interaktiven Anwendungen zu finden, um dem Benutzer die Möglichkeit zu geben, die Auswirkungen getätigter Aktionen rückgängig zu machen. Modelle, die über die Funktionalität linearer Undo/Redo-Modelle hinausgehen, sind in der Literatur zahlreich vorhanden. In [JEACS84] wird die Folge der ausgeführten Benutzer-Aktionen als Skript aufgefasst, das die Veränderung von einem Ausgangszustand zu einem Endzustand spezifiziert. Das vorgestellte Modell beschreibt Möglichkeiten zur Modifikation dieses Skriptes mit Hilfe der Funktionen *truncate* und *reappend*, wodurch auch das Einfügen von Aktionen in ein bestehendes Skript möglich ist. Baumartige Verzweigungen der Skriptstruktur lässt das in [Vit84] vorgestellte *US&R* (Undo, Skip & Redo) Modell zu, wodurch Redo-Operationen nicht eindeutig sind. Zwei zusätzliche Konzepte erweitern die Redo-Funktionalität: Die *Skip*-Operation erlaubt das Überspringen von Aktionen beim Redo. Darüber hinaus ist eine Redo-Operation auch möglich, wenn sich

der aktuelle Zustand in einem Blatt des Undo/Redo-Baumes befindet. US&R definiert für diesen Fall den Sprung in einen anderen Ast des Baumes. Solche erweiterten Undo/Redo-Funktionen können für die Arbeit mit primitiven Datenstrukturen hilfreich sein. Die Reihenfolge der Operationen, die bei der Arbeit mit Graphstrukturen getätigt werden, ist jedoch selten veränderbar ohne dabei die Konsistenz der Daten zu beeinträchtigen. Solche Modelle sind deshalb für DRAGOS ungeeignet.

Die Integration von kurzfristiger und langfristiger Änderungsverwaltung in ein einheitliches Modell wird in wenigen Arbeiten vollzogen. Im Nachfolgenden sollen die beiden Systeme GRAS3 und CoObRA, die im Rahmen dieser Arbeit besondere Beachtung fanden, näher betrachtet werden. Aufgrund der ähnlichen Anforderungen die sie erfüllen, lassen sich in ihnen viele Konzepte finden, die für die DRAGOS-Versionskontrolle übernommen werden können. In GRAS3, mit dem sich Abschnitt 4.1 beschäftigt, wurde eine Änderungsverwaltung für ein graphbasiertes Datenbanksystem realisiert. Das in Abschnitt 4.2 vorgestellte CoObRA realisiert hingegen ein Rahmenwerk, das Objektstrukturen um Versionskontrollfunktionen erweitert und sie gleichzeitig persistent speicherbar macht. Die Ergebnisse der Betrachtung beider Systeme werden in Abschnitt 4.3 zusammengefasst.

4.1 GRAS3

GRAS3 [Bau99] ist als Vergleichssystem besonders interessant, weil es als direktes Vorgänger-Projekt von DRAGOS für ähnliche Anwendungsgebiete geschaffen wurde. Es ist Nachfolger von RGRAS [KSW95], auf dessen Basis das Graphersetzungssystem PROGRES entwickelt worden ist. Wie RGRAS und DRAGOS stellt auch GRAS3 ein Datenbanksystem zur Verfügung, das an die Bedürfnisse von integrierten Softwareentwicklungsumgebungen angepasst ist. Wie der von *GRAph Storage* abgeleitete Name vermuten lässt, verwendet GRAS3 ein graphbasiertes Datenmodell zur Speicherung von Dokumenten, das in Abschnitt 4.1.1 vorgestellt wird. Dabei werden Gemeinsamkeiten und Unterschiede zwischen den in GRAS3 und DRAGOS implementierten Graphmodellen herausgearbeitet. Für diese Arbeit ist eine nähere Betrachtung der implementierten Funktionen zur Änderungsverwaltung von besonderer Bedeutung, die in Abschnitt 4.1.2 erfolgt. Anschließend widmet sich Abschnitt 4.1.3 dem realisierten Versionsmodell.

4.1.1 Graphmodell

Im Unterschied zu DRAGOS benutzt GRAS3 keine externen Datenbanksysteme als Datenspeicher. Die komplette Datenbankfunktionalität ist als Teil des Systems implementiert worden. Dabei sorgt ein Page-Server für die Client/Server-Verteilung auf niedriger Datenbankebene.

Ähnlich wie DRAGOS verwaltet GRAS3 Graphen in einem Pool. Graphen können neben Knoten (Node) und Kanten (Edge) auch andere Graphen enthalten, wodurch die Konstruktion beliebig tief geschachtelter hierarchischer Graphen möglich ist. Ziel und Quelle einer Kante kann ein Graph oder ein Knoten sein. Im Unterschied dazu lässt DRAGOS Verbindungen mit beliebigen Graphelemente zu. Graphübergreifende Kanten sind auch in GRAS3 möglich. Es existiert jedoch kein Konstrukt, das n-äre Relationen zwischen Graphelementen realisiert. Auch die Attributierung von Kanten ist nicht vorgesehen.

Alle Graphelemente in GRAS3 instantiieren Typen, die in einem Graphschema deklariert werden. Obgleich dieses in der Realisierung einige grundlegende Unterschiede zum DRAGOS-Graphschema aufweist, sind die generellen Konzepte ähnlich. Da, wie im nächsten Abschnitt erläutert wird, das Schema für die Änderungsverwaltung keine Bedeutung hat, sollen hier nur die wichtigsten Unterschiede zum DRAGOS-Schema erwähnt werden. Kantentypen nehmen in GRAS3 eine Sonderrolle ein, da sie weder Attribute beinhalten noch Vererbungsbeziehungen zwischen ihnen definiert werden können. Für Knotenklassen und Graphklassen ist beides hingegen möglich. GRAS3 unterscheidet Graphklassen von Graphtypen beziehungsweise Knotenklassen von Knotentypen. Klassen können von anderen Klassen erben, wobei auch Mehrfachvererbung möglich ist. Im Unterschied zu Typen können sie nicht von Graphelementen instanziiert werden. Sie dienen folglich dazu, gemeinsame Eigenschaften von Typen zusammenzufassen. Jeder Typ instanziiert genau eine Klasse und erbt dadurch deren Eigenschaften. Das Typsystem von GRAS3 ist demnach zweistufig aufgebaut.

Schemadeklarationen werden gruppiert durch Schemapakete, die jeweils einen separaten Namensraum für Schemaelemente darstellen. Durch Import-Beziehungen zwischen Schemapaketen kann das Schema modularisiert werden. Zudem legt jeder Graphtyp explizit fest, welche Knoten- und Graphtypen in seinen Instanzen vorkommen dürfen. Diese beiden Aspekte lassen sich in DRAGOS nicht wiederfinden, da sie semantische Einschränkungen darstellen, die bei Bedarf durch das Anwendungsgraphmodell realisiert werden können.

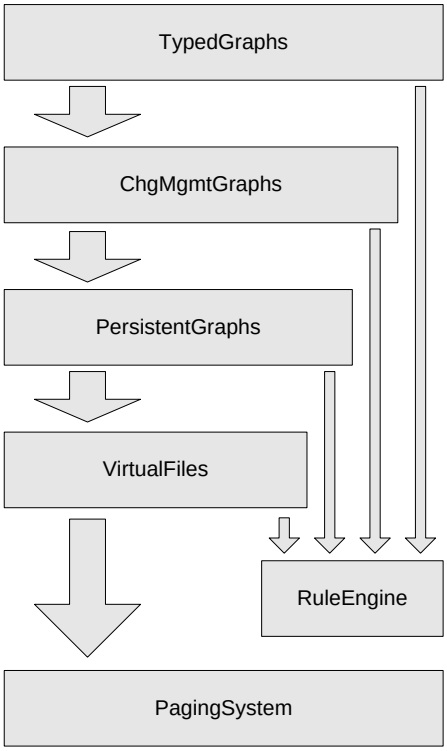

Abbildung 4.1: Systemarchitektur von GRAS3

4.1.2 Änderungsverwaltung

Wie in Abbildung 4.1 dargestellt, befindet sich das Teilsystem für die Änderungsverwaltung in GRAS3 (*ChgMgmtGraphs*) unterhalb des Teilsystems, welches das Typsystem realisiert (*TypedGraphs*). Es arbeitet direkt auf dem Teilsystem *PersistentGraphs*, das die persistente Speicherung attributierter Graph implementiert. Hierarchische Graphen werden in dem darüberliegenden Typsystem realisiert, weshalb die Änderungsverwaltung nur flache Graphstrukturen verwaltet. Attribute sind ebenfalls in *PersitentGraphs* untypisiert und speichern beliebig lange Bytefolgen als Werte.

Die langfristige Änderungsverwaltung für GRAS3 ist in [Bau99] zwar vorgesehen jedoch nicht realisiert worden. Im Folgenden wird deshalb nur die kurzfristige Änderungsverwaltung erläutert.

Die kurzfristige Änderungsverwaltung in GRAS3 [Bau99] realisiert ein baumartiges Undo/Redo-Modell. Das grundlegende Konzept basiert auf der Protokollierung aller graphverändernder Operationen. Diese werden in Form von Kommandosequenzen verwaltet. Zum Protokollieren beliebiger Graphveränderun-

gen ist die Unterscheidung von sieben unterschiedlichen Kommandos notwendig:

1. *CreateNode*: Erzeugt einen neuen Knoten
2. *DeleteNode*: Löscht einen existierenden Knoten
3. *ChangeLabel*: Ändert die Markierung eines Knoten
4. *ModifyAttribute*: Ersetzt die Bytefolge eines Attributwertes ab einem gegebenen Index durch eine neue Bytefolge
5. *TruncateAttribute*: Verkürzt die Bytefolge eines Attributwertes auf eine gegebene Länge
6. *CreateEdge*: Erzeugt eine neue Kante
7. *DeleteEdge*: Löscht eine existierende Kante

Undo-Schritte sind nur zwischen markierten Sicherungspunkten (*Checkpoints*) in der Kommandosequenz möglich. Sie markieren Zustände, deren Rekonstruierbarkeit gewünscht ist. Die Ausführung einer Undo-Operation macht die zwischen zwei Checkpoints protokollierten Operationen rückgängig, ein Redo wiederholt sie. Wird durch eine Folge von Undo-Schritten ein historischer Zustand hergestellt und dieser anschließend durch die Ausführung einer von Undo und Redo verschiedenen Operation verändert, so verzweigt der Kommandostrom am gewählten Checkpoint. Durch die so entstehende Baumstruktur bleiben alle nachfolgenden Checkpoints erhalten und können bei Bedarf wiederhergestellt werden.

Abbildung 4.2 zeigt im linken Teil eine lineare Kommandosequenz, die von einer Anwendung erzeugt wurde. Nach jedem Kommando ist ein Checkpoint in die Sequenz eingefügt. Der resultierende Graph ist im unteren Teil der Abbildung dargestellt. Das Ausführen von drei Undo-Operationen macht die letzten drei Kommandos rückgängig, woraufhin sich der Graph im Zustand des mittleren Bildes befindet, der durch den hervorgehobenen Checkpoint repräsentiert ist. Wird nun eine von Undo/Redo verschiedene Operation ausgeführt, wie beispielsweise das Erzeugen einer neuen Kante, so entsteht am aktuellen Checkpoint eine Verzweigung der Kommandosequenz. Das Resultat ist im rechten Teil des Bildes dargestellt.

In GRAS3 besitzt jeder Graph auf oberster Ebene ein eigenes Kommando-Log, in dem auch Graphveränderungen an beliebig tief geschachtelten Untergraphen protokolliert werden. Das Log hat die Form eines Checkpoint-Baums, in dem jeder Checkpoint einen historischen Zustand des Graphen darstellt. Einer dieser Zustände repräsentiert den aktuellen Zustand des Graphen, weshalb der

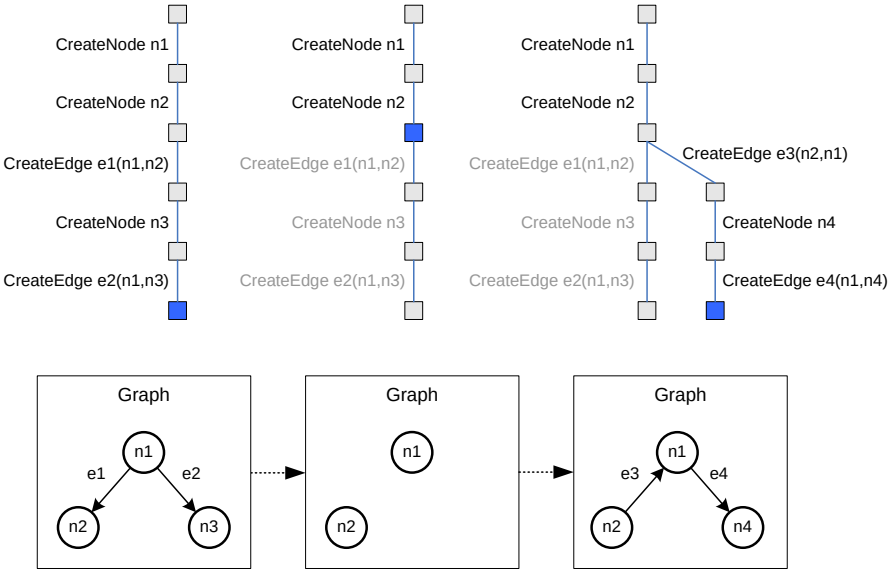

Abbildung 4.2: Beispiel für baumartiges Undo/Redo in GRAS3

zugehörige Checkpoint als *aktueller Checkpoint* bezeichnet wird. Zudem existiert zu jedem Zeitpunkt ein *aktueller Pfad*, der ausgehend von der Wurzel durch den aktuellen Checkpoint verläuft und zu einem Blatt des Baumes führt. Der Teil des Pfades von der Wurzel zum aktuellen Checkpoint enthält die Kommandosequenz, die abgearbeitet werden muss, um den Graph vom ursprünglichen in den aktuellen Zustand zu überführen.

Jeder Checkpoint hat einen eindeutigen Vorgänger. Durch eine Undo-Operation wird dieser Vorgänger zum aktuellen Checkpoint. Da ein Checkpoint mehrere Nachfolger haben kann, gibt es neben dem einfachen Redo, das sich auf den Nachfolger im aktuellen Pfad bezieht, drei weitere Redo-Operationen. Diese unterscheiden die Nachfolger anhand der Reihenfolge ihres Einfügens in den Checkpoint-Baum:

1. *RedoNext*: macht den nächst jüngeren Nachfolger zum aktuellen Checkpoint.
2. *RedoPrev*: bezieht sich auf den nächst älteren Nachfolger.
3. *RedoIth*: wählt den *i*-ten Nachfolger aus.

Bietet eine GRAS3-Anwendung Undo/Redo-Funktionen an, so sollen die Benutzerkommandos als atomar erscheinen. Intern können jedoch weitere Checkpoints benötigt werden, die vor der Benutzersicht versteckt bleiben und bei einem Undo übersprungen werden. Dies kann am Beispiel einer PROGRES-An-

wendung verdeutlicht werden. Die Ausführung einer PROGRES-Graphtransformation kann während der Ausführung ihren eigenen internen Checkpoint-Baum benötigen. Nur das Ergebnis des erfolgreichen Pfades dieses Baumes ist aus Benutzersicht interessant. Alle anderen Zweige, deren Scheitern zum Backtracking führte, können verborgen bleiben beziehungsweise sogar gelöscht werden. GRAS3 bietet zwei Möglichkeiten an, mit internen Checkpoints umzugehen. Die erste ist eng an Transaktionen gekoppelt. Alle Checkpoints, die innerhalb von Transaktionen gesetzt werden, werden als intern betrachtet. Nach dem Transaktions-Commit wird nur die lineare Kommandosequenz des erfolgreichen Pfades in das persistente Log des Graphen übernommen. Alle internen Checkpoints werden dabei entfernt. Sind demnach alle Benutzer-Operationen in Transaktionen gehüllt, entstehen keine überflüssigen internen Checkpoints.

Darüber hinaus bietet GRAS3 noch die Möglichkeit an, Checkpoints mit Nummern zu markieren. Mit Hilfe der Operation *GotoCheckpoint* lassen sich so markierte Checkpoint direkt anspringen. Intern laufen dabei die entsprechenden Undo/Redo-Operationen ab.

Wenn ein Checkpoint nicht markiert ist, lässt er sich nur erreichen, wenn der Pfad zu ihm bekannt ist. Es kann jedoch auch sinnvoll sein, unmarkierte Pfade nachzuvollziehen, ohne dass zielgerichtet Checkpoints angesprungen werden. Beim Debugging von PROGRES-Spezifikationen möchte ein Entwickler beispielsweise den kompletten Checkpoint-Baum traversieren, um die Arbeitsweise des Backtracking-Mechanismus zu überprüfen. Dies war einer der Gründe für die Einführung der Funktionen *Backstep* und *Forstep*. Ähnlich einem linearen Undo/Redo-Mechanismus erlauben diese Funktionen Rückwärts- und Vorwärts-Schritte, die eine lineare Liste traversieren. In dieser Liste sind neben allen ausgeführten graphverändernden Kommandos auch Undo- und Redo-Schritte protokolliert. Durch Backstep-Operationen wird somit der Zustand jedes exitierenden Checkpoints rekonstruiert.

4.1.3 Versionsmodell

Das in GRAS3 realisierte Versionsmodell lässt sich anhand der in Abschnitt 3.1 eingeführten Kriterien charakterisieren. Dies liefert folgendes Ergebnis:

Versionierte Objekte Graphen auf oberster Ebene stellen die versionierten Objekte dar, für die separate Kommando-Logs verwaltet werden. Die durch Checkpoints repräsentierten gespeicherten Zustände eines Graphen können als seine Versionen betrachtet werden.

Identifikation von Versionen Die eindeutige Versionsidentifikation wird intern geregelt, Anwendungen können jedoch auch Markierungen in Form von Nummern setzen.

Struktur des Versionsraums Der Versionsraum ist baumartig strukturiert und erlaubt kein Zusammenführen von Versionen.

Beziehungen zwischen Versionsräumen Da geschachtelte Graphen nicht separat versioniert werden, arbeitet die Versionskontrolle nur mit flachen Graphstrukturen. Es existiert demnach keine Enthaltensein-Beziehung zwischen versionierten Graphen, die eine Beziehungen zwischen Versionsräumen verlangen würde. Auf die Problematik, die durch graphübergreifende Kanten entsteht, wird im nächsten Abschnitt eingegangen.

Versionierungsstrategie GRAS3 gibt keine Versionierungsstrategie vor. Das Setzen von Checkpoints wird alleine durch die Anwendung initiiert.

Es ist zu erkennen, dass das Versionsmodell in vielen Punkten dem in Abschnitt 3.4 definierten Modell entspricht. Wichtige Unterschiede treten aber in Bezug auf die Beziehungen zwischen Versionsräumen auf. In Abschnitt 4.3 wird darauf näher eingegangen, wenn zusammenfassend sowohl das GRAS3- als auch das CoObRA-Versionsmodell mit dem DRAGOS-Modell verglichen werden.

4.1.4 Realisierung

Innerhalb eines Graphpools werden alle graphverändernden Kommandos protokolliert, wobei Kommandosequenzen entstehen. Für jeden Graph auf oberster Ebene werden die ihn betreffenden Graphänderungen in einem eigenen Log in Form eines Checkpoint-Baums verwaltet. Jeder Checkpoint in diesem Baum besitzt ein Delta zu seinem Vorgänger-Checkpoint, das aus zwei Kommandosequenzen besteht: Die Sequenz *forward* speichert alle Kommandos, die abgearbeitet werden müssen, um vom Vorgänger zum betrachteten Checkpoint zu gelangen. Nach einem in [Bau99] definierten formalen Modell lässt sich zu jedem protokollierten Kommando eine Folge von Operationen bestimmen, welche die hervorgerufene Änderung rückgängig macht. Zu jeder als *forward* gespeicherten Kommandosequenz existiert demnach eine inverse *backward*-Sequenz. In Abbildung 4.3 sei Checkpoint *CP2* der Nachfolger von *CP1*. Zwischen der Erzeugung der Checkpoints wurden drei Graphveränderungen protokolliert. Diese werden in der *forward*-Sequenz von CP1 gespeichert. Gleichzeitig wird eine

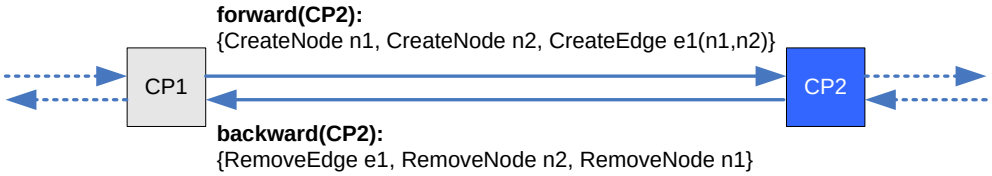

Abbildung 4.3: Beispiel für ein Delta in einer Checkpoint-Sequenz

backward-Sequenz gespeichert, welche die inversen Kommandos in umgekehrter Reihenfolge beinhaltet. Beim Ausführen einer Undo-Operation wird also zunächst die Kante *e1* gelöscht bevor die beiden Knoten entfernt werden.

GRAS3 unterscheidet zwischen flüchtigen und persistenten Logs. Ein flüchtiges Log existiert nur während der Dauer einer Transaktion, beim Commit wird nur der aktuelle Pfad als persistentes Log gespeichert.

Da Checkpoint-Bäume monoton wachsen, wurden für GRAS3 aufwendige Verfahren für die Optimierung von Kommandosequenzen entwickelt. Diese Optimierung geschieht während der Protokollierung, wodurch der nötige Rechenaufwand nicht punktuell auftritt sondern über die Laufzeit verteilt wird. Ziel der eingesetzten Algorithmen ist es, die Länge einer Kommandosequenz so gering wie möglich zu halten, ohne das Ergebnis, das durch ihre Ausführung erzielt wird, zu verändern.

Auch graphübergreifende Kanten sind Gegenstand der Änderungsverwaltung und müssen vom Undo/Redo-Mechanismus erfasst werden. Da graphübergreifende Beziehungen zwischen Untergraphen vom Log des Obergraphen erfasst werden, stellen nur solche Kanten ein Problem dar, die zwischen zwei separat versionierten Graphen bestehen. Würde die Erzeugung einer solchen graphübergreifenden Kante im Log eines der beteiligten Graphen protokolliert, könnte ein späteres Wiederholen der Operation in Folge von Undo/Redo-Schritten fehlschlagen. Der entfernte Graph könnte inzwischen in einen Zustand versetzt worden sein, in dem das durch die Kante referenzierte Element nicht existiert. Auch das gleichzeitige Protokollieren der Operation in den Logs beider Graphen führt zu Inkonsistenzen. GRAS3 bietet aus diesem Grund die Möglichkeit zur Definition sogenannter Log-Gruppen. In ihnen werden Graphen zusammengefasst, zwischen denen graphübergreifende Kanten existieren, deren Konsistenz gewährleistet werden muss. In Kapitel 5 werden Log-Gruppen näher erläutert, da das Problem graphübergreifender Beziehungen auch in DRAGOS besteht.

4.2 CoObRA

Die Anforderungen an das *Concurrent Object Replication Framework* (CoObRA) [Sch03] sind im Rahmen des FUJABA-Projektes [KNNZ99] entstanden. FUJABA (From UML to Java And Back Again) ist ein CASE-Tool, das die objektorientierte Modellierung von Softwaresystemen ermöglicht. Die Modellierung geschieht mit Hilfe einer erweiterten Form der *Unified Modelling Language* (UML). Klassendiagramme sowie das Verhalten von Objekten können dabei weitgehend graphisch spezifiziert werden. Aus den Spezifikationen kann lauffähiger Java-Code erzeugt werden.

Das CoObRA-Projekt ergänzt FUJABA um Funktionen zur Versionsverwaltung, die die Arbeit mehrerer Benutzer an einer Spezifikation möglich machen. Dabei ist der Einsatz von CoObRA nicht auf FUJABA beschränkt. Es stellt ein Rahmenwerk zur Verfügung, das es möglich macht, komplexe Objektstrukturen um Eigenschaften zu ergänzen, die für viele Anwendungen im Bereich der Softwareentwicklung von Bedeutung sind:

- Persistente Speicherung
- Undo-/Redo-Mechanismen
- Mehrbenutzerfähigkeit

Um diese Anforderungen zu erfüllen, verfolgt CoObRA ein ähnliches Konzept wie die Änderungsverwaltung in GRAS3. Das System protokolliert alle Änderungen an der Datenbasis und speichert sie persistent. Die verwalteten Objektstrukturen stellen Graphen dar, in denen die Objekte durch Knoten repräsentiert sind. Beinhaltet ein Feld eines Objektes eine Referenz auf ein anderes Objekt, entsteht eine gerichtete Verbindung. Felder können auch Kollektionen von Werten enthalten. Um den Zugriff auf solche Kollektionen zu erleichtern, unterscheidet CoObRA zwischen einfach besetzten Feldern und Feldern, die Mengen von Elementen speichern. Letztere werden in [Sch03] *Mengen-Felder* genannt.

Zur Veranschaulichung ist im linken Teil von Abbildung 4.4 beispielhaft die Definition von zwei Klassen angegeben, die gemeinsam eine verkettete Liste implementieren. Objekte der Klasse *List* speichern in den Feldern *first* und *last* Referenzen auf Objekte der Klasse *ListItem*, die das erste beziehungsweise letzte Element der Liste darstellen. Jedes *ListItem*-Objekt referenziert über das Feld *next* das jeweils nächste Element der Liste. Die Objekte, die am Aufbau der Liste beteiligt sind, bilden einen Graphen, in dem gerichtete Kanten Referenzen repräsentieren. Der Objektgraph zu einer drei-elementigen Liste ist im rechten Teil der Abbildung dargestellt.

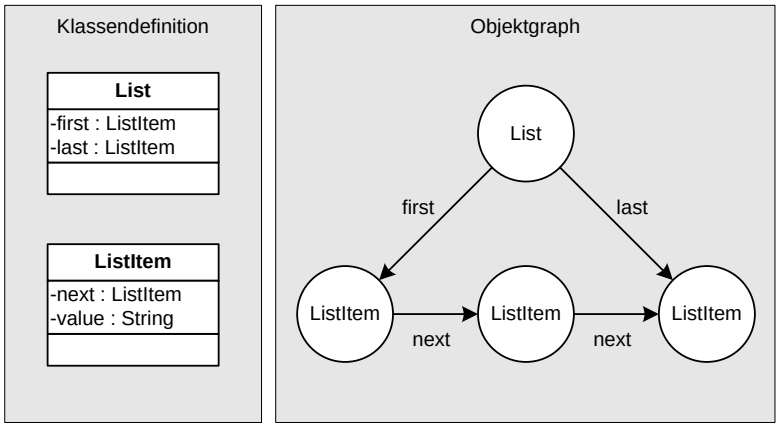

Abbildung 4.4: Beispiel für eine Objektstruktur

CoObRA unterscheidet fünf grundlegende Arten von Änderungen an der Datenbasis:

1. Erzeugen eines Objektes
2. Entfernen eines Objektes
3. Setzen eines Feldes
4. Hinzufügen eines Elementes in ein Mengen-Feld
5. Entfernen eines Elementes aus einem Mengen-Feld

Aufbauend auf der Protokollierung dieser Änderungen werden Mechanismen realisiert, welche die drei zuvor erwähnten Anforderungen erfüllen. Änderungsprotokolle stellen lineare Listen dar, deren persistente Speicherung problemlos realisierbar ist. Werden die in Listen gespeicherten Änderungen in der Reihenfolge des Protokollierens abgearbeitet, kann die Datenstruktur an beliebiger Stelle rekonstruiert werden. Bei der Arbeit mit den Daten kann das Änderungsprotokoll auch benutzt werden, um einzelne Änderungen rückgängig zu machen. Die Konzepte zur Realisierung der kurzfristigen Änderungskontrolle sind demnach denen ähnlich, die in GRAS3 umgesetzt wurden.

CoObRA beinhaltet zudem die Funktionalität, um verteiltes Arbeiten zu ermöglichen. Dazu werden alle Änderungslisten der Datenbasis in einem zentralen Server-Repository verwaltet. Ein Entwickler verwendet ein lokales Repository, welches die Änderungslisten per *Update*-Operation beziehen kann, um die von ihnen repräsentierte Datenstruktur lokal zu replizieren. Alle Änderungen, die der Benutzer tätigt, verändern zunächst die lokal replizierten Objekte. Erst

beim Ausführen einer *Check-In*-Operation werden seine lokalen Änderungsprotokolle an das zentrale Repository übermittelt und stehen so allen Entwicklern zur Verfügung.

Der Check-In-Vorgang läuft in mehreren Schritten ab. Zunächst werden alle Änderungen, die bislang nur lokal erfolgt sind, rückgängig gemacht. Dann stellt eine *Update*-Operation sicher, dass der Zustand der lokalen Daten dem auf dem Server gespeicherten Zustand entspricht. Dabei werden alle eventuell vorhandene Änderungen, die zwischenzeitlich von anderen Benutzern im Server-Repository gespeichert wurden, auf die lokalen Daten angewandt. Anschließend werden die zuvor rückgängig gemachten lokalen Änderungen wiederholt. Erst bei diesem Schritt können Konflikte auftreten, die lokal beseitigt werden müssen, bevor das Kopieren der Änderungen in das Server-Repository erfolgt. Durch diese Vorgehensweise wird gewährleistet, dass die übertragenen Änderungen keine Konflikte innerhalb der Server-Datenbasis erzeugen, wodurch deren Konsistenz gewahrt wird. Die lokale Konfliktbehandlung ist in vielen Fällen nicht automatisch zu bewerkstelligen, da sie vom anwendungsspezifischen Datenmodell abhängt. Kann ein Konflikt nicht von der Anwendung gelöst werden, bietet CoObRA auch die Möglichkeit zur interaktiven Konfliktbehandlung an. Hierbei werdem dem Benutzer die lokalen Änderungen, die nach dem Update-Vorgang nicht mehr durchführbar sind, angezeigt und Möglichkeiten zur Lösung angeboten. Wurde beispielsweise ein zu änderndes Objekt inzwischen durch eine von einem anderen Benutzer getätigte Änderung gelöscht, so könnte die Lösung dieses Konfliktes darin bestehen, das Objekt neu zu erzeugen, und anschließend die Änderung durchzuführen. Darüber hinaus besteht immer die Möglichkeit, die lokale Änderung zu verwerfen.

Zu jeder protokollierten Änderung speichert CoObRA zusätzlich den Grund der Änderung. Dieser Grund kann zum Beispiel eine andere Änderung sein, wodurch Abhängigkeiten zwischen Änderungen definiert werden können. Im Falle des Erzeugens einer beidseitigen Verbindung wird beispielsweise für beide beteiligten Objekte jeweils eine Änderung gespeichert, die das Setzen einer Feldwertes beinhaltet. Indem im Änderungsgrund eine Abhängigkeit definiert wird, kann das alleinige Ausführen einer der Änderungen verhindert werden. Als Änderungsgrund können außerdem Zusatzinformationen gespeichert werden. Beim gemeinsamen Check-In mehrerer Änderungen werden diese beispielsweise alle mit dem gleichen Grund markiert, um die Änderungshistorie zu strukturieren.

4.2.1 Versionsmodell

Die im vorherigen Abschnitt beschriebenen Eigenschaften ergeben nach den in Abschnitt 3.1 eingeführten Merkmalen folgendes Versionsmodell:

Versionierten Objekte In CoObRA lassen sich nur von einem gesamten lokalen Repository Versionen erzeugen. Eine Versionierung auf feinerer Ebene ist nicht möglich.

Identifikation von Versionen In der kurzfristigen lokalen Änderungsverwaltung kann die Anwendung einzelne Änderungen durch die Definition von Änderungsgründen markieren. Wird der gleiche Grund für eine Folge von Änderungen verwendet, können sie auf diese Weise gruppiert werden. Eine gewünschte Version lässt sich darüber aber nur indirekt wiederherstellen. Die Identifikation von langfristigen Versionen im Server-Repository geschieht ebenfalls indirekt über die Unterscheidung der Check-In-Vorgänge, die durch eindeutige Änderungsgründe repräsentiert sind.

Struktur des Versionsraumes Die Zustände im Versionsraum sind linear voneinander abgeleitet, Verzweigungen sind nicht möglich.

Beziehungen zwischen Versionsräumen Da nur ein Versionsraum für ein Repository existiert, müssen Beziehungen zwischen Versionsräumen nicht beachtet werden.

Versionierungsstrategie Die Versionierungsstrategie ist der Anwendung überlassen.

Es existieren bedeutende Unterschieden zu dem in Abschnitt 3.4 definierten Versionsmodell für DRAGOS. Abschnitt 4.3 wird darauf genauer eingehen, wenn zusammenfassend alle betrachteten Versionsmodelle verglichen werden.

4.2.2 Realisierung

Prinzipiell kann CoObRA in jede Anwendung integriert werden, die Anwendungsdaten in Form von Objektstrukturen erzeugt. Dazu werden die Klassen, deren Instanzen Gegenstand der Änderungsverwaltung sein sollen, durch CoObRA-Code ergänzt. Die Implementierung dieser Klassen muss dafür einige Voraussetzungen erfüllen. Alle Informationen, die persistent gespeichert werden sollen, müssen in Feldern der Objekte abgelegt sein. Der Zugriff auf die Felder muss dabei stets über spezialisierte Zugriffsmethoden erfolgen, wie dies

auch in der JavaBeans Spezifikation zu Properties [Ham97] vorgesehen ist. Die Namen der Zugriffsmethoden setzen sich stets zusammen aus dem Präfix *get* beziehungsweise *set* gefolgt vom Namen des Feldes und werden deshalb auch als Getter- und Setter-Methoden bezeichnet. Auf Felder, die Mengen von Werten speichern, erfolgt der Zugriff über Methoden, die ein neues Element zur gespeicherten Menge hinzufügen beziehungsweise aus ihr entfernen. Zur Bezeichnung dieser Methoden wird das Präfix *addTo* beziehungsweise *removeFrom* verwendet.

Eine weitere Voraussetzung für die Integration von CoObRA besagt, dass zwischen zwei Modifikationen durch Setter-Methoden der Wert eines Feldes nicht verändert werden darf. Dies beinhaltet auch, dass ein über ein Feld referenziertes Objekt nicht geändert werden darf, ohne dabei das Feld neu zu besetzen. Sind diese Anforderungen erfüllt, wird CoObRA integriert, indem alle Setter-Methoden um Code ergänzt werden, der Änderungen an dem Feld protokolliert. Das Protokollieren der Erzeugung und des Löschens von Objekten wird vom lokalen Repository übernommen. Deshalb sollte der Konstruktor einer Klasse die Registrierung des Objekts im Repository sicherstellen. Die Abmeldung aus dem Repository ist beim Löschen des Objektes zu vollziehen.

Im Repository wird neben einer Liste der registrierten Objekte auch die Liste protokollierter Änderungen verwaltet. Die Methode `undo` bewirkt einen RückwärtsSSchritt, die Methode `redo` einen Vorwärts-Schritt innerhalb dieser Liste. Jede Änderung muss demnach in beide Richtungen ausführbar sein, je nachdem ob sie wiederholt oder rückgängig gemacht wird. Änderungen sind als Instanzen der Klasse `ObjectChange` realisiert, die Informationen über die Art der Änderung und zusätzlich notwendige Parameter enthalten. Änderungen, die das Erzeugen beziehungsweise Löschen von Objekten betreffen, benötigen als Parameter nur die Referenz auf das Objekt. Undo- und Redo-Operationen bewirken, dass dieses Objekt im Repository angemeldet beziehungsweise abgemeldet wird.

Alle anderen Änderungen beziehen sich auf Werte von Felder und benötigen mehrere Parameter. Dabei sind die unterschiedliche Arten von Feldern zu unterscheiden:

1. **Felder mit einzelnen Werten**: Über den in der Änderung gespeicherten Namen des Feldes in Kombination mit dem Präfix `set` wird die zu verwendende Setter-Methode bestimmt und aufgerufen. Der Aufruf geschieht über das Reflection API von Java [SUN], das den Zugriff auf Klassen- und Objekt-Informationen zur Laufzeit erlaubt. In der Änderung sind sowohl der alte als auch der neue Wert des Attributs gespeichert, die je nach Ausführungsrichtung der Änderung gesetzt werden.

2. **Felder mit Mengen von Werten**: Hinzufügen und Entfernen sind als eigene Änderungstypen realisiert, wobei jeweils die Vorwärts-Richtung des einen der Rückwärts-Richtung des anderen entspricht. In der Änderung muss zusätzlich zum Namen des Feldes nur ein Wert gespeichert sein. Je nach Ausführungsrichtung der Änderung wird dieser zu der im Feld gespeicherten Menge hinzugefügt oder aus ihr entfernt. Dazu wird mit dem Feldnamen und dem Präfix `addTo` oder `removeFrom` die aufzurufende Methode identifiziert. Der Aufruf geschieht auch in diesem Fall über das Reflection API.

An dieser Stelle wird auch der Sinn der speziellen Behandlung von Feldern, die Mengen von Elementen enthalten, deutlich: Oftmals ersetzen Änderungen an solchen Feldern nicht die gespeicherte Kollektion durch eine neue, sondern es werden nur einzelne Elemente aus der Kollektion entfernt oder neue hinzugefügt. Durch Verwendung der Methoden `addToX` und `removeFromX` wird in der protokollierten Änderung nur das jeweils einzige hinzuzufügende beziehungsweise zu löschende Element gespeichert. Eine Modifizierung des Feldes mit Hilfe von Methoden der Form `getX` und `setX` hätte zur Folge, dass sowohl der alte Wert als auch der neue Wert, der sich nur geringfügig vom alten unterscheidet, in der Änderung gespeichert würden. Bei großen Kollektionen ist dies sehr ineffizient.

Der Aufruf der Methode `compact` optimiert einen Teil der Änderungsliste, indem Redundanzen entfernt werden. Beispielsweise wird dadurch die mehrfache Änderung eines Feldes zu einer einzigen Änderung zusammengefasst. Die so optimierten Änderungen werden gleichzeitig zu einem Block zusammengefasst und sind nicht mehr einzeln per Undo rückgängig zu machen. Der Aufruf von `compact` ist deshalb besonders vor dem Speichern der Daten beziehungsweise vor Check-In-Vorgängen ratsam.

Das zentrale CoObRA-Repository kann auf einem entfernten Server liegen. Die Kommunikation zwischen dem Server und den lokalen Repositories verläuft mittels CORBA [Vin97]. Die wichtigsten Methoden, die der Server anbietet, sind dabei `update` und `checkin`, deren Funktion bereits erläutert wurde. Sie initieren den Austausch von Änderungslisten mit einem lokalen Repository. Die Listen von ObjektChange-Instanzen werden dabei in einem XML-Format übertragen.

4.2.3 Integration in FUJABA

Die Integration von CoObRA in die Entwicklungsumgebung FUJABA wurde auf zwei Ebenen vollzogen:

1. Funktionen der Entwicklungsumgebung wurden mittels CoObRA realisiert.

2. Die Codegenerierung wurde erweitert, um die Integration von CoObRA-Funktionalität in die von FUJABA generierten Werkzeugen zu ermöglichen.

Zu den mittels CoObRA realisierten Funktionen der Entwicklungsumgebung zählt zum einen ein Undo/Redo-Mechanismus, der das Rückgängigmachen und Wiederherstellen einzelner Benutzeraktionen erlaubt. Zum anderen werden Funktionen angeboten, die ähnlich einem Versionsverwaltungssystem wie CVS die Arbeit mehrerer Entwicklern an der gleichen FUJABA-Spezifikation ermöglichen. Dazu wird ein CoObRA-Server eingerichtet, auf den die einzelnen FUJABA-Instanzen als Clients zugreifen können. Der Server realisiert eine Benutzerauthentifizierung und verwaltet das zentrale Repository. Die Funktionen *Update* und *Check-In* initiieren den Datenaustausch mit dem client-seitigen lokalen Repositories. Treten Konflikte auf, können diese per Benutzerinteraktion beseitig werden.

Durch die Integration von CoObRA wird auch die Funktionalität der Codegenerierung erweitert. Dem FUJABA-Anwender wird dadurch ermöglicht, CoObRA-Funktionalität in eine spezifizierte Anwendung zu integrieren. Um dies zu erreichen werden alle innerhalb einer Spezifikation angelegten Klassen, die persistente Anwendungsdaten speichern, als *persistent* markiert. Die Codegenerierung erzeugt automatisch ein lokales Repository, in dem die Instanzen so markierter Klassen verwaltet werden. Der generierte Code der Klassen selbst wird erweitert, um das erforderliche Protokollieren der Änderungen zu gewährleisten. Über das Repository steht dann der erzeugten Anwendung die CoObRA-Funktionalität zur Verfügung. In [Sch03] wird dieser Vorgang am Beispiel der Spezifikation eines einfachen Klassendiagramm-Editors demonstriert. Dieser kann durch CoObRA mit geringem Aufwand Undo/Redo-Funktionen auf den von ihm erzeugten Daten realisieren.

	GRAS3	CoObRA	DRAGOS-Anforderungen
Versionierte Objekte	Graphen auf oberster Ebene	Gesamtes Repository	Einzelne Graphen
Versions-identifikation	Nummern	Indirekt	Beliebiger Bezeichner
Struktur des Versionsraumes	Baumartig	Linear	Baumartig
Beziehungen zwischen Versionsräumen	Graphübergreifende Kanten	Keine	Graphübergreifende Kanten und Relationen, hierarchische Graphen
Versionierungs-strategie	Offen	Offen	Offen

Tabelle 4.1: Vergleich der Versionsmodelle

4.3 Zusammenfassung

Tabelle 4.1 stellt die Versionsmodelle von GRAS3 und CoObRA den in 3.4 identifizierten Anforderungen gegenüber, um ihre Anwendbarkeit in Bezug auf DRAGOS zu prüfen. Anhand der vorhandenen Unterschiede lässt sich erkennen, dass die in den Vergleichssystemen realisierten Konzepte nicht für DRAGOS übernommen werden können.

Das Versionsmodell von GRAS3 entspricht zwar den Anforderungen in vielen Punkten, jedoch finden sich bedeutende Unterschiede in den zu berücksichtigenden Beziehungen zwischen Versionsräumen. Die Änderungsverwaltung in GRAS3 arbeitet mit flachen Graphenstrukturen und muss nicht wie die DRAGOS-Versionskontrolle separate Versionsräume hierarchisch geschachtelter Graphen berücksichtigen. Der grundsätzliche Mechanismus der Protokollierung von Graphveränderungen ist dennoch auch in DRAGOS anwendbar. Er wird um Konzepte ergänzt werden, die geeignet sind, die erweiterten Anforderungen zu erfüllen.

Die in CoObRA realisierten Versionierungsfunktionen erfüllen die in Abschnitt 3 formulierten Anforderungen an die Versionskontrolle für DRAGOS nur teilweise. Sie sollen für Anwendungen lineare Undo/Redo-Funktionen bereitstellen und die Arbeit mehrerer Benutzer auf einer gemeinsamen Datenba-

sis ermöglichen. Darüber hinaus wird ein Persistenz-Mechanismus realisiert, der auf den Änderungsdaten aufbaut, die die Versionskontrolle liefert. Für DRAGOS wird hingegen ein Mechanismus benötigt, der sich in ein vorhandenes Datenbanksystem integriert und an beliebige Anwendungsfunktionalität angepasst werden kann.

Bedingt durch diese verschiedenen Ansprüche, die CoObRA erfüllt, unterscheidet sich das realisierte Versionsmodell von den für DRAGOS identifizierten Anforderungen vor allem in Bezug auf die versionierten Objekte. Dadurch dass pro Repository nur ein einziger Versionsraum existiert, ergeben sich neben erheblichen Vereinfachungen für das realisierte Modell auch Einschränkungen, die den DRAGOS-Anforderungen nicht gerecht werden. Dies gilt auch für die lineare Struktur des Versionsraumes, die keine parallelen Entwicklungszweige zulässt. Da alle vergangenen Zustände über das lineare Undo/Redo erreichbar sind, ist Versionsidentifikation über eindeutige Bezeichner nicht vorgesehen. In Bezug auf DRAGOS sind daher vor allem die über das Versionsmodell hinausgehenden Konzepte von CoObRA interessant, wie beispielsweise die Replikation von Änderungen und die Mechanismen zur Verteilung der Daten über mehrere Repositories.

Der Vergleich zeigt, dass sowohl in GRAS3 als auch in CoObRA Konzepte zu finden sind, die Teilaspekte der Anforderungen von DRAGOS erfüllen. Einige dieser Konzepte finden sich in angepasster Form in der Realisierung der Versionierungserweiterung wieder, mit der sich das nächste Kapitel beschäftigt.

5 Realisierung

Dieses Kapitel beschäftigt sich mit der im Rahmen dieser Arbeit entwickelten Realisierung einer Versionierungserweiterung für DRAGOS. Das Kapitel ist in drei Abschnitte unterteilt. Abschnitt 5.1 stellt die Konzepte vor, die als geeignet erkannt wurden, um die in Abschnitt 3 formulierten Anforderungen zu erfüllen. Abschnitt 5.2 erläutert, wie diese Konzepte in der Implementierung umgesetzt wurden. Am Ende des Kapitels gibt Abschnitt 5.3 einen Überblick über den aktuellen Entwicklungsstand der Versionskontrolle.

5.1 Konzepte

In den folgenden Abschnitten werden die Konzepte vorgestellt, die zur Realisierung der Versionskontrolle geeignet sind. Abschnitt 5.1.1 definiert dabei zunächst den versionierten Zustand und den Aufbau des Versionsraumes eines Graphen, bevor in Abschnitt 5.1.2 mit dem Logging-Konzept der zentrale Mechanismus zum Erzeugen und Wiederherstellen von Versionen vorgestellt wird. In den nachfolgenden Abschnitten 5.1.3 und 5.1.4 werden zwei zusätzliche Konzepte zur Behandlung graphübergreifender Verbindungen und Konfigurationen erläutert. Danach wird in Abschnitt 5.1.5 erklärt, wie mit Hilfe des Logging-Mechanismus die Replikation von Graphen bewerkstelligt werden kann. Darauf aufbauend werden im letzten Abschnitt erste Ansätze zum Zusammenführen von Versionen entwickelt.

5.1.1 Versionierung

Der Begriff *Version* ist definiert als ein wiederherstellbarer Zustand eines versionierten Objektes. In Bezug auf DRAGOS ist es daher zunächst wichtig, den versionierten Zustand eines Graphen zu definieren.

Versionierter Zustand

Der *versionierter Zustand eines Graphen* umfasst die Merkmale, die von der Versionierung des Graphen betroffen sind. Er setzt sich zusammen aus folgenden Punkten:

1. dem lokalen Zustand des Graphen selbst,
2. der Komposition der enthaltenen Graphelemente,
3. dem lokalen Zustand aller enthaltenen Graphelemente, die keine Graphen sind,
4. dem Verbindungszustand aller enthaltenen Kanten und Relationsenden, sowie
5. dem versionierten Zustand aller enthaltenen Graphen.

Der letzte Punkt gibt der Definition einen rekursiven Charakter. Die folgenden drei Definitionen erklären die anderen aufgeführten Merkmale:

Lokaler Zustand eines Graphelements Dieser sei definiert als die Wertbelegung aller Attribute und Meta-Attribute des Graphelements. Hierbei ist auch die Möglichkeit unbewerteter (Meta-)Attribute zu berücksichtigen.

Komposition der enthaltenen Graphelemente Dies ist die Menge aller zu einem bestimmten Zeitpunkt in einem Graphen enthaltenen Graphelemente. Aufrufe von Methoden, die Graphelemente erzeugen beziehungsweise entfernen, verändern diese Menge.

Verbindungszustand Dieser beinhaltet für Relationsenden die Information über das referenzierte Graphelement. Für Kanten sind Informationen über die durch Quelle und Ziel referenzierten Graphelemente erforderlich. Bei graphübergreifenden Kanten und Relationsenden besteht die Möglichkeit, dass referenzierte Graphelemente nicht Teil des betrachteten versionierten Zustands sind.

Neben den aufgezählten Merkmalen kann der Zustand eines Graphen auch Merkmale enthalten, die von der Versionskontrolle nicht berücksichtigt werden. Solche unversionierten Anteile werden jedoch ausschließlich für die interne Realisierung verwendet. Auf sie wird deshalb in Kapitel 5.2 näher eingegangen. Alle aus Anwendersicht sichtbaren Merkmale sollen automatisch Gegenstand der Versionskontrolle sein, um die Komplexität der Anwenderschnittstelle minimal zu halten.

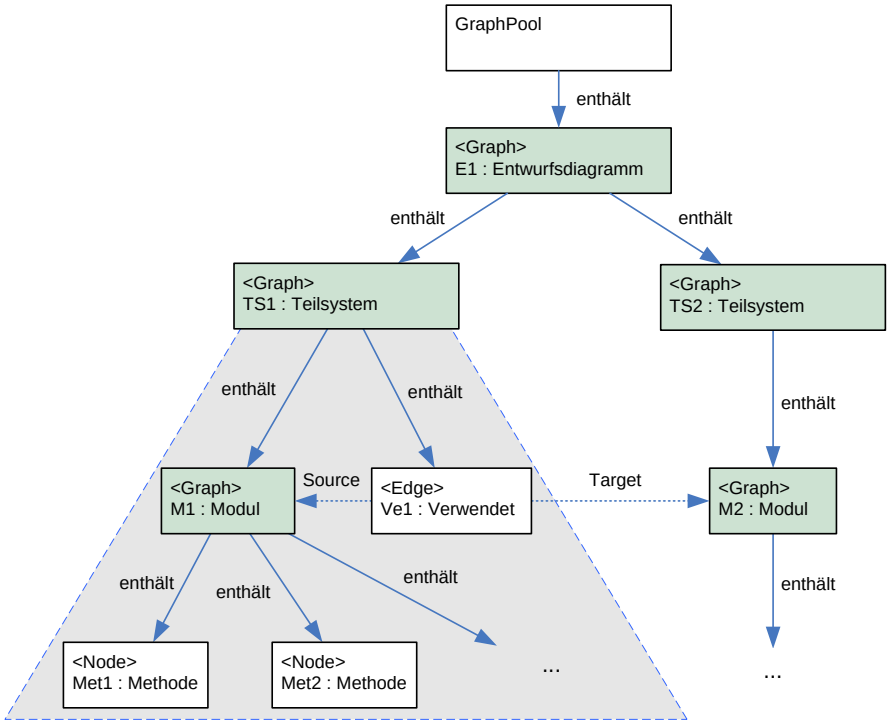

Abbildung 5.1: Zustand eines Graphen

Abbildung 5.1 verdeutlicht den versionierten Zustand eines Graphen anhand eines Beispiels. In dem dargestellten Entwurfsdiagramm soll der Graph `TS1` versioniert werden. Die grau hinterlegte Fläche markiert den Bereich, der zum versionierten Zustand des Graphen gehört. Er umfasst unter anderem den versionierten Zustand des Untergraphen `M1` inklusive aller in `M1` enthaltenen Graphelemente. Da die Kante `Ve1` in `TS1` enthalten ist, gehört sowohl ihr lokaler Zustand als auch ihr Verbindungszustand zum betrachteten versionierten Zustand, während das referenzierte Graphelement `M2` nicht eingeschlossen ist. Der Abschnitt 5.1.3 beschäftigt sich eingehend mit dieser Problematik.

Versionsraum

Durch Nutzung der Versionskontrolle wird jedem Graphen ein eigener Versionsraum zugeordnet, in dem die gespeicherten versionierten Zustände des Graphen verwaltet werden. Zwei Versionen nehmen dabei eine besondere Rolle ein: Die *Ursprungsversion* und die *aktuelle Version*. Die aktuelle Version markiert den Ausgangspunkt für alle zukünftigen Graphveränderungen. War die letzte Versionierungsoperation eine Versionserzeugung, so ist die erzeugte Version

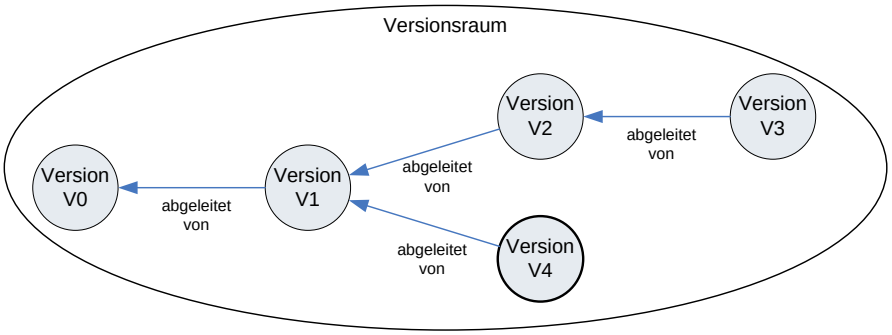

Abbildung 5.2: Versionsraum eines Graphen

die neue aktuelle Version. War die letzte Versionierungsoperation hingegen eine Versionswiederherstellung, nimmt die wiederhergestellte Version die Rolle der aktuellen Version ein.

Unmittelbar nach der Erzeugung eines neuen Graphen existiert genau eine Version im Versionsraum des Graphen, die auch als *Ursprungsversion* bezeichnet wird. Solange keine weitere Version erzeugt wird, ist sie gleichzeitig die aktuelle Version. Der von ihr repräsentierte Zustand ist stets ein leerer Graph, dessen Attribute und Meta-Attribute ungesetzt sind. Beim Erzeugen einer Version eines Graphen wird der versionierte Zustand des Graphen, wie er zum aktuellen Zeitpunkt im Graphpool existiert, gespeichert. Im Versionsraum wird eine neue Version angelegt, die den gespeicherten Zustand repräsentiert, also die Rolle der aktuellen Version annimmt. Die vorherige aktuelle Version wird über eine *abgeleitet von*-Beziehung als Vorgängerversion referenziert.

Der Versionsgraph kann verzweigen, wenn die aktuelle Version bereits eine Nachfolger-Version besitzt. Die nächste erzeugte Version stellt dann einen zweiten Nachfolger dieser Version dar. Abbildung 5.2 verdeutlicht diesen Fall. Hier wurden zunächst ausgehend von der Ursprungsversion `V0` linear die Versionen `V1` bis `V3` erzeugt. Nachdem Version `V1` wiederhergestellt worden ist, stellt die anschließend erzeugte Version `V4` einen zweiten Nachfolger von `V1` dar.

Später zu implementierende Undo/Redo-Funktionen stellen die Anforderung, zu jedem Zeitpunkt einen *aktuellen Pfad* identifizieren zu können, der von der Ursprungsversion durch die aktuelle Version zu einem Blatt des Versionsgraphen führt. Deshalb ist eine Ordnung auf den Nachfolgern einer Version definiert, die ihnen eine Zahl nach der Reihenfolge ihrer Erzeugung zuordnet. Ist die aktuelle Version selbst kein Blatt des Versionsgraphen, kann über die Nachfolger-Ordnung der Pfad zu einem eindeutigen Blatt festgelegt werden.

5.1.2 Logging

Für die Umsetzung der Versionierungsmechanismen in DRAGOS wurde das in GRAS3 realisierte und in Kapitel 4.1.2 vorgestellte Konzept aufgegriffen und an die speziellen Eigenschaften von DRAGOS angepasst. Es besteht im Wesentlichen aus der Protokollierung aller graphverändernden Operationen. Aus der Anforderung, dass zu jedem Zeitpunkt ein eindeutiger aktueller Zustand des Graphpools existieren muss, ergibt sich die gleiche Forderung auch für jeden einzelnen Graphen. Dies beinhaltet, dass jeder historische, als Version gespeicherte Zustand eines Graphen bei Bedarf konstruiert wird und anschließend den aktuellen Zustand des Graphen darstellt. Das *Logging*-Konzept ist ein Verfahren, um die Informationen, die für die Rekonstruktion benötigt werden, zu sammeln und in Form von Versionsdeltas zu speichern.

Um das Logging-Konzept zu erläutern werden im nächsten Abschnitt zunächst nur flache Graphen, also Graphen, die keine Untergraphen beinhalten, betrachtet. Anschließend wird erklärt, wie das Konzept auf geschachtelte Graphen erweitert wird.

Flache Graphen

Wie in Kapitel 3.1 dargestellt, beinhaltet ein Delta den Unterschied zwischen zwei aufeinanderfolgenden Versionen eines Graphen. Dieser Unterschied ist die Konsequenz aus einer Abfolge von graphverändernden Operationen, die von der Anwendung initiiert wurden. Werden zunächst ausschließlich flache Graphen betrachtet, reichen folgende sieben Basis-Operationen, um die Veränderung am versionierten Zustand eines Graphen protokollieren zu können:

1. Erzeugen eines neuen Graphelements
2. Löschen eines Graphelements
3. Setzen eines Attributs
4. Ungültigsetzen eines Attributs
5. Setzen eines Meta-Attributs
6. Ungültigsetzen eines Meta-Attributs
7. Ändern der Verbindungsreferenzen einer Kante oder eines Relationsendes

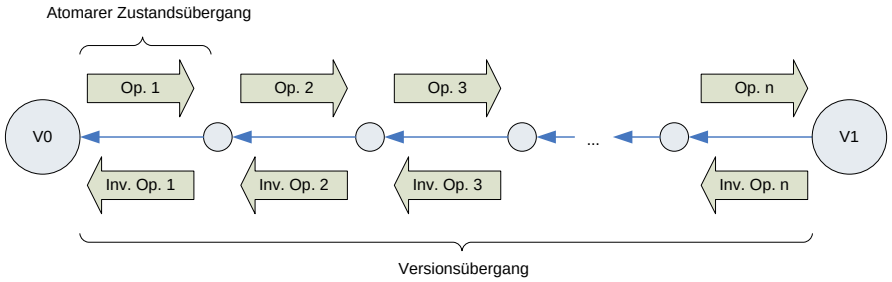

Abbildung 5.3: Protokollierte Operationen

Wie in Abbildung 5.3 dargestellt lässt sich der Zustandsübergang zwischen zwei Versionen auch als eine Folge atomarer Zustandsübergänge auffassen. Dabei ist jeder atomare Zustandsübergang Konsequenz der Ausführung einer einzelnen Operation. Zu jeder Operation lässt sich eine eindeutige *inverse Operation* bestimmen, die den atomaren Zustandsübergang rückgängig macht. Beispielsweise wird das Erzeugen eines neuen Graphelements rückgängig gemacht, indem das betreffende Graphelement gelöscht wird.

Indem für jeden atomaren Zustandsübergang sowohl die getätigte Operation als auch ihre inverse Operation protokolliert wird, lässt sich der Übergang in beide Richtungen vollziehen. Im folgenden wird deshalb die protokollierte getätigte Operation als *Vorwärts-Kommando* und ihre inverse Operation als *Rückwärts-Kommando* bezeichnet. Durch die Ausführung des letzten protokollierten Rückwärts-Kommandos wird ein atomarer Undo-Schritt vollzogen.

Tabelle 5.1 ordnet jeder graphverändernden Operation ihre inverse Operation zu. Wie man erkennen kann, ist beim Setzen von Attributwerten die Auswahl der inversen Operation abhängig von der vorherigen Attributbelegung: War das Attribut zuvor mit einem Wert belegt, so muss dieser durch die inverse Operation gesetzt werden. War das Attribut hingegen ungesetzt muss auch dieser Zustand wiederhergestellt werden. Der gleiche Zusammenhang besteht zwischen Operationen zur Veränderung von Meta-Attributen und ihren inversen Operationen.

Beim Entfernen eines Graphelements tritt eine besondere Problematik auf, die in den in der Tabelle aufgeführten Vorbedingungen deutlich wird. So kann laut der ersten Vorbedingung eine inverse Operation nur dann bestimmt werden, wenn das zu löschende Graphelement keine inzidenten Kanten und Relationsenden besitzt. Andernfalls würden diese durch die Operation ebenfalls gelöscht werden. In dem in Abbildung 5.1 dargestellten Szenario würde also beim Löschen von Graph `M1` auch die Kante `V1` entfernt werden. Eine inverse Operation müsste demnach zwei Graphelemente wiederherstellen, um die Konsistenz des

Operation	**Vorbedingung**	**Inverse Operation**
Erzeuge Graphelement G		Entferne Graphelement G
Entferne Graphelement G	1. Keine Kante und kein Relationsende ist inzident zu G 2. G befindet sich in Zustand Z	Erzeuge Graphelement G und stelle Zustand Z her
Setze Attribut A auf Wert W_{neu}	Attribut A ist ungültig	Setze Attribut A ungültig
	Attribut A hat Wert W_{alt}	Setze Attribut A auf Wert W_{alt}
Setze Attribut A ungültig	Attribut A hat Wert W_{alt}	Setze Attribut A auf Wert W_{alt}
Setze Meta-Attribut M auf Wert W_{neu}	Meta-Attribut M ist ungesetzt	Setze Meta-Attribut M auf ungesetzt
	Meta-Attribut M hat Wert W_{alt}	Setze Meta-Attribut M auf Wert W_{alt}
Setze Meta-Attribut M auf ungesetzt	Meta-Attribut M hat Wert W_{alt}	Setze Meta-Attribut M auf Wert W_{alt}
Setze Verbindungsreferenz von Kante/Relationsende K auf Graphelement G_{neu}	Kante/Relationsende hat Verbindungsreferenz auf Graphelement G_{alt}	Setze Verbindungsreferenz von Kante/Relationsende K auf Graphelement G_{alt}

Tabelle 5.1: Graphverändernde Operationen und inverse Operationen

versionierten Zustands des Vater-Graphen nicht zu verletzen. Die Lösung besteht darin, dass nur Lösch-Operationen bezüglich Graphelementen protokolliert werden, die keine inzidenten Kanten und Relationsenden besitzen. Wird diese Vorbedingung nicht erfüllt, kann die angestrebte komplexe Lösch-Operation in eine Folge einfacher Lösch-Operationen zerlegt werden, die die Vorbedingung erfüllen. Die Zerlegung leistet folgender Algorithmus:

```
löscheKomplex(G): {

  Für jedes zu G inzidente Graphelement I: {
    löscheKomplex(I);
  }
  lösche G und protokolliere atomare Operation;
}
```

Im Beispiel würde also die komplexe Lösch-Operation von `M1` in die Folge {Lösche `V1`, Lösche `M1`} zerlegt werden. Zu jeder dieser atomaren Operationen lässt sich eine inverse Operation bestimmen.

Die zweite in der Tabelle aufgeführte Vorbedingung bezieht sich auf den Zustand des zu löschenden Graphelements. Die inverse Operation muss beim Rekonstruieren des Graphelements den Zustand wiederhergestellen, in dem sich das Element vor dem Entfernen befand. Dabei müssen auch eventuell unversionierte Merkmale berücksichtigt werden. Für einen Knoten muss beispielsweise die Wertbelegung aller Attribute und Meta-Attribute rekonstruiert werden. Die Problematik lässt sich aber durch die für DRAGOS gewählte Implementierungsart umgehen, die in Abschnitt 5.2.2 erläutert wird.

Werden anstatt atomarer Zustandsübergänge nun Versionsübergänge betrachtet, bedeutet ein Rückschritt von der aktuellen Version zur Vorgänger-Version die Ausführung der Liste von Rückwärts-Kommandos, die seit der Versionserzeugung protokolliert wurde. Der Übergang von der aktuellen Version zu einer Nachfolger-Version vollzieht sich durch die Abarbeitung der Liste der Vorwärts-Kommandos, die entlang des Versionsübergangs protokolliert wurden. Wenn ein Versionsgraph baumartig verzweigt, ist der protokollierte Kommandostrom ebenfalls baumartig. Es ist daher zweckmässig, jeder Version eine Vorwärts- und Rückwärts-Kommandoliste zuzuordnen, die den Versionsübergang zur eindeutigen Vorgängerversion dokumentiert. Da die Kombination aus Rückwärts- und Vorwärts-Kommandoliste alle Informationen über den Unterschied der betrachteten Version zur Vorgänger-Version beinhaltet, kann sie als *Delta* zwischen den Versionen betrachtet werden.

Während ein Graph verändert wird, werden die ausgeführten Operationen im *Log* des Graphen protokolliert. Wird durch die Anwendung eine neue Version erzeugt, wird der Inhalt des Logs in das neu entstandene Delta übernommen. Das Log ist daraufhin leer. Der Einfachheit halber wird das Delta, das zwischen zwei Versionen besteht, immer der Nachfolger-Version zugeordnet. Das *Delta von Version X* bezeichnet also das Delta zwischen Version X und ihrer (eindeutigen) Vorgänger-Version.

Abbildung 5.4 verdeutlicht das Logging-Konzept anhand eines Beispiels. Im oberen Teil der Abbildung ist auf der linken Seite der Graph `D1` dargestellt, der eine verkettete Listenstruktur beinhaltet, die mit Hilfe von Knoten und Kanten realisiert ist. Der dargestellte Zustand des Graphen ist als Version `V1` bereits im Versionsraum gespeichert. Die Anwendung, die mit dieser Listenstruktur arbeitet, führt nun die Operation `AddListItem` aus, die ein neues Element in die Listenstruktur einfügt. Dazu werden auf Datenbankebene drei atomare graphverändernde Operationen ausgeführt: Zunächst wird ein neuer `ListItem`-

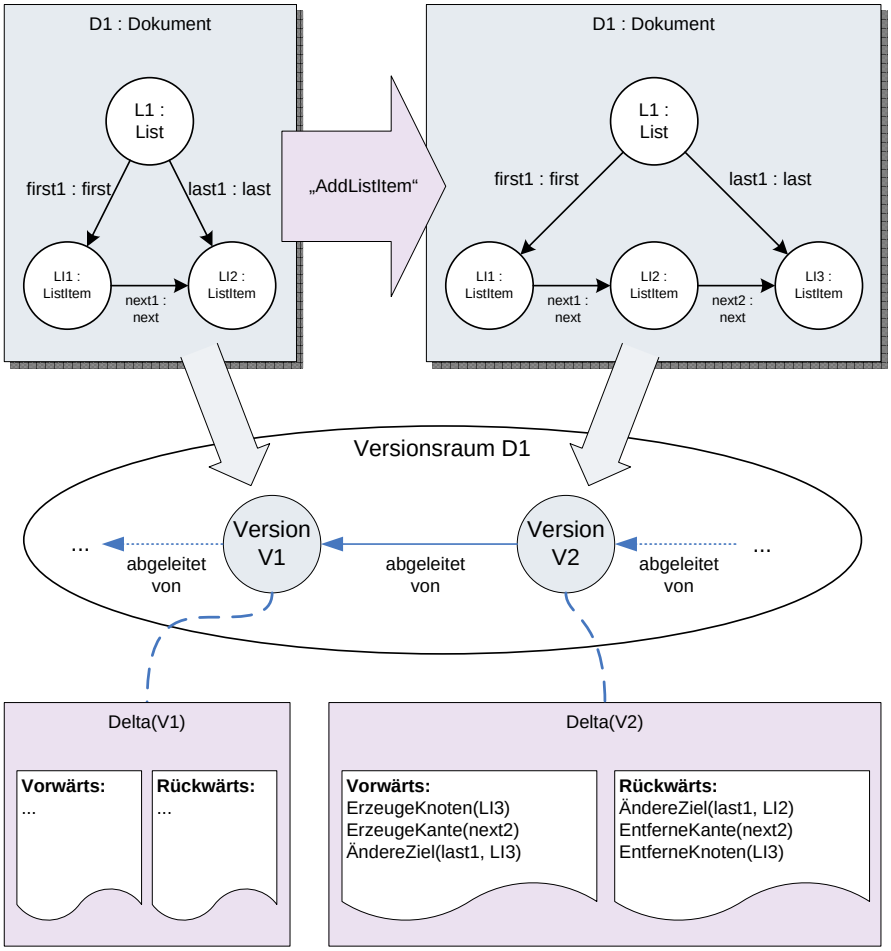

Abbildung 5.4: Versionierung in DRAGOS

Knoten erzeugt und danach eine `next`-Kante, die das letzte Listenelement mit dem neu erzeugten Element verbindet. Zuletzt wird die Ziel-Verbindung der `last` Kante auf den neuen Knoten umgeleitet. Die drei ausgeführten atomaren Operationen werden als Vorwärts-Kommandos im Log des Graphen protokolliert, ihre inversen Operationen als Rückwärts-Kommandos. Wird der neue Zustand des Graphen als Version `V2` gespeichert, werden die protokollierten Kommandolisten als Delta der neuen Version gespeichert. Die Rückwärts-Kommandos sind in der Zeichnung in umgekehrter Reihenfolge dargestellt, um der Abarbeitungsreihenfolge bei einem Versionsrückschritt gerecht zu werden.

Da der in einer Version gespeicherte Zustand nicht mehr veränderlich ist, werden die Kommandolisten eines Deltas nach der Versionserzeugung nicht mehr verändert. Alle ab diesem Zeitpunkt getätigten Operationen werden dann für

die nächste zu erzeugende Version im Log protokolliert.

Die Anwendung soll beliebige Versionen aus dem Versionsraum eines Graphen wiederherstellen können. Wird eine Version gewählt, die nicht direkter Vorgänger oder Nachfolger der aktuellen Version ist, wird der eindeutige Pfad bestimmt, der von der aktuellen Version zur gewählten Version führt. Der Pfad lässt sich als Folge einzelner Versionsübergangsschritte beschreiben. Aufgrund der baumartigen Struktur des Versionsgraphen besteht diese Folge aus einer Sequenz von Rückwärts-Schritten gefolgt von einer Sequenz von Vorwärts-Schritten. Beide Sequenzen können auch leer sein. Durch das Abarbeiten der gesamten Folge wird die Wiederherstellung der gewählten Version erreicht.

Geschachtelte Graphen

Gemäß Abschnitt 5.1.1 gehören zum versionierten Zustand eines Graphen auch rekursiv die versionierten Zustände seiner Untergraphen. Die Möglichkeit, alle den Untergraph verändernden Operationen im Log des Obergraphen zu protokollieren, ist unzweckmäßig, da dadurch der Untergraph nicht mehr getrennt versionierbar ist. Dies impliziert, dass in jeder Graph-Hierarchie Versionierung nur auf einer einzigen Ebene möglich ist. Diese Lösung widerspricht demnach der Anforderung, dass jeder Graph seinen eigenen Versionsraum hat, Versionierung also flexibel auf beliebiger Ebene möglich ist.

Durch ein doppeltes Protokollieren der den Untergraphen verändernden Operationen sowohl im Log des Untergraphen als auch in dem des Vater-Graphen, bleibt die Trennung der Versionsräume beider Graphen bestehen. Diese Realisierung erzeugt jedoch erhebliche Synchronisationsprobleme zwischen den Deltas. Wird eine einzelne Operation rückgängig gemacht, ist sicherzustellen, dass sie in allen betroffenen Deltas rückgängig gemacht wird. Dies ist besonders dann schwierig, wenn mehrere Geschwister-Untergraphen existieren, die abwechselnd verändert werden.

Die für DRAGOS gefundene Lösung besteht darin, jede Graphveränderung wie im Falle von flachen Graphen nur im Log des direkt betroffenen Graphen zu protokollieren. Ändert sich die aktuelle Version eines Graphen wird dieser Vorgang der Versionsänderung im Log des Vater-Graphen protokolliert. Die inverse Operation besteht in diesem Falle aus einem Wiederherstellen der Version, die zuvor die Rolle der aktuellen Version eingenommen hatte.

In Abbildung 5.5 ist dies an einem Beispiel veranschaulicht. Dieses geht davon aus, dass der Untergraph `TS1` in der Ursprungsversion `V0` vorlag als die Version `Vx` von Graph `E1` erzeugt wurde. Die Erzeugung der zwei Modul-Knoten wird

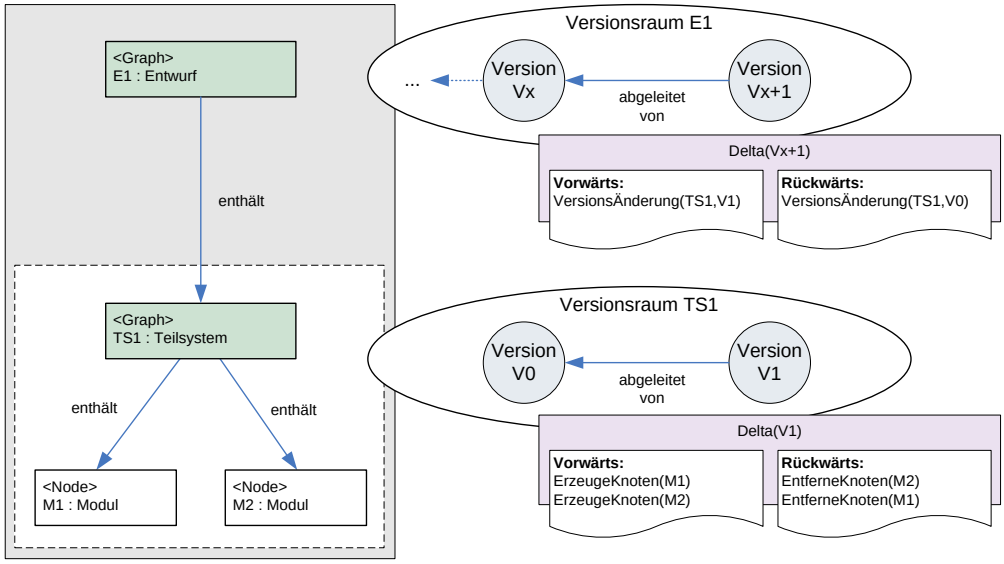

Abbildung 5.5: Veränderung eines Untergraphen

lokal im Log des Untergraphen protokolliert. Erst die Versionserzeugung von `V1`, genauer gesagt die damit verbundene Änderung der aktuellen Version von `TS1`, wird im Log des Vater-Graphen `E1` protokolliert. Der veränderte Zustand von `E1` wird daraufhin ebenfalls als neue Version `Vx+1` gespeichert. Beim Wiederherstellen der Vorgängerversion `Vx` wird das Rückwärtskommando abgearbeitet, das für den Untergraphen `TS1` die Ursprungsversion wiederherstellt. Im Zuge dieses Schrittes werden die beiden Modul-Knoten wieder entfernt, der in `Vx` gespeicherte Zustand ist also korrekt rekonstruiert.

Damit der oben dargestellte Mechanismus funktioniert, muss bei der Versionserzeugung sichergestellt werden, dass alle Änderungen am versionierten Zustand des betreffenden Graphen gespeichert werden. Dazu müssen rekursiv Versionserzeugungen von Untergraphen initiiert werden, falls diese verändert wurden. Zur Formalisierung dieses Zusammenhangs, ist folgende Definition notwendig:

Ein Graph heißt *ungeändert*, wenn

1. sein Log leer ist und
2. jeder seiner Untergraphen ungeändert ist.

Ist ein Graph demnach *nicht* ungeändert, existiert an mindestens einer Stelle innerhalb seines versionierten Zustands eine Änderung, die nicht in einem Versionsdelta gespeichert wurde. Die Versionserzeugung muss sicherstellen, dass auch solche Änderungen gespeichert werden, die sich im Log eines direkten

oder indirekten Untergraphen befinden. Dies wird erreicht, indem die Versionserzeugung nach folgendem Algorithmus abläuft:

```
erzeugeVersion(G): {
  Für jeden Untergraph U von G: {
    falls (U ist nicht ungeändert): {
      erzeugeVersion(U);
    }
  }
  lege Version V an;
  Delta(V) := Log(G);
  Protokolliere Versionsänderung im Vater-Graph von G;
}
```

Für jeden im versionierten Graphen enthaltenen Untergraphen wird überprüft, ob er ungeändert ist. Ist dies nicht der Fall, wird rekursiv eine Versionserzeugung auf dem Untergraphen angestoßen, die wiederum als Versionsänderung im Log des Vater-Graphen protokolliert wird. Änderungen, die in tieferliegenden Graphen stattgefunden haben, werden also indirekt als Versionsänderungs-Kommandos in den höherliegenden Ebenen protokolliert. Auf diese Weise werden alle Änderungen am versionierten Gesamtzustand bis in das Log des versionierten Graphen propagiert. Am Ende des Vorgangs wird die Log-Information als Delta der neuen Version gespeichert.

Der dargestellte Logging-Mechanismus stellt eine geeignete Möglichkeit dar, um Graphzustände speichern und wiederherstellen zu können. Werden keine graphübergreifenden Verbindungen verwendet (siehe dazu Abschnitt 5.1.3), so bleibt die Konsistenz des Graphpools dabei stets gewahrt.

Anwendungsspezifisches Logging

Neben den in den vorherigen Abschnitten vorgestellten Vorgängen, die für den Versionierungsmechanismus notwendig sind, wird das Logging-Konzept zusätzlich um einen anwendungsspezifischen Aspekt erweitert. So können an beliebiger Stelle Kommentare in das Log eines Graphen eingefügt werden, die für den Versionierungsprozess keine Bedeutung haben, aber von der Anwendung interpretiert werden können. Dazu stehen Kommentar-Markierungen und Custom-Kommandos zur Verfügung.

Kommentar-Markierungen tragen eindeutige Bezeichner und können beliebige Zusatzformationen speichern. Es existieren zwei unterschiedliche Arten von Kommentar-Kommandos: Einzel-Markierungen und Gruppierungs-Markierungen. Einzel-Markierungen kommen einzeln vor und markieren eine Stelle im protokollierten Kommandostrom. Gruppierungs-Markierungen hingegen bestehen immer aus einem öffnenden und einem schließenden Kommentar-Kommando, die den gleichen Bezeichner tragen. Damit ist es möglich, eine beliebige Menge anderer Kommandos zu umschließen, um so eine Folge von Änderungen zu gruppieren. Gruppierungs-Markierungen können beliebig geschachtelt sein. Bei der Versionserzeugung werden für alle noch offenen Gruppierungen schließende Kommandos eingefügt. Ist ein Kommentar-Kommando in einer Kommandoliste enthalten, die im Zuge einer Versionswiederherstellung abgearbeitet wird, erzeugt es ein Ereignis im Ereignis-Manager von DRAGOS. Dadurch wird der Anwendung ermöglicht, nach definierbaren Regeln auf einen Kommentar zu reagieren.

Neben Kommentaren hat die Anwendung zudem die Möglichkeit, eigene Kommandos zu definieren, und in den protokollierten Kommandostrom einzufügen. Diese sogenannten *Custom-Kommandos* können aktive Aspekte beinhalten, die ausgeführt werden, wenn sie bei einer Versionswiederherstellung abgearbeitet werden.

Zusammenfassung

Der Logging-Mechanismus umfasst drei Arten von Protokollierungsaktivitäten, die in Abbildung 5.6 zusammenfassend dargestellt sind. Notwendig für die Basis-Versionierung ist das Protokollieren von atomaren Graphveränderungen und Versionsänderungen. Anwendungen haben darüber hinaus die Möglichkeit anwendungsspezifische Kommandos in die Logs einzufügen.

Graphveränderungen werden lokal im Log des Graphen protokolliert, dessen versionierter Zustand direkt durch die Änderungen betroffen wird. Aus diesem Grund werden Attributsänderungen an Knoten, Kanten, Relationen und Relationsenden im Log des Vater-Graphen protokolliert. Attributsänderungen an einem Untergraphen hingegen werden im Log des Untergraphen selbst protokolliert, da dessen Zustand durch die Änderung direkt betroffen ist.

Versionsänderungen werden im Log des Vater-Graphen protokolliert. Da der Graphpool selbst nicht versionierbar ist, entfällt dieser Aspekt des Loggings für Graphen auf oberster Graphpool-Ebene. Ihre Versionsänderungen werden

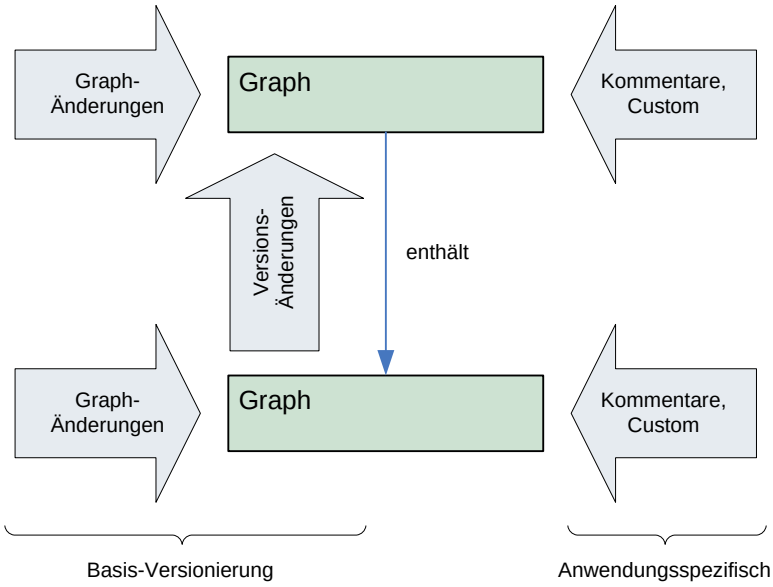

Abbildung 5.6: Logging in DRAGOS

nur dann protokolliert, wenn die im nächsten Abschnitt 5.1.3 vorgestellte Versionsgruppen eingesetzt werden.

Die Anwendung hat die Möglichkeit, die Logs beliebiger Graphen mit anwendungsspezifischen Kommandos anzureichern, um so zusätzliche Informationen zu speichern. Neben Kommentaren werden auch von der Anwendung definierte Kommandos akzeptiert.

5.1.3 Graphübergreifende Verbindungen

In Abschnitt 5.1.1 wurde bereits auf die Problematik graphübergreifender Verbindungen von Kanten und Relationsenden aufmerksam gemacht. In Bezug auf die Versionierung ist für derartige Verbindungen charakteristisch, dass sie nicht innerhalb des versionierten Zustandes *eines* Graphen existieren, sondern vom versionierten Zustand eines zweiten Graphen abhängig sind. Durch die getrennte Versionierung der beiden betroffenen Graphen ist nicht gewährleistet, dass Verbindungsreferenzen bei einer Versionswiederherstellung rekonstruiert werden können. Dabei ist zu beachten, dass im Falle einer Kante sowohl Quell- als auch Ziel-Verbindung graphübergreifend sein können, wenn sich beide referenzierten Graphelemente außerhalb des Vater-Graphen der Kante befinden. Von einer einzelnen graphübergreifenden Verbindung sind jedoch je zwei Graphen betroffen.

Die in Abbildung 5.1 dargestellte Situation verdeutlicht diesen Zusammenhang. Die Kante `V1` referenziert mit ihrer Ziel-Verbindung ein Graphelement, das nicht im gleichen Graph enthalten ist wie die Kante selbst. Werden die Graphen `TS1` und `TS2` getrennt versioniert, kann es zu einer Inkonsistenz des Verbindungszustandes von `V1` und damit des versionierten Zustand von `TS1` kommen. Angenommen `TS1` wird in den Zustand der Ursprungsversion versetzt, enthält also keine Graphelemente. Wird Graph `M2` gelöscht und anschließend für `TS1` die im Bild dargestellte Version wiederhergestellt, kann die Ziel-Referenz der Kante nicht rekonstruiert werden, da das referenzierte Graphelement nicht mehr existiert.

Neben dem dargestellten Problem nicht mehr existierender Graphelemente können auch Kardinalitätsverletzungen auftreten. Angenommen zu verschiedenen Zeitpunkten existieren in verschiedenen Graphen Kanten, die das gleiche Graphelement *G* referenzieren. Dann kann durch Versionsveränderungen der Graphen ein Gesamtzustand erzeugt werden, in dem alle diese Kanten gleichzeitig existieren. Ist die Kardinalität des gemeinsamen Kantentyps beschränkt, kann dabei die maximal erlaubte Anzahl der zu *G* inzidenten Kanten überschritten werden. Eine Konsistenz-Prüfung des Graphpools würde folglich einen Fehler melden.

Graphübergreifende Verbindungen können die dargestellten Probleme nur dann verursachen, wenn sie die Grenze eines versionierten Zustandes überschreiten. Sind beide betroffenen Graphen Teil eines gemeinsamen versionierten Zustandes, ist die Konsistenz zwischen ihnen gewährleistet. Insbesondere ist dies der Fall, wenn einer der Graphen versioniert wird und der andere direkt oder indirekt in ihm enthalten ist. Eine allgemeine Möglichkeit zur Vermeidung von Inkonsistenzen besteht darin, nur gemeinsame Obergraphen von konsistent zu haltenden Graphen zu versionieren. In Abbildung 5.7 ist dies für das vorherige Beispiel dargestellt. Indem nur noch Versionen des Graphen `E1` erzeugt und wiederhergestellt werden, gehören `TS1` und `TS2` zum versionierten Gesamtzustand.

Durch diese Vorgehensweise zur Vermeidung von Inkonsistenzen wird jedoch die Flexibilität der Versionskontrolle erheblich eingeschränkt. Der Anwendungsentwickler muss schon bei der Modellierung der Dokumente berücksichtigen, zwischen welchen Graphen konsistent zu haltende Verbindungen bestehen werden und entsprechende gemeinsame Obergraphen einplanen. Ein nachträgliches Ändern der Versionierungsebene ist somit nur eingeschränkt möglich.

GRAS3 löst das Problem über die Möglichkeit zur Definition sogenannter *Log-Gruppen*. Diese ermöglichen es, Graphen dynamisch zu gruppieren, um die

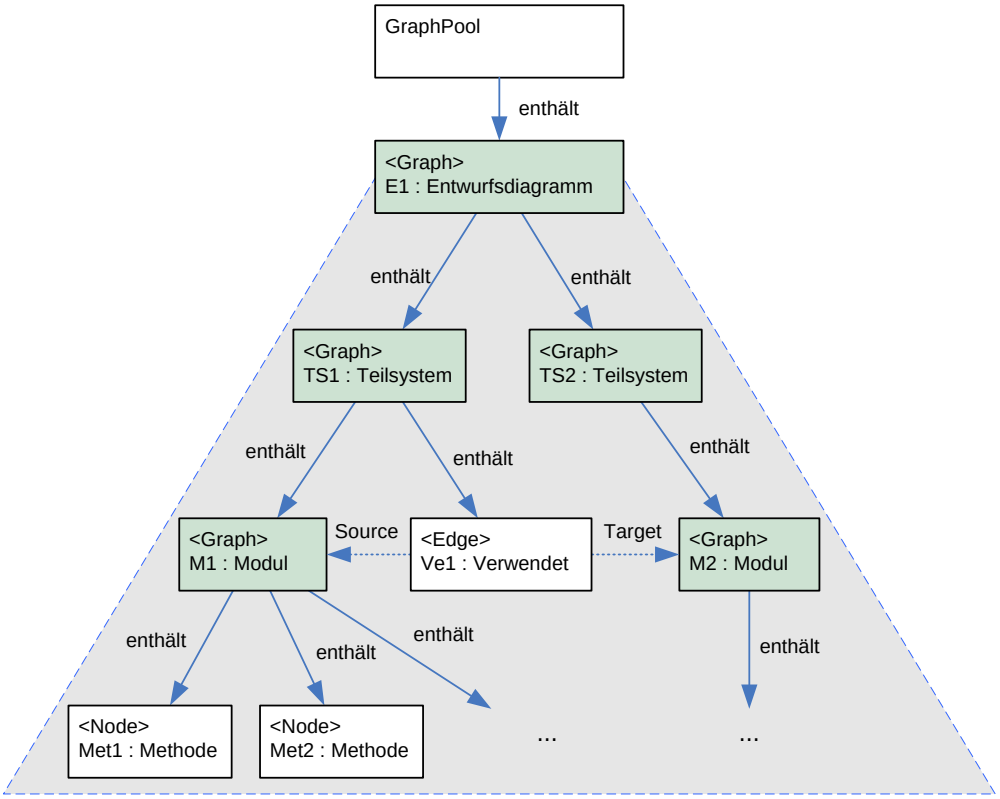

Abbildung 5.7: Versionierter Obergraph

Konsistenz graphübergreifender Kanten gewährleisten zu können. Wird ein Graph in eine Log-Gruppe aufgenommen, werden alle zukünftigen graphverändernden Operationen nicht mehr im Log des Graphen selbst, sondern in einem für alle Gruppenmitglieder gemeinsamen Log protokolliert. Die Erzeugung graphübergreifender Kanten wird nur dann in der Log-Gruppe protokolliert, wenn beide beteiligten Graphen Mitglieder der Gruppe sind. Dadurch können weiterhin Inkonsistenzen entstehen, wenn eine graphübergreifende Kante existierte, bevor einer der Graphen zur Gruppe hinzugefügt wurde. Zudem ist die Flexibilität dieses Verfahrens beschränkt, da Graphen nur unter der Bedingungen in Log-Gruppen aufgenommen werden dürfen, dass sie nicht geöffnet sind und keine Log-Informationen enthalten. Log-Gruppen sind demnach nur für die kurzfristige Änderungsverwaltung konzipiert, da in der langfristigen Änderungsverwaltung die zweite Bedingung nur für leere Graphen erfüllt ist.

Versionsgruppen

Das Konzept der Log-Gruppen kann nur als Ansatz zur Lösung der Problematik in DRAGOS verwendet werden. Da kurzfristige und langfristige Versionskontrolle nicht unterschieden werden, beinhaltet jeder nicht-leere Graph in DRAGOS Log-Informationen, die im Allgemeinen niemals gelöscht werden. Zudem sind geschachtelte Graphen zu berücksichtigen, die auf allen Ebenen versioniert werden können. Als an diese speziellen Voraussetzungen angepasstes Konstrukt wurden *Versionsgruppen* entwickelt. Ähnlich wie Log-Gruppen werden sie verwendet, um Graphen zu gruppieren, zwischen denen graphübergreifende Verbindungen existieren. Versionsgruppen sind genauso wie Graphen versionierbare Objekte, das heißt die Anwendung kann Versionen von ihnen erzeugen und wiederherstellen.

Der *versionierte Zustand einer Versionsgruppe* setzt sich zusammen aus:

1. der Komposition ihrer Mitgliedsgraphen, sowie
2. dem versionierten Zustand ihrer Mitgliedsgraphen.

Durch den zweiten Punkt der Definition ist rekursiv auch der Zustand aller Untergraphen von Mitgliedsgraphen erfasst. Versionsgruppen stellen also eine flexible Versionierungsebene bereit, die dynamisch an sich ändernde Graphstrukturen angepasst werden kann. Dabei wird die übliche Logging-Aktivität der Mitgliedsgraphen nicht verändert sondern nur erweitert, wodurch auch die Mitgliedschaft eines Graphen in mehreren Versionsgruppen ermöglicht wird. Das erweiterte Logging ist in Abbildung 5.8 dargestellt. Während alle Graphveränderungen an einem Versionsgruppenmitlied weiterhin lokal protokolliert werden, werden Versionsänderungen nicht mehr nur im Vater-Graphen protokolliert, sondern zusätzlich in der Versionsgruppe. Das Versionieren der Versionsgruppe kommt demzufolge dem Versionieren eines gemeinsamen Vater-Graphen gleich. Die Konsistenz der graphübergreifenden Verbindungen ist also nur dann garantiert, wenn keine Versionswiederherstellungen von Mitgliedsgraphen oder ihren Untergraphen vollzogen werden.

Der versionierte Zustand der Versionsgruppe beinhaltet auch die Komposition der Mitgliedsgraphen. Dies ist dadurch begründet, dass die Konsistenz nur dann gewährleistet ist, wenn beim Erzeugen einer graphübergreifenden Verbindung beide beteiligten Graphen Mitglieder der Gruppe sind. Ein Zustand einer Versionsgruppe beinhaltet deshalb nicht die nachträglich aufgenommenen Graphen. Das Aufnehmen und Entfernen von Mitgliedern wird im Log der Versionsgruppe protokolliert.

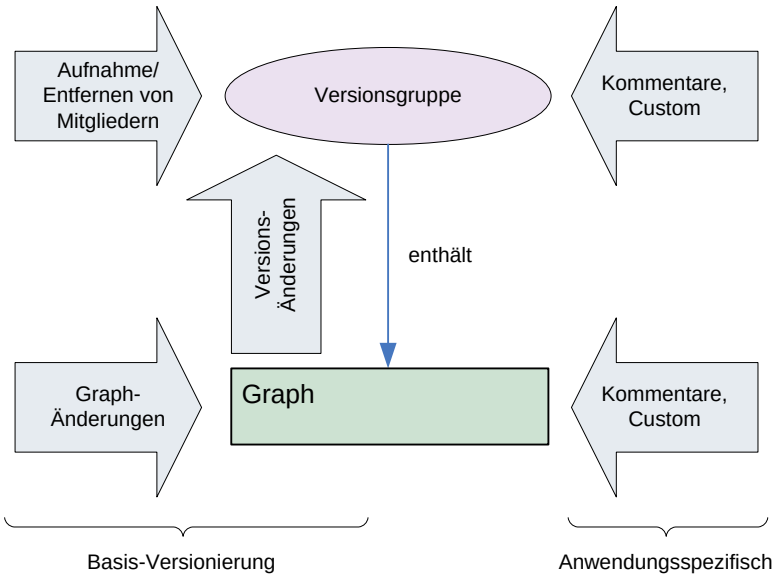

Abbildung 5.8: Logging und Versionsgruppen

Der anwendungsspezifische Aspekt des Logging-Konzeptes wird auch auf Versionsgruppen ausgeweitet, so dass sich ihre Kommandolisten mit Kommentaren und anwendungsspezifischen Kommandos anreichern lassen.

Das Erzeugen einer Version einer Versionsgruppe läuft analog zur Versionserzeugung für Graphen ab: Für jeden Mitgliedsgraphen wird überprüft, ob er ungeändert ist. Ist dies nicht der Fall, muss eine Versionserzeugung auf dem Mitgliedsgraphen initiiert werden.

Abbildung 5.9 zeigt beispielhaft zwei Graphen, anhand derer der Einsatz einer Versionsgruppe demonstriert wird. In diesem Fall ist das Beispiel so gewählt, dass die beiden Graphen Dokumentteile aus unterschiedlichen Phasen des Softwareentwicklungszyklus darstellen: `TS1` ist erneut ein Teilsystem aus einem Entwurfsdiagramm, während `P1` ein Paket aus der Implementierungsphase darstellt, das eine programmiersprachlich implementierte Klasse beinhaltet. Entwurfsdokumente und Pakete seien so modelliert, dass sie keine gemeinsamen Obergraphen besitzen. Trotzdem kann eine Softwareentwicklungsumgebung Beziehungen zwischen ihnen verwalten, die in DRAGOS als graphübergreifende Kanten modelliert werden. Im Beispiel stellt die Kante `I1` eine solche Beziehung dar. Ein Zustand in dem die Ziel-Referenz von `I1` undefiniert ist, ist aus Anwendungssicht nicht erwünscht. Um die Zustände beider Graphen konsistent zu halten, wird deshalb eine Versionsgruppe definiert, in die beide Graphen aufgenommen werden. Die Aufnahme-Operationen haben

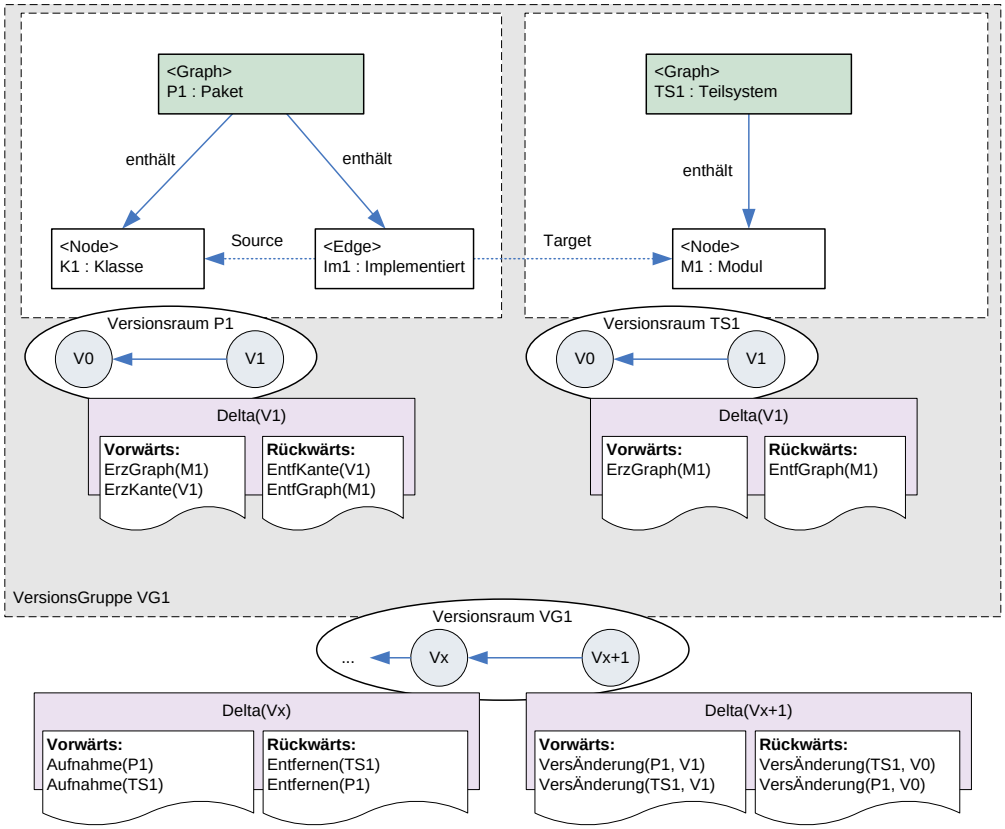

Abbildung 5.9: Beispiel für eine Versionsgruppe

bereits vor der Erzeugung von Version `Vx` stattgefunden. Die Anwendung versioniert nun den gemeinsamen Zustand von `P1` und `TS1` auf Versionsgruppen-Ebene. Bei der Erzeugung der Version `Vx+1` werden rekursiv Versionserzeugungen der geänderten Mitgliedsgraphen initiiert. Die hervorgerufenen Versionsänderungen werden wiederum im Log der Versionsgruppe protokolliert und im Delta von Version `Vx+1` gespeichert. Version `Vx+1` repräsentiert also den im Bild dargestellten Zustand, während der Zustand der Vorgänger-Version `Vx` zwei leere Mitgliedsgraphen enthält. Ein inkonsistenter Zustand, in dem zwar Kante `Im1` nicht jedoch ihre Ziel-Referenz `M1` vorhanden ist, existiert auf Versionsgruppen-Ebene nicht.

Einsatz von Versionsgruppen

Versionsgruppen stellen für den Anwendungsentwickler ein flexibles Hilfsmittel für die Versionierung graphübergreifender Verbindungen dar. Die Flexibili-

tät hat jedoch auch Grenzen, die im Folgenden näher betrachtet werden.

Zunächst einmal existiert die Beschränkung, dass ein Graph nur dann in eine Versionsgruppe aufgenommen werden kann, wenn

1. keiner seiner direkten oder indirekten Obergraphen und

2. keiner seiner direkten oder indirekten Untergraphen

bereits Mitglied derselben Versionsgruppe ist. Betrachtet man also zwei Graphen, von denen einer direkter oder indirekter Untergraph des anderen ist, kann nur einer von beiden Mitglied der Versionsgruppe sein. Die Mitgliedschaft zweier Graphen aus der gleichen Graph-Hierarchie ist also eine Verletzung dieser Vorbedingungen. Dieser Fall widerspricht jedoch dem Sinn einer Versionsgruppe, graphübergreifende Verbindungen konsistent zu halten. Da sich der tieferliegende Graph ohnehin im versionierten Zustand des höherliegenden befindet, bringt die gemeinsame Mitgliedschaft in einer Versionsgruppe keinen Vorteil. Durch sich verändernde graphübergreifende Verbindungen sind jedoch Anwendungsfälle denkbar, in denen es nicht nur gewünscht, sondern auch unproblematisch wäre, nachträglich die Mitgliedschaft eines Graphen auf einen direkten oder indirekten Obergraphen auszudehnen, beziehungsweise auf einen oder mehrere seiner Untergraphen zu beschränken. Im Folgenden werden die Algorithmen erläutert, die diese Operationen durch die Abfolge mehrerer Entfernen- und Aufnahme-Operationen simulieren.

Eine Ausdehnung der Mitgliedschaft eines Graphen G auf einen beliebigen direkten oder indirekten Obergraph O ist problemlos zu bewerkstelligen. In diesem Fall ist die erste der zuvor erwähnten Vorbedingungen für O erfüllt, die zweite lässt sich erfüllen, indem der Baum von Untergraphen mit der Wurzel O traversiert wird und jedes gefundene Mitglied der Versionsgruppe aus dieser entfernt wird. Anschließend kann O in die Versionsgruppe aufgenommen werden.

Soll die Mitgliedschaft eines Graphen G auf die Untergraphen U_1 bis U_n eingeschränkt werden, ist für jeden dieser Graphen zunächst nur die zweite Vorbedingung erfüllt. Die erste kann unmittelbar erfüllt werden, indem G aus der Versionsgruppe entfernt wird. Anschließend können U_1 bis U_n aufgenommen werden.

Auf die Umsetzung dieser Funktionalität in DRAGOS wurde verzichtet, da ihre Simulation, wie gezeigt, durch die bestehenden Operationen zum Aufnehmem

und Entfernen von Mitgliedern problemlos möglich ist. Ein automatisches Abarbeiten der dargestellten Algorithmen ist aber nicht gewünscht. Dies ist begründet durch die Eigenschaft, dass Versionsgruppen nur dann die Konsistenz graphübergreifender Verbindungen gewährleisten können,

1. wenn beide betroffenen Graphen zum Zeitpunkt der Erzeugung der Verbindung bereits Mitglieder der Versionsgruppe sind, und
2. solange beide betroffene Graphen Mitglieder der Gruppe bleiben.

Veränderungen des versionierten Zustandes der Versionsgruppe sollten der Kontrolle der Anwendung unterliegen, da nur sie Informationen darüber besitzt, zwischen welchen Graphen konsistent zu haltende Beziehungen bestehen. Um nicht zusätzliche Fehlerquellen bei der Anwendungsentwicklung zu schaffen, ist es sinnvoll, nur die Basis-Funktionalität des Aufnehmens und Entfernens von Mitgliedern zur Verfügung zu stellen.

5.1.4 Konfigurationen

Im Bereich des Softwarekonfigurations-Managements werden nicht nur Versionen eines Dokuments verwaltet, sondern auch dokumentübergreifende Konfigurationen. Der Begriff *Konfiguration* bezeichnet eine Konstellation von Versionen, die gemeinsam bestimmte Konsistenz-Kriterien erfüllen. Der Konsistenzbegriff wird dabei um eine anwendungsseitige semantische Dimension erweitert, die über die Konsistenz der Datenbasis hinausgeht. In einem Softwareentwicklungsprojekt stellt beispielsweise ein veröffentlichter Release eine Konfiguration des Software-Systems dar. Diese enthält die Gesamtheit der Versionen von Programmcode-Dokumenten, die zusammen das an einen Kunden ausgelieferte System ergeben.

Da in DRAGOS Dokumente in Form von Graphen gespeichert werden, ist es sinnvoll, den Konfigurationsbegriff auf Graphen zu abstrahieren. Demzufolge ist eine Konfiguration eine Konstellation von Graph-Versionen. Die Semantik der Konsistenz dieser Konstellation wird von der Anwendung definiert und ist auf Datenbank-Ebene zu vernachlässigen. Eine Konfiguration stellt also einen Querschnitt durch die Versionsräume der von ihr betroffenen Graphen dar. Dies ist in Abbildung 5.10 veranschaulicht. Die im linken Teil der Abbildung dargestellte Konfiguration repräsentiert einen konsistenten Gesamtzustand, der aus den Versionen `V3`, `V1` und `V2` der jeweiligen Graphen besteht. Im Unterschied dazu besteht die rechts dargestellte Konfiguration aus den Versionen `V3`, `V2` und `V3`. Version `V2` des Graphen *1* kommt also in beiden Konfigurationen vor.

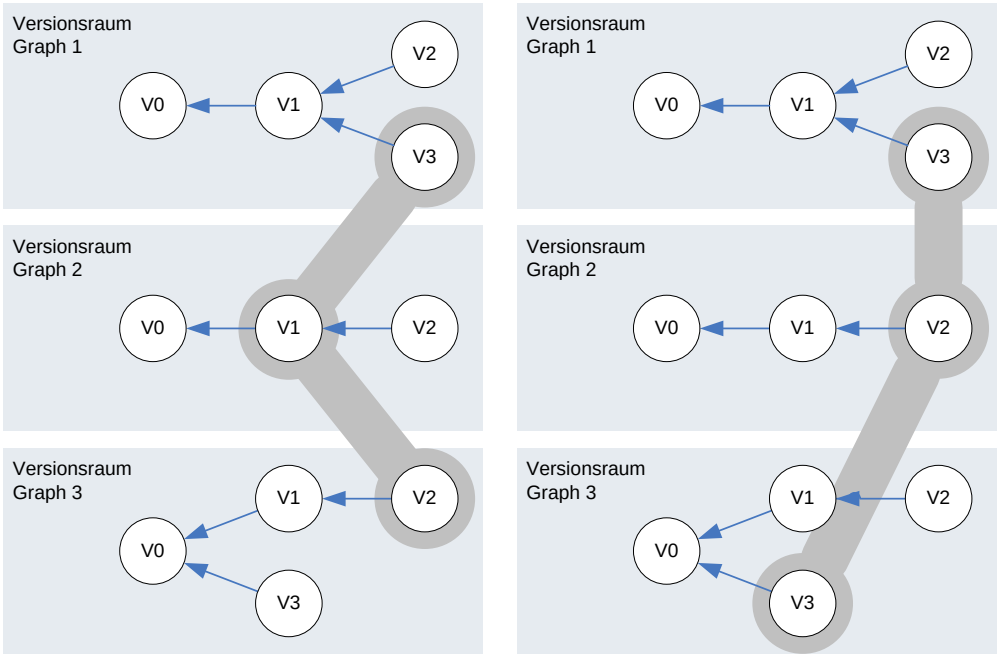

Abbildung 5.10: Konfigurationen

Der aktuelle Gesamtzustand des Graphpools stellt eine Konfiguration bestehend aus der aktuellen Version jedes in ihm existierenden Graphen dar. Genauso stellt jede beliebige Version eines Graphen eine Konfiguration aus den aktuellen Versionen der enthaltenen Untergraphen dar. Die Nutzung von Obergraphen zur Verwaltung von Graph-Konfigurationen ist jedoch nicht zweckmäßig, da eine Erweiterung der Konfiguration auf beliebige Graphen außerhalb des Obergraphen nicht möglich ist. Auch der Einsatz von Versionsgruppen ist ungeeignet, da jede zu speichernde Konfiguration ihrer Mitgliedsgraphen im Graphpool explizit erzeugt werden muss, um sie als Version der Versionsgruppe und damit als Konfiguration abspeichern zu können. Zudem ist die Aufnahmebeschränkung für Versionsgruppenmitglieder unakzeptabel für Konfigurationen, in denen es durchaus zweckmäßig sein kann, eine Version eines Graphen und gleichzeitig Versionen beliebiger direkter oder indirekter Untergraphen zu verwalten.

DRAGOS-Konfigurationen, im folgenden nur noch als Konfigurationen bezeichnet, stellen ein Konstrukt zur Verfügung, das die flexible Definition von Versionskonstellationen zulässt. Die Aufnahme von Versionen in eine Konfiguration unterliegt dabei keinen Einschränkungen, die Semantik kann also frei durch die Anwendung bestimmt werden. In einer Konfiguration kann von jedem Graphen nur eine Version enthalten sein. Wird eine zweite Version eines

Graphen hinzugefügt, ersetzt sie die vorhandene. Das Entfernen einer Version aus einer Konfiguration ist ebenfalls ohne Einschränkungen möglich.

Eine mächtige Eigenschaft von Konfigurationen ist ihre Versionierbarkeit. Da sie durch Hinzufügen, Ersetzen und Entfernen von Versionen Veränderungen unterworfen sein können, ist es sinnvoll, Zustände der Konfiguration speichern und wiederherstellen zu können. Im Beispiel aus Abbildung 5.10 könnten die beiden Konfigurationen im linken und rechten Teil des Bildes demnach auch als zwei Versionen derselben Konfiguration aufgefasst werden. Das Delta der rechten Version besteht dann aus zwei Operationen, die nacheinander `V2` von Graph 2 und `V3` von Graph 3 zur Konfiguration hinzugefügen, und damit die zuvor enthaltenen Versionen der beiden Graphen ersetzen.

Die DRAGOS-Versionskontrolle stellt einen Mechanismus zur Verfügung, mit dem der Graphpool in den von einer Konfiguration definierten Zustand versetzt werden kann. Bei diesem als *Ausführung (Apply)* bezeichneten Vorgang wird für jeden Graph, von dem eine Version enthalten ist, versucht, den Zustand der enthaltenen Version wiederherzustellen. Die Reihenfolge ist dabei festgelegt durch die Tiefe der Schachtelungsebene auf denen sich die Graphen befinden. So wird, einer Breitensuche ähnlich, die Versionswiederherstellungen für Graphen, die sich näher an der Wurzel der Graph-Hierarchie befinden, eher angestoßen als für die Graphen auf tieferen Ebenen. Auf diese Weise wird verhindert, dass die Versionsänderung eines Obergraphen die eines direkten oder indirekten Untergraphen überschreibt.

Da die Semantik der Konfiguration allein der Kontrolle der Anwendung unterliegt, existiert keine Möglichkeit zu gewährleisten, dass bei Ausführung einer Konfiguration die Konsistenz des Graphpools gesichert ist. Die im vorherigen Abschnitt dargestellte Problematik graphübergreifender Kanten ist also anwendungsseitig zu lösen.

Abgesehen von der Konsistenz-Problematik kann bei der Ausführung einer Konfiguration das Problem auftreten, dass von der Konfiguration betroffene Graphen im aktuellen Zustand des Graphpools nicht existieren. Grund hierfür kann entweder das Ausführen einer expliziten Lösch-Aktion durch die Anwendung oder eine Versionsänderung eines Obergraphen sein. Um der Anwendung Flexibilität beim Umgang mit dieser Situation zu ermöglichen, stehen drei verschiedene Ausführungsarten zur Verfügung:

1. **Apply:** Ein nicht existierender Graph erzeugt einen Fehler, keine enthaltene Version wird wiederhergestellt.

2. **Apply And Ignore:** Nicht existierende Graphen werden ignoriert, enthaltene Versionen werden für alle existierenden Graphen wiederherstellt.
3. **Apply And Force:** Nicht existierende Graphen werden wiederhergestellt, alle enthaltenen Versionen werden wiederherstellt.

Bei der letzten Ausführungsart wird das Wiederherstellen von nicht existierenden Graphen erreicht, indem im Vater-Graph die inverse Operation zur ausgeführten Entfernen-Operation ausgeführt wird. Eventuell nicht mehr exitierende Obergraphen werden vorher ebenfalls wiederhergestellt.

5.1.5 Replikation von Graphen

Wie in Abschnitt 3.3 erläutert, stellt die Replizierbarkeit von Graphen die Grundlage für eine Reihe weitergehender Anwendungsmöglichkeiten der Versionskontrolle dar. Insbesondere das Erzeugen von Arbeitskopien der Daten für den Mehrbenutzerbetrieb wird hierbei angestrebt.

Im Folgenden wird davon ausgegangen, dass nur solche Zustände eines Graphen repliziert werden, die als Version gespeichert wurden. Da jeder Zustand gespeichert werden kann, stellt dies keine Einschränkung der Allgemeingültigkeit dar.

Der Logging-Mechanismus stellt alle Informationen bereit, die zur Replikation eines Graphen benötigt werden. Am Beginn des Replikationsvorgangs steht das Wiederholen der Operation, die zur Erzeugung des Original-Graphen führte. Mit ihr kann die Anwendung an beliebiger Stelle im Graphpool einen Graphen erzeugen, der die gleiche Graphklasse besitzt wie der Original-Graph. Der neue Graph befindet sich also in einem der Ursprungsversion des Original-Graphen äquivalenten Zustand. Anschließend wird im Versionsgraphen des Original-Graphen der Pfad identifiziert, der von der Ursprungsversion bis zu der Version führt, die den zu replizierenden Zustand darstellt. Die Vorwärts-Deltas der Version auf diesem Pfad werden dann der Reihe nach abgearbeitet. Bei der Ausführung der einzelnen enthaltenen Kommandos dient jedoch der neue Graph als Grundlage der Veränderung. Sein versionierter Zustand entspricht anschließend der zu replizierenden Version des Original-Graphen.

Der beschriebene Vorgang ist nicht auf den ursprünglichen Graphpool beschränkt. Das Erstellen einer Arbeitskopie für den Mehrbenutzerbetrieb soll beispielsweise in einem lokalen Graphpool erfolgen. Dazu muss die Voraussetzung erfüllt sein, dass das Schema des neuen Graphpools alle Definitionen beinhaltet, die beim Aufbau des Graphen benötigt werden. Da das Schema nicht

Gegenstand der Versionierung ist, müssen dazu weitergehende Mechanismen entwickelt werden.

Existiert bereits eine Arbeitskopie eines Graphen, kann der Replikationsvorgang auch verwendet werden, um die Arbeitskopie mit dem Server zu synchronisieren. Dies geschieht, indem nur die Änderungen seit der letzten Erzeugung der Arbeitskopie repliziert werden. Auf Merge-Mechanismen, die mit auftretenden Konflikten umgehen können, geht der nächste Abschnitt näher ein.

5.1.6 Zusammenführen von Versionen

Bei der Formulierung der Anforderungen an die Versionkontrolle wurde in Abschnitt 3.4 auch die Möglichkeit zum Zusammenführen von Versionen gefordert. Dabei soll die Baumstruktur des Versionsgraphen erhalten bleiben, da die explizite Verwaltung von mehreren Vorgänger-Beziehungen nicht benötigt wird. Basierend auf der Replikationsfähigkeit von Änderungen, die im letzten Abschnitt erläutert wurde, kann ein Merge-Mechanismus entwickelt werden. Das Prinzip ist in Abbildung 5.11 dargestellt. Ausgangssituation für eine Merge-Operation sind stets zwei Versionen aus parallelen Entwicklungszweigen, die verschmolzen werden sollen. Dabei wird die *Hauptversion* von der *Nebenversion* unterschieden. Von der Hauptversion (im Bild *V6*) wird die resultierende Version abgeleitet, während zur Nebenversion (*V4*) keine Ableitungsbeziehung entsteht. Über Meta-Attribute wird jedoch eine implizite Beziehung erzeugt, die von der Anwendung für weitergehende Funktionalität genutzt werden kann.

Das Zusammenführen der Versionen wird erreicht, indem zunächst die Version identifiziert wird, die den Ausgangspunkt der Verzweigung darstellt. Die Deltas entlang des eindeutigen Pfades von dieser Wurzelversion zur Nebenversion beinhalten die zu replizierenden Änderungen. Als Ausgangspunkt für die Replikation dient die Hauptversion.

Würde die in Abbildung 5.11 gezeigte Situation ein Mehrbenutzerszenario darstellen, könnte beispielsweise von Version *V2* eine Arbeitskopie erstellt worden sein, die lokal als Version *V3* und *V4* weiterentwickelt wurde. Parallel dazu wurde der Hauptentwicklungszweig bis zur Version *V6* weitergeführt. Nun sollen die lokalen Änderungen mit dem Hauptentwicklungszweig verschmolzen werden. Dazu werden die Änderungen der Deltas zwischen der Wurzelversion *V2* und Version *V4* im wiederhergestellten Zustand von Version *V6* repliziert. Die resultierende Version *V7* enthält dann alle Änderungen des lokalen Entwicklungszweigs.

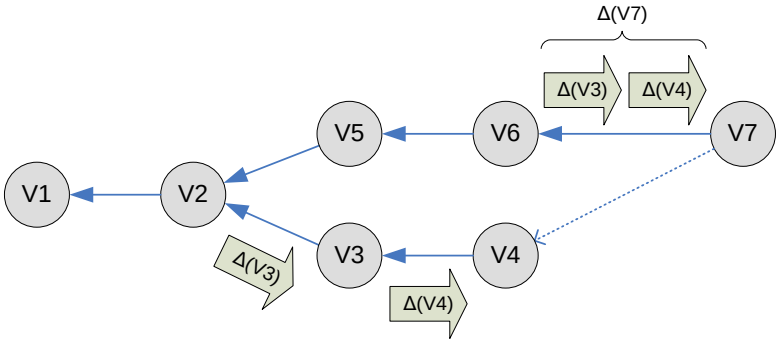

Abbildung 5.11: Zusammenführen von Versionen

Beim Ausführen der Änderungen können Konflikte auftreten. Ist im Hauptentwicklungszweig zwischenzeitlich beispielsweise ein Graphelement gelöscht wurden, kann eine protokollierte Attributsänderung an diesem Graphelement nicht repliziert werden. Die Lösung für solche Konflikte ist Aufgabe der Anwendung, da sie die Semantik von Änderungen bestimmt. DRAGOS kann diesbezüglich nur Unterstützung leisten. Derartige Funktionalität übersteigt jedoch den Umfang dieser Arbeit.

5.1.7 Zusammenfassung

Die vorgestellten Konzepte basieren auf dem Protokollieren aller Änderungen, die eine Anwendung an der Datenbasis vornimmt. Ähnliche Mechanismen sind sowohl in GRAS3 als auch in CoObRA zu finden. Obwohl GRAS3 ebenfalls Daten in Form von Graphen speichert, kann der implementierte Logging-Mechanismus nicht übernommen werden, da er nur für flache Graphstrukturen ausgelegt ist. Geschachtelte Graphen erfordern die Erweiterung des Logging-Konzeptes um eine zweite Dimension. Versionsänderungen von Untergraphen werden im Vater-Graph protokolliert und erlauben es, die strikte Trennung der Versionsräume beizubehalten. Damit ist die flexible Versionierung auf beliebiger Graph-Ebene möglich. Eine dritte Logging-Dimension wird geschaffen, indem der Anwendung erlaubt wird, die Graphen-Logs mit spezifischen Informationen anzureichern. Dabei können nicht nur Kommentare eingefügt werden, sondern darüber hinaus eigene Kommandos definiert werden, die von der Versionierungserweiterung ausgewertet werden.

Durch die Versionierung graphübergreifender Verbindungen können Inkonsistenzen innerhalb des Graphpools entstehen. Versionsgruppen sind ein flexibles

Konstrukt zur Gruppierung von Graphen, zwischen denen Beziehungen bestehen. Das Versionieren auf Versionsgruppen-Ebene kann die Konsistenz der gruppierten Graphen gewährleisten.

Konfigurationen dienen zum Gruppieren von Graph-Versionen. Die Semantik einer Konfiguration ist dabei allein der Anwendung überlassen, wodurch ihre Einsatzmöglichkeiten vielfältig sind. Anwendungen, die in Konfigurationen konsistente Versions-Kombinationen speichern, können einen Mechanismus zum Ausführen der Konfigurationen nutzen. Dadurch werden die betroffenen Graphen in die gespeicherten Versionen versetzt.

Replikationsfähigkeit von Graphen ist für die Realisierung optimistischer Transaktionen wichtig, ohne die Mehrbenutzerfähigkeit nur über das Sperren von Dokumenten möglich ist. Der Logging-Mechanismus stellt alle Informationen zur Rekonstruktion von Graphen an beliebiger Stelle zur Verfügung. Mit der Replikation von Graphen schafft die Versionskontrolle zudem die Voraussetzungen für eine zukünftig zu entwickelnde Merge-Funktionalität.

5.2 Implementierung

Die im vorherigen Abschnitt vorgestellten Konzepte sind geeignet, um die in Kapitel 3 formulierten Anforderungen zu erfüllen. In diesem Abschnitt werden die wichtigsten Implementierungsdetails zur Realisierung der Konzepte aufgezeigt. Abschnitt 5.2.1 zeigt, wie die Integration der Erweiterung durch das Wrapper-Prinzip erfolgt. Mit der Integration der Logging-Funktionalität und der Implementierung von Kommandos beschäftigt sich Abschnitt 5.2.2. Abschnitt 5.2.3 führt erste Ansätze zur Optimierung von Kommando-Listen auf. Danach erläutert Abschnitt 5.2.4, wie Hilfsstrukturen im Graphpool gespeichert und vor der Anwendungsicht verborgen werden. Die nachfolgenden drei Abschnitte erläutern darauf aufbauend die Implementierung der drei wichtigsten Hilfsstrukturen. Dazu zählen Versionsräume (5.2.5), Versionsgruppen (5.2.6) und Konfigurationen (5.2.7). Anschließend wird in Abschnitt 5.2.8 auf die Replikation von Graphen eingegangen, bevor der letzte Abschnitt die öffentliche Schnittstelle der Versionserweiterung zusammenfasst.

5.2.1 Integration in DRAGOS

Die Integration der Versionskontrolle in DRAGOS erfolgt mit Hilfe des in Abschnitt 2.5.1 vorgestellten Wrapper-Konzepts. Die für die Einbindung der Er-

weiterung benötigten Informationen liefert die Erweiterungsbeschreibung, die im Graphpool registriert wird.

Die Erweiterungsbeschreibung für die Versionskontrolle besteht aus:

1. dem `VersionWrapper`
2. dem `VersionGraphPool`
3. dem `VersionGraphPoolChecker`

Aufgabe des `VersionWrapper`-Objekts ist es, beliebige Graphelemente und Elemente des Graphschemas durch Objekte zu kapseln, die sie um die Funktionalität der Versionierungsmechanismen erweitern. So wird jede Instanz einer Unterklasse von `GraphEntity` in einem Objekt der entsprechenden Unterklasse von `VersionGraphEntity` und jede Instanz von `GraphEntityClass` in einem `VersionGraphEntityClass`-Objekt gekapselt. Der in DRAGOS integrierte Wrapper-Mechanismus stellt die Anwendung von `VersionWrapper` auf alle vom Graphmodell zurückgelieferten Objekte sicher.

Durch Nutzung der Versionierungserweiterung wird zusätzlich der Wrapping-Mechanismus auf die Klassen `GraphPool` und `Schema` ausgeweitet. Durch die Registrierung der Erweiterungsbeschreibung wird DRAGOS veranlasst, der Anwendung einen in ein `VersionGraphPool`-Objekt gekapselten Graphpool zur Verfügung zu stellen. Als Schema liefert dieser wiederum ein gekapseltes `VersionSchema`-Objekt.

Die Versionierungserweiterung definiert darüber hinaus spezifische Konsistenzprüfungen in der Klasse `VersionGraphPoolChecker`. Die Notwendig dafür erschließt sich im nächsten Abschnitt.

5.2.2 Versionierung und Logging

Durch die Kapselung von Graphelementen in erweiterungsspezifische Objekte besteht die Möglichkeit, die Funktionalität zu erweitern oder zu verändern. Jede graphverändernde Operation wird durch einen Methodenaufruf auf einem Objekt des Graphmodells initiiert. So ist beispielsweise die Erzeugung eines Knotens das Ergebnis eines Aufrufs der Methode `createNode` auf einem Graph-Objekt. Ihre Funktionalität kann in der Implementierung von `VersionGraph` erweitert werden.

Auf diese Weise kann das Protokollieren von Graphveränderungen realisiert werden. Das Vorgehen dabei wird in Abbildung 5.12 deutlich, in der die

```
public Node createNode(NodeClass type) {

    /* Erzeugung des Knotens */
    Node newNode = getWrappedGraph().createNode(type);

    /* Logging: Protokollieren der Operation */
    getLog().logCreateNode(newNode);

    /* Ergebnis */
    return newNode;
}
```

Abbildung 5.12: Erweiterung einer Methode um Logging-Funktionen

`VersionGraph`-Implementierung von `createNode` vereinfacht dargestellt ist. Zunächst wird der Methodenaufruf an die darunterliegende Schicht weitergeleitet, also an das von der `VersionGraph`-Instanz gekapselte `Graph`-Objekt. Die Erzeugung des Knotens wird anschließend protokolliert, indem eine entsprechende Methode des Logs des Graphen aufgerufen wird.

In die Ausführung der Methode ist also ein zusätzlicher Methodenaufruf eingefügt worden. Die geloggten Kommandos sind jeweils als eigene Klassen implementiert. Alle Kommando-Klassen implementieren das `Command`-Interface, das im Wesentlichen nur die Methode `apply()` definiert. Diese Methode wird beim Abarbeiten des protokollierten Kommandos aufgerufen. Indem jedes Kommando so sein aktives Verhalten selbst festlegt, ist der Mechanismus durch anwendungsspezifisches Logging beliebig erweiterbar. Ein Logging-Vorgang erzeugt je ein Kommando-Objekte für die Vorwärts- und Rückwärts-Kommandoliste.

Für Kommandos, die graphverändernde beziehungsweise versionsverändernde Operationen repräsentieren, muss ein Aufruf der apply-Methode die gleiche Veränderung des Graphpools bewirken wie die Operation, die zur Protokollierung des Kommandos führt. Bei der Erzeugung jedes Kommandos werden deshalb alle Informationen gespeichert, die benötigt werden, um die betreffende Operation bei Bedarf wiederholen zu können. In Tabelle 5.2 ist zu jeder graph- und versionsverändernde Operation das entsprechende Kommando und die Menge der für dessen Erzeugung benötigten Parameter aufgelistet. Ein `ChgAttributeValue`-Kommando protokolliert demnach das Ändern eines Attributwertes und benötigt folgende Parameter:

1. GraphEntityID entity: Das Graphelement, dessen Attribut verändert wird.
2. AttributeID attribute: Das zu verändernde Attribut.

Operation	Kommando	Parameter
Erzeuge Graphelement	UnhideGraphEntity	GraphEntityID entity
Entferne Graphelement	HideGraphEntity	GraphEntityID entity
Ändere Attributwert	ChgAttributeValue	GraphEntityID entity AttributeID attribute Serializable newValue
Setze Attribut ungültig	SetAttributeInvalid	GraphEntityID entity AttributeID attribute
Ändere Meta-Attributwert	ChgMetaAttributeValue	GraphEntityID entity Serializable metaAttribute Serializable newValue
Setze Meta-Attribut ungesetzt	UnsetMetaAttribute	GraphEntityID entity Serializable metaAttribute
Ändere Quell-Verbindung einer Kante	ChgEdgeSource	EdgeID edge GraphEntityID newSource
Ändere Ziel-Verbindung einer Kante	ChgEdgeTarget	EdgeID edge GraphEntityID newTarget
Ändere Verbindung eines Relationsendes	ChgRelatedEntity	RelationEndID relationEnd GraphEntityID newRelatedEntity
Ändere Version	ChgVersion	GraphID graph Serializable versionID

Tabelle 5.2: Graphverändernde Kommandos

3. Serializable newValue: Der Wert, der dem Attribut zugeordnet wird.

Die Bedeutung der `ID`-Typen wird an späterer Stelle genauer erläutert. Sie stellen serialisierbare Bezeichner dar, über die die betreffenden Graphelemente oder Attribute referenziert werden.

Jede der in Abschnitt 5.1.2 identifizierten und in Tabelle 5.1 aufgelisteten Operationen findet ihre Entsprechung als Kommando und damit als `Command`-Klasse.

In Abschnitt 5.1.2 wurde bereits auf die Problematik mit der zum Entfernen eines Graphelements inversen Operation hingewiesen. Eine solche Operation muss für das Graphelement den Zustand wiederherstellen, in dem sich das Element vor dem Entfernen befand. Die DRAGOS-Implementierung umgeht diese Problematik, indem entfernte Graphelemente nicht aus dem Graphpool

gelöscht werden. Sie werden stattdessen nur aus dem Ergebnis aller Anwendungsanfragen herausgefiltert, sind also aus Anwendungssicht gelöscht. Das so erreichte Verbergen von Graphelementen erfolgt über das Setzen eines Meta-Attributs, was die Bezeichnung der Kommandos `HideGraphEntity` und `UnhideGraphEntity` erklärt. Beim Wiederherstellen eines gelöschten Graphelements wird das Meta-Attribut wieder entfernt. Der Zustand des Graphelements bleibt dabei unverändert, entspricht also dem dem Zustand vor der Entfernen-Operation.

Durch das Verbergen von Kanten und Relationsenden können Konsistenzprobleme in Form von Kardinalitätsverletzungen auftreten. Angenommen eine Kante, die Instanz einer Kantenklasse mit auf 1 beschränkter Quell- und Ziel-Kardinalität verbindet zwei Knoten und wird von der Anwendung gelöscht. Aus Anwendungssicht ist nun das Erzeugen einer neuen Kante zwischen den Knoten unproblematisch. Da die alte Kante jedoch intern noch existiert, meldet die Graphpool-Prüfung nach Ende der Transaktion eine Überschreitung der maximalen Kardinalität. DRAGOS sieht keine Möglichkeit vor, die standardmäßige Graphpool-Prüfung zu ersetzen. Sie kann jedoch um erweiterungsspezifische Prüfungen ergänzt werden. Das Problem wird gelöst, indem bei der Definition der Schemaklassen beschränkte Kardinalitäten abgefangen und in Meta-Attributen gespeichert werden. Für die Klassendefinition werden dann unbeschränkte Kardinalitäten benutzt, weshalb die standardmäßige Graphpool-Prüfung erfolgreich verläuft. Um aus Anwendungssicht auftretende Konsistenzverletzungen zu verhindern, wird eine zusätzliche Prüfung definiert, die nur die im aktuellen Zustand nicht verborgenen Graphelemente in Bezug auf die in den Meta-Attributen gespeicherten Kardinalitäten überprüft. Der `VersionGraphPoolChecker` wird als Teil der Erweiterungsbeschreibung im Graphpool registriert.

In Tabelle 5.2 sind die Kommandos für das anwendungsspezifische Logging nicht berücksichtigt. Zur Definition von Kommentaren werden die beiden in Tabelle 5.3 aufgeführten Kommandos bereitgestellten. Die Ausführung der Kommandos veranlasst den Ereignismanager ein `LoggingEvent` zu feuern. Dieses beinhaltet einen internen Bezeichner des Graphen, in dessen Versionsdelta sich der Kommentar befindet. Zusätzlich besitzt jedes Kommentar-Kommando einen serialisierbaren und im betrachteten Delta eindeutigen Kommentarbezeichner. Die für Gruppierungs-Markierungen zusätzlich aufgeführten Parameter vom Typ `boolean` definieren, ob der Kommentar den Beginn oder das Ende der Folge gruppierter Änderungen darstellt, und ob er sich im Vorwärts- oder Rückwärts-Delta befindet.

Kommentar	Kommando	Parameter
Einzel-Markierung	CommentMark	GraphID graph Serializable commentID
Gruppierungs-Markierung	CommentGroup	GraphID graph Serializable commentID boolean isStarting boolean isForward

Tabelle 5.3: Weitere Kommandos

5.2.3 Optimierung von Deltas

Der zusätzliche Speicherplatzbedarf, der durch Einsatz der Versionierungserweiterung entsteht, wird wesentlich durch die protokollierten Kommando-Listen verursacht, die in Graphen-Logs und Versionsdeltas gespeichert sind. Darüber hinaus ist die für eine Versionswiederherstellung benötigte Rechenzeit von der Länge der abzuarbeitenden Kommandolisten abhängig. Eine Optimierung dieser Kommandolisten ist deshalb wünschenswert. Da ein einzelner Versionsübergang die kleinste Einheit für Versionsänderungen darstellt, werden die Kommandolisten eines Versionsdeltas stets als Ganzes abgearbeitet. Zwischenzustände können von der Versionskontrolle nicht wiederhergestellt werden. Innerhalb eines Deltas können bestimmte Kommandos zusammengefasst werden, wenn dadurch das Ergebnis der Kommando-Abarbeitung in beide Richtungen nicht verändert wird. Auf diese Art können Redundanzen entfernt werden, ohne die Konsistenz des Graphpools zu beeinträchtigen.

Die aktuell implementierten Optimierungen beschränken sich auf Kommandos zum Ändern von Attributen und Meta-Attributen. Mehrfache Änderungen des gleichen Attributes innerhalb eines Deltas können durch ein einziges Kommando ausgedrückt werden. Der Optimierungsvorgang erfolgt schon während des Protokollierens neuer Kommandos im Log des Graphen. Gegenüber einer nachträglichen Optimierung bei der Versionserzeugung hat dies Vorteile, da die zur Optimierung benötigte Rechenzeit über die Laufzeit verteilt wird und nicht punktuell auftritt. Werden außerdem nur selten neue Versionen erzeugt, und werden zwischenzeitlich viele Änderungen protokolliert, sollte die Länge der Kommandoliste schon im Log optimiert sein.

Tabelle 5.4 zeigt das Schema, nach dem Attributsänderungen zusammengefasst werden. Die Optimierung von Änderungen an Meta-Attribut erfolgt analog. Die aufgeführten Änderungsbeschreibungen $X \rightarrow Y$ beziehen sich jeweils auf die gemeinsame Betrachtung von Vorwärts- und Rückwärts-Kommando, die

zu protokollierende Attributsänderung	letzte Attributsänderung im Log	resultierende Attributsänderung
Wert A → Wert B	Wert C → Wert A	Wert C → Wert B
	ungesetzt → Wert A	ungesetzt → Wert B
	Wert B → Wert A	KEINE
ungesetzt → Wert A	Wert B → ungesetzt	Wert B → Wert A
	Wert A → ungesetzt	KEINE
Wert A → ungesetzt	Wert B → Wert A	Wert B → ungesetzt
	ungesetzt → Wert A	KEINE

Tabelle 5.4: Optimierung von Attributsänderungen

zusammen ein Paar an einer bestimmten Listenposition bilden. Y ist dabei das Resultat des Vorwärts-Kommandos und X das des Rückwärts-Kommandos. Beispielsweise bedeutet $ungesetzt \rightarrow WertA$, dass an einer Listenposition das Vorwärts-Kommando `ChgAttributeValue` mit dem Rückwärts-Kommando `SetAttributeInvalid` kombiniert ist. Soll nun eine Operation protokolliert werden, die ein Attribut verändert, wird überprüft, ob das gleiche Attribut innerhalb des Logs schon einmal geändert wurde. Ein gefundenes Kommando-Paar wird dann gelöscht und das zu protokollierende Kommando-Paar entsprechend angepasst. Wird beispielsweise der Wert des Attributs von A auf B gesetzt und wurde er innerhalb des Logs zuvor von C auf A geändert, ändert das resultierende Kommando-Paar das Attribut von C auf B. Entspricht das Rückwärts-Kommando des gefunden Paars dem Vorwärts-Kommando des zu protokollierenden Paars, heben sich beide Paare auf und es wird keine Operation protokolliert.

Optimierungen, die sich auf andere Kommandos beziehen, sind ebenfalls denkbar. So könnte beispielsweise das wiederholte Ändern einer Verbindungsreferenz zu einer Operation zusammengefasst werden. Dabei können jedoch Kardinalitäts-Probleme auftreten, die der Optimierungs-Mechanismus berücksichtigen muss. Auch die Optimierung von Kommandos zum Erzeugen und Entfernen eines Graphelements ist problematisch. Solche Kommandos dürfen nicht entfernt werden, wenn zwischenzeitlich Verbindungs-Referenzen zu dem Graphelement existieren. Zukünftig sind komplexere Optimierungsmechanismen denkbar, deren Realisierung jedoch den Umfang dieser Arbeit übersteigt.

5.2.4 Hilfsstrukturen

Die Implementierung der in Abschnitt 5.1 vorgestellten Konzepte erfordert die persistente Speicherung von Zusatzinformationen für die Versionskontrolle. DRAGOS bietet zu diesem Zweck die Nutzung von Meta-Attributen an. In ihnen können beliebig komplexe Objekte gespeichert werden. Die einzige Voraussetzung, die zur Speicherbarkeit solcher Objekte erfüllt sein muss, ist ihre Serialisierbarkeit. Dabei ist jedoch zu beachten, dass der Vorgang der Serialisierung mit steigender Komplexität der Objekte sehr viel Zeit in Anspruch nimmt. Die Nutzung von Meta-Attributen sollte deshalb beschränkt sein auf die Speicherung von Objekten mit geringer Komplexität.

Für die Versionskontrolle ist die Verwaltung komplexer Strukturen, wie beispielsweise Versionsräume und Deltas, notwendig. Diese können mit zunehmender Nutzung der Versionierungsfunktionen einen Umfang erreichen, der die Speicherung in Meta-Attributen ineffizient macht. Selbst wenn keine Versionen erzeugt werden, wachsen die im Log eines Graphen protokollierten Kommandolisten mit jeder getätigten Graphveränderung.

Als Lösungsmöglichkeit der Problematik bietet sich an, komplexe Strukturen der Versionierungserweiterung in einem externen Datenspeicher zu speichern, und in Meta-Attributen nur Referenzen auf die externen Objekte abzulegen. Da die Nutzung eines zweiten Datenbanksystems nicht gewünscht ist, ist die Verwendung der von DRAGOS ohnehin genutzten Datenbank sinnvoll. Dazu müsste in der Implementierung jeder Datenbank-Anbindung eine Schnittstelle für die Versionierung realisiert werden, was aufwändig und unflexibel ist.

Die realisierte Lösung besteht darin, die Hilfsstrukturen in Form von Graphen zu modellieren und das DRAGOS-Graphmodell für ihre Speicherung zu verwenden. Dabei sollen die Hilfsstrukturen nicht von der Anwendung verändert werden dürfen — idealerweise sind sie aus Anwendungssicht nicht existent. Dies wird durch Filterfunktionen realisiert, die aus dem Ergebnis des Aufrufes einer Graphpool-Methode die Graphen herausfiltern, die nur der Erweiterung intern zur Verfügung stehen sollen. Die Identifizierung der Graphen erfolgt über die Verwendung eines Namenspräfixes, das für die Versionierungserweiterung reserviert ist. Auch die zur Modellierung der Hilfsgraphen erzeugten Schemaklassen müssen aus Schemaanfragen herausgefiltert werden. Zur Verbergung der Hilfsstrukturen ist es demnach ausreichend, erweiterte Graphpool- und Schema-Klassen zur Verfügung zu stellen, die die Filterung realisieren. Da diese Funktionalität universell auch von zukünftigen Erweiterungen genutzt werden kann, stehen die Klassen `FilteringGraphPool` und `FilteringSchema` standardmäßig im DRAGOS-Kernel zur Verfügung.

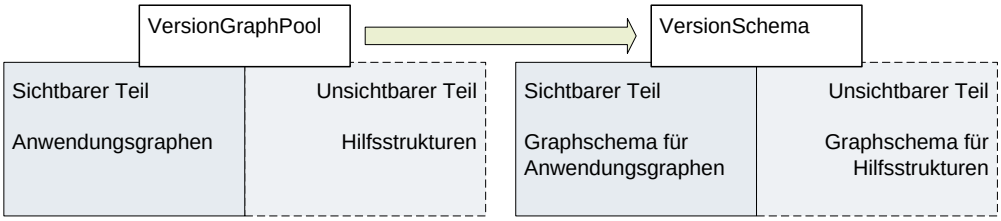

Abbildung 5.13: GraphPool und Schema der Versionierungserweiterung

Da die Versionskontrollerweiterung über die Filterung hinausgehende Funktionalität in Graphpool und Schema benötigt, definiert sie eine eigene `VersionGraphPool`-Klasse, die von `FilteringGraphPool` erbt. Durch die Registrierung der Erweiterungsbeschreibung wird DRAGOS veranlasst, der Anwendung einen in `VersionGraphPool` gekapselten Graphpool zur Verfügung zu stellen. Als Schema liefert dieser wiederum ein Objekt der Klasse `VersionSchema`, die von `FilteringSchema` erbt. Durch Filterfunktionen wird die Definition von Schemaklassen und Hilfsgraphen möglich, die vor der Anwendungssicht verborgen sind. Die Unterteilung des Graphpools und des Schemas in sichtbare und unsichtbare Anteile ist in Abbildung 5.13 dargestellt.

Beim erstmaligen Öffnen eines versionierten Graphpools sorgt ein Initialisierungsprozess für die Definition aller Schemaklassen, die für die Erzeugung der Hilfsstrukturen benötigt werden.

An verschiedenen Stellen ist es notwendig, Referenzen auf Graphelemente in Attributen oder Meta-Attributen zu speichern. Dies ist aufgrund der Serialisierung bei der Ablage in der Datenbank nicht direkt möglich. Jedes Graphelement besitzt aber einen eindeutigen serialisierbaren Bezeichner, der nur DRAGOS-intern verwendet wird, um Graphelemente zu unterscheiden. Diese Bezeichner können auch von Erweiterungen genutzt werden. Um Typsicherheit zu erreichen, existieren typisierte *Graphelement-Identifizierer* (Graphelement-IDs), welche die serialisierbare Referenzierung von Graphelementen und Elementen des Graphschemas ermöglichen. Zu jeder Unterklasse `Xyz` von `GraphEntity` existiert eine entsprechende `XyzID`-Klasse, die Unterklasse von `GraphEntityID` ist. Das gleiche Verhältnis besteht zwischen den Unterklassen von `GraphEntityClass` und den entsprechenden Unterklassen von `GraphEntityClassID`. Über eine Methode der ID-Objekte kann die Anwendung direkt an das repräsentierte Graphelement gelangen, sowie an bestimmte zusätzliche Informationen. So wird beispielsweise die ID der implementierten Schemaklasse in der ID des Graphelements gespeichert, um ohne Zugriff auf das eigentliche Objekt zur Verfügung zu stehen.

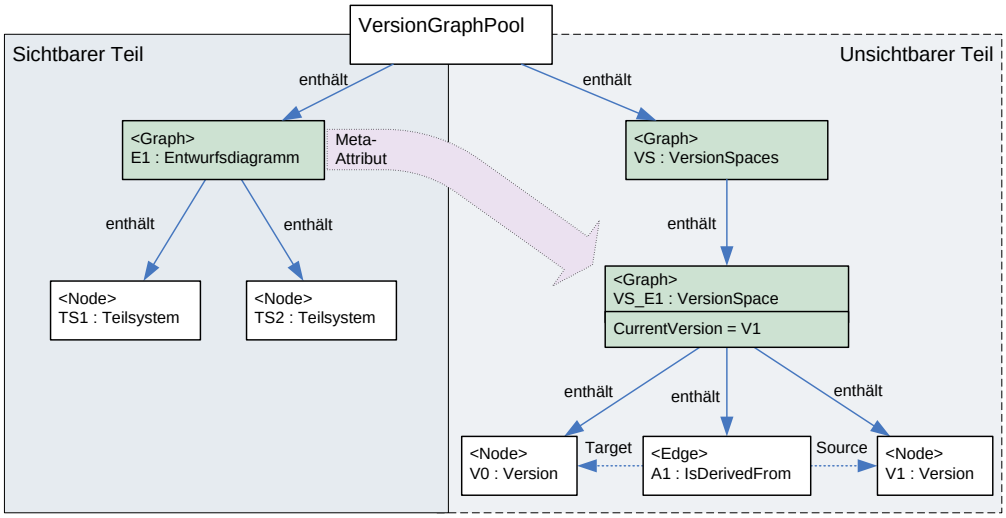

Abbildung 5.14: Realisierung von Versionsräumen

5.2.5 Versionsraum

Jedem von der Anwendung erzeugten Graph wird bei seiner Erzeugung ein Versionsraum zugeordnet. Ein Versionsraum verwaltet eine Menge von Versionen, die einen Ableitungs-Baum aufspannen. Ein Versionsraum kann also als Graph aufgefasst werden. In ihm werden Versionen durch Knoten dargestellt, zwischen denen Kanten existieren, die die Ableitungs-Beziehung repräsentieren. Diese Struktur kann im DRAGOS-Schema modelliert werden durch die Definition folgender Schemaklassen:

1. `VersionSpace` : Graphklasse für Versionsräume
2. `Version` : Knotenklasse für Versionen
3. `IsDerivedFrom` : Kantenklasse für Ableitungs-Beziehungen

Abbildung 5.14 zeigt die Implementierung von Versionsräumen in Form von Graphen. Alle Graphen, die Versionsräume repräsentieren, werden in einem Graphen oberster Graphpool-Ebene angelegt, der vor der Anwendungssicht verborgen ist. Die Beziehung zwischen dem von der Anwendung erzeugten Graph `E1` und den für ihn verwalteten Versionsraum `VS_E1` wird in Form einer in einem Meta-Attribut gespeicherten Referenz realisiert. Die Identifizierung der aktuellen Version erfolgt durch ein in der Graphklasse definiertes Attribut, das die Referenz auf den entsprechenden Knoten beinhaltet.

Die Schnittstelle, die intern zur Arbeit mit dem Versionsraum verwendet wird, ist durch die Java-Klasse `VersionSpace` definiert, die einen Graphen der

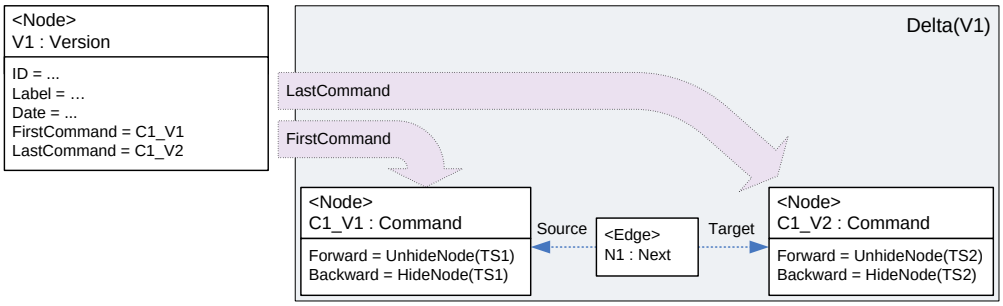

Abbildung 5.15: Realisierung von Versionsdeltas

gleichnamigen Graphklasse kapselt. Versionsraumänderungen bestehen aus einer Abfolge mehrerer atomarer Graphveränderungen. So sorgt beispielsweise die Methode `VersionSpace.createVersion()` für das Erzeugen eines neuen Knotens vom Typ `Version` und einer neuen Kante vom Typ `IsDerivedFrom`. Letztere verbindet den neuen Knoten mit dem Knoten, der die aktuelle Version repräsentiert. Zuletzt wird das Attribut `currentVersion` gesetzt und so die aktuelle Version geändert. Als Wert des Attributs wird ein serialisierbares ID-Objekt verwendet, das den neuen Knoten referenziert. Auch Knoten, die Versionen repräsentieren, werden intern gekapselt. Die Java-Klasse `Version` besitzt zudem eine öffentliche Schnittstelle, die der Anwendung den Umgang mit Versionen ermöglicht. So kann diese beispielsweise an die für jede Version gespeicherten Delta-Informationen gelangen.

Die Speicherung der Kommando-Listen, die das Delta einer Version darstellen, erfolgt ebenfalls mit Hilfe des DRAGOS-Graphmodells, und ist in Abbildung 5.15 veranschaulicht. Da die Listen sehr lang werden können, ist die Speicherung in einem Attribut des Versionsknotens mit Effizienznachteilen verbunden. Stattdessen erzeugt jedes protokollierte Kommando einen Knoten der Knotenklasse `Command`, der in zwei Attributen das protokollierte sowie das inverse Kommando speichert. Die Kommandos sind durch `next`-Kanten verkettet. Der Versionsknoten speichert nur eine Referenz auf das erste und das letzte Glied.

Da die Versionierung von Versionsräumen nicht sinnvoll ist, sollten Änderungen an den Hilfsgraphen nicht protokolliert werden. Die Versionierungserweiterung unterscheidet deshalb zwischen versionierten und unversionierten Graphklassen. Alle von der Anwendung definierten Graphklassen sind versioniert. Für einen Graphen, der eine unversionierte Graphklasse instanziiert, wird kein Versionsraum angelegt. Zudem werden für ihn wie auch für alle in ihm enthaltenen Graphelemente keine Logging-Vorgänge ausgeführt.

Im Versionsraum jedes versionierten Graphen existiert eine Version, die intern das Log des Graphen repräsentiert. Diese *Log-Version* ist vor der Anwendung verborgen. Alle protokollierten Kommandos werden im Delta dieser Version gespeichert. Die Log-Version ist stets von der aktuellen Version abgeleitet. Nach einer Versionswiederherstellung wird das Delta der Log-Version gelöscht und ihre Vorgänger-Beziehung auf die neue aktuelle Version umgeleitet. Wenn die Anwendung das Erzeugen einer neuen Version initiiert, wird die Log-Version zur aktuellen Version. Anschließend wird eine neue Log-Version angelegt. Durch diese Art der Implementierung kann auf ein separates Konstrukt zur Verwaltung von Graphen-Logs verzichtet werden.

5.2.6 Versionsgruppen

Versionsgruppen enthalten eine Menge von versionierten Graphen, die dynamisch vergrößert und verkleinert werden kann. Für die Modellierung dieser Menge bietet sich ein Graph an, in dem für jedes Gruppenmitglied ein Knoten existiert. Demnach werden folgende Schemaklassen definiert:

1. `VersionGroup` : Graphklasse für Versionsgruppen
2. `VersionGroupMember` : Knotenklasse für Versionsgruppenmitglieder

Ähnlich wie im vorhergehenden Abschnitt beschrieben existiert eine Java-Klasse namens `VersionGroup`, deren Objekte Graphen kapseln. Über ein `VersionGroup`-Objekt kann die Anwendung neue Mitglieder in die Gruppe aufnehmen. Dabei wird ein neuer Knoten angelegt, der in seinem einzigen Attribut eine Referenz auf den in die Gruppe aufgenommenen Anwendungsgraphen speichert, was in Abbildung 5.16 dargestellt ist.

Die Nutzung von Graphen zur Implementierung von Versionsgruppen hat einen weiteren Vorteil, der sich auf die Versionierbarkeit von Versionsgruppen bezieht. Besitzt der von der Versionsgruppe gekapselte Graph einen Versionsraum, lässt sich dieser als Versionsraum der Versionsgruppe nutzen. Indem die Graphklasse `VersionGroup` als versionierbar definiert wird, werden Versionsgruppen intern wie zusätzliche Obergraphen der Mitgliedsgraphen behandelt. Die Funktionalität zur Erzeugung und Wiederherstellung von Versionen ist für Versionsgruppen also durch den versionierten Graphen gegeben. Es ist lediglich eine Anpassung des Logging-Mechanismus erforderlich. Dieser muss alle Versionsänderungen eines Graphen nicht nur in dessen Vater-Graphen protokollieren, sondern zusätzlich in allen Graphen, die die Versionsgruppen des

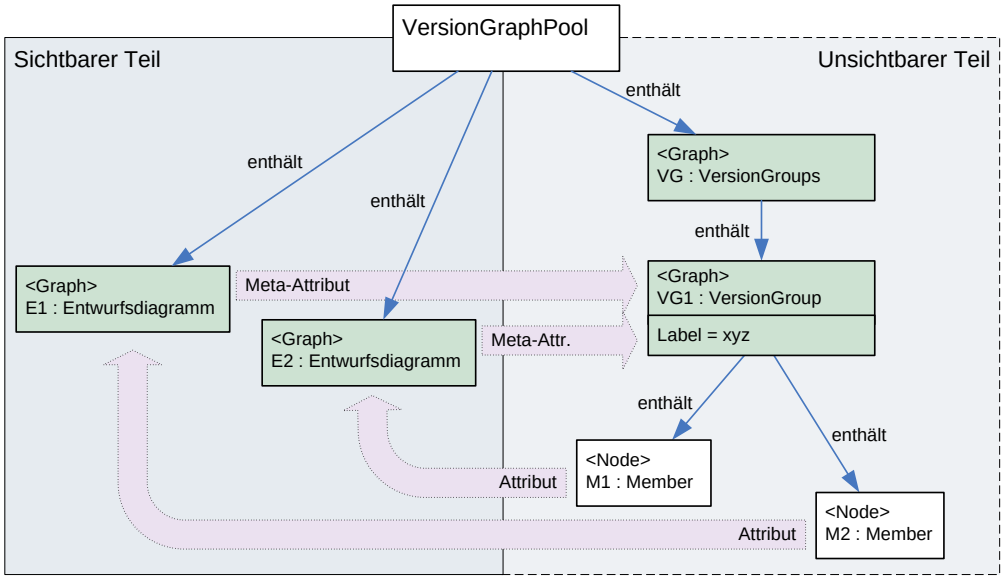

Abbildung 5.16: Realisierung von Versionsgruppen

Graphen darstellen. Dazu ist es notwendig, dass ein Anwendungsgraph Referenzen auf alle Versionsgruppen verwaltet, in denen er Mitglied ist.

Wie in Abschnitt 5.1.3 erläutert wurde, ist die Komposition der Mitglieder Teil des versionierten Zustands der Versionsgruppe. Dies wird durch die beschriebene Implementierungsart dadurch gewährleistet, dass das Hinzufügen und Entfernen von Gruppenmitgliedern dem Erzeugen und Löschen von Knoten im Graph der Versionsgruppe entspricht. Diese Graphveränderungen werden vom Logging-Mechanismus erfasst.

Zur Verwaltung aller definierten Versionsgruppen steht der Anwendung der Versionsgruppen-Manager zur Verfügung. Der Versionsgruppen-Manager ist für das Anlegen und Löschen von Versionsgruppen verantwortlich. Er greift auf den Graphen `VersionGroups` zu, der den gemeinsamen Obergraphen für alle Graphen darstellt, die Versionsgruppen repräsentieren.

5.2.7 Konfigurationen

Die Implementierung von Konfigurationen ähnelt der im vorherigen Abschnitt erläuterten Realisierung von Versionsgruppen. Es werden dazu zwei Schemaklassen definiert:

1. `Configuration` : Graphklasse für Konfigurationen

2. `ConfigurationMember` : Knotenklasse für Konfigurationsmitglieder

Konfigurationsmitglieder sind Knoten, die auf die in einer Konfiguration enthaltenen Versionen von Graphen verweisen. Sie enthalten also neben der Referenz auf den betroffenen Graphen ein zweites Attribut `VersionID`, das den Bezeichner der Version speichert. Dieses Attribut wird verändert, wenn eine andere Version des selben Graphen aufgenommen wird.

Konfigurationen sind versionierbar, was durch das Versionieren der Graphen vom Typ `Configuration` erreicht wird. Das Hinzufügen und Löschen von Konfigurationsmitgliedern wird folglich genauso protokolliert wie Versions-Ersetzungen, die durch Änderungen am Wert des `VersionID`-Attributs erfasst sind. Somit ist gewährleistet, dass die Versionierung von Konfigurationen alle Merkmale des in Abschnitt 5.1.4 definierten versionierten Zustand speichert.

Alle Graphen, die Konfigurationen repräsentieren, werden in einem gemeinsamen Obergraphen erzeugt. Das Anlegen und Löschen von Konfigurationen erfolgt über einen Konfigurationen-Manager, der analog zum Versionsgruppen-Manager implementiert ist.

5.2.8 Replikation von Graphen

Bei der Implementierung des in Abschnitt 5.1.5 beschriebene Algorithmus zur Replikation eines Graphen sind einige Besonderheiten zu beachten. Die für den Original-Graphen protokollierten Kommandos enthalten Graphelement-Referenzen, die sich auf Graphelemente im Original-Graphen beziehen. Wurde als erste graphverändernde Operation beispielsweise ein Attributwert gesetzt, würde das Ausführen des protokollierten `ChgAttributeValue`-Kommandos das Attribut des Original-Graphen verändern. Um dies zu verhindern, muss eine Übersetzung aller gespeicherten Graphelement-Referenzen auf die neuen Graphelemente erfolgen.

Die dazu notwendige bijektive Abbildung wird durch die protokollierten `UnhideGraphEntity`-Kommandos konstruiert. Wird ein solches Kommando abgearbeitet, existiert im neuen Graphen die Kopie des betreffenden Graphelements noch nicht und kann deshalb nicht sichtbar gemacht werden. Das Abarbeiten eines `UnhideGraphEntity`-Kommandos muss also das Erzeugen der Graphelement-Kopie initiieren. Alle nachfolgenden Kommandos, die sich auf das Original-Element beziehen, werden dann auf das Kopie-Element bezogen.

Durch den Logging-Mechanismus werden alle bei der Replikation ausgeführten graphverändernden Operationen protokolliert, wodurch gewährleistet ist, dass auch der neue Graph versioniert werden kann.

Für die Nutzung der Replikation für Anwendungen wie Mehrbenutzerfunktionalität reicht die beschriebene Funktionalität nicht aus. Das Synchronisieren einer bestehenden Arbeitskopie mit einem Server erfodert das Replizieren einzelner Änderungen in einem bereits existierenden Zielgraphen. Mit der entsprechenden Erweiterung des beschriebenen Replikations-Mechanismus müssen sich zukünftige Arbeiten beschäftigen. Aus diesem Grund ist auch der in Abschnitt 5.1.6 beschriebene Merge-Mechanismen noch nicht realisiert worden.

5.2.9 Schnittstelle

Die in den vorherigen Abschnitten vorgestellten Klassen, die Versionen, Deltas, Versionsgruppen und Konfigurationen repräsentieren, gehören zur öffentlichen Schnittstelle der Versionierungserweiterung und können folglich von der Anwendung genutzt werden. Alle Methoden, die ein versioniertes Objekt bereitstellt, sind in der Schnittstelle `Versionable` definiert, die in Abbildung 5.17 dargestellt ist. Sie wird von den Klassen `VersionGraph`, `VersionGroup` und `Configuration` implementiert.

Die von `Versionable` definierten Funktionen lassen sich in vier Gruppen einteilen:

1. **Versionierung**: Methoden zum Erzeugen und Wiederherstellen von Versionen. Nach den in Kapitel 3 formulierten Anforderungen an die Versionierungsstrategie soll die Anwendung die Versionierung nach eigenen Bedürfnissen initiieren. Mit `removeVersion` wird eine Methode zur Verfügung gestellt, mit der Versionen gelöscht werden können. Dabei wird die gegebene Version und alle ihre Nachfolgeversionen aus dem Versionsraum entfernt. Die Anwendung kann diese Funktionalität nutzen, um durch die Versionierung genutzten Speicherplatz freizugeben. Dabei muss jedoch durch die Anwendungssemantik sichergestellt sein, dass keine Inkonsistenzen auftreten. Die Versionierungserweiterung liefert diesbezüglich keine Unterstützung.

2. **Logging**: Methoden die sich auf ungespeicherte Änderungen beziehen. Als lokale Änderungen werden solche Kommandos angesehen, die sich im Log des betrachteten Graphen befinden. Die Methoden `hasChanged` und `undoChanges` beziehen hingegen die Logs aller Untergraphen mit

```
public interface Versionable {

 /* Versionierung */
 Version createNewVersion(Serializable label);
 Version getCurrentVersion();
 void selectCurrentVersion(Serializable name);
 void selectCurrentVersion(Version version);
 boolean hasVersion(Version version);
 void removeVersion(Version version);

 /* Logging */
 List<Command> getLocalChanges();
 boolean hasChangedLocally();
 boolean hasChanged();
 void undoChanges();

 /* Versionsraum */
 List<Command> getLocalChangesToVersion(Version version);

 /* Anwendungsspezifisches Logging */
 void logCommentMark(Serializable markID);
 void logCommentChangeStart(Serializable changeID);
 void logCommentChangeEnd();
 void logCustomCommand(Command forward, Command backward);
}
```

Abbildung 5.17: Die Schnittstelle Versionable

ein. Sie betreffen also den gesamten versionierten Zustand des betrachteten Graphen.

3. **Versionsraum**: Methode, die alle Änderungen entlang des Pfades vom aktuellen Zustand des Graphen zur gegebenen Version zurückliefert.
4. **Anwendungsspezifisches Logging**: Methoden, über die eine Anwendung Kommentare und selbst definierte Kommandos ins Log aufnehmen kann.

5.2.10 Zusammenfassung

Durch die Nutzung des Wrapping-Mechanismus ist die transparente Integration von Logging-Funktionalität möglich. Die Anwendung kann dadurch je-

derzeit auf die Versionierungsfunktionen zugreifen, sie jedoch gleichermaßen ignorieren, wenn sie nicht benötigt werden. Darüber hinaus ermöglicht das Wrapper-Konzept die Realisierung des Logging-Mechanismus mit geringem Aufwand.

Die Modellierung aller Hilfsstrukturen als Graphen ist notwendig, um auf den Einsatz externer Datenspeicher verzichten zu können. Indem die Funktionalität des Graphmodells ausgenutzt wird, ist außerdem die Implementierung von Versionsräume, Deltas, Versionsgruppen und Konfigurationen erheblich vereinfacht. Der Filter-Mechanismus zum Verbergen von Hilfsstrukturen wird für das Verbergen von gelöschten Graphelementen aufgegriffen. So wird ein simples und effizientes Verfahren zur Konservierung ihres Zustands geschaffen.

Dadurch, dass versionierte Graphen für die Implementierung von Versionsgruppen und Konfigurationen genutzt werden, wird Redundanz bei der Implementierung dreier versionierbarer Konstrukte vermieden. Die erforderliche Erweiterung des Logging-Mechanismus ist minimal. Der Anwendung wird eine Schnittstelle geboten, die sich nahtlos in das Graphmodell einfügt, und mit der alle Funktionen der Versionskontrolle intuitiv zugänglich sind.

5.3 Entwicklungsstand

Die Implementierung aller vorgestellten Konzepte ist zum Zeitpunkt des Fertigstellens dieser Arbeit weitestgehend abgeschlossen. Dabei ist anzumerken, dass sich die Funktionen zur Replikation von Graphen erst in einem frühen Stadium befinden. Während die Replikation eines Graphen von der Ursprungsversion bereits funktioniert, kann die Replikation einzelner Änderungen in einem existierenden Zielgraphen von der Implementierung nicht geleistet werden. Die Erweiterung des Konzeptes ist nicht trivial zu bewerkstelligen und übersteigt den Umfang dieser Arbeit, die diesbezüglich deshalb nur einen Ausgangspunkt für zukünftige Erweiterungen darstellen kann.

Die Funktionsfähigkeit und Anwendungstauglichkeit aller anderen Konzepte ist hingegen bereits bewiesen. Es existiert eine umfangreiche Test-Suite, die die Funktionsweise der Versionierung in zahlreichen Tests bestätigt. Dabei werden neben der Ausführung von Basis-Funktionen auch komplexere Anwendungssituationen simuliert. Sichergestellt wird zudem, dass die sehr umfangreichen Tests der DRAGOS-Standard-Testsuite auch bei eingebundender Versionierungserweiterung erfolgreich verlaufen. Da darin alle Basis-Funktionen

des Graphmodells und des Schemas getestet werden, kann dies als Beleg für die nahtlose Integrationsfähigkeit der Erweiterung gewertet werden.

Eine erste Anwendung findet die Versionierungserweiterung in der Implementierung der PROGRES Codegenerierung (PROGRES Graph Code, PGC) [Kle93] für DRAGOS. In einer PROGRES-Spezifikation werden Graphersetzungsregeln verwendet, um die Funktionalität eines Werkzeugs zu spezifizieren. Die Ausführung einer Graphersetzungsregel verläuft nach einem nicht-deterministischen Mechanismus. Dieser sucht in der Graphstruktur eine Stelle, an der die Ausführung der Regel eine erfolgreiche Transformation erzeugt. Scheitert die Ausführung, wird mittels Backtracking der letzte Zustand wiederhergestellt, von dem aus in einen potentiell erfolgreichen Ausführungspfad verzweigt werden kann. Dieser Backtracking-Mechanismus kann mit Hilfe von Versionskontrolle realisiert werden. Durch die Versionierungserweiterung für DRAGOS wurde es möglich, die Codegenerierung so anzupassen, dass generierte Werkzeuge unabhängig von GRAS3 lauffähig sind. Indem eine DRAGOS-Anbindung die GRAS-Anbindung ersetzt, wird eine höhere Plattformunabhängigkeit der Werkzeuge erreicht.

6 Literaturvergleich und Ausblick

Die im vorhergehenden Kapitel beschriebene Realisierung der Versionskontrolle für DRAGOS greift Konzepte auf, die auch in GRAS3 und CoObRA umgesetzt sind. Der folgende Abschnitt 6.1.1 liefert einen abschließenden Vergleich der Systeme bezüglich der angebotenen Versionierungsfunktionen. Anschließend werden die wichtigsten Ergebnisse dieser Arbeit in Abschnitt 6.2 zusammengefasst. Wie diese Ergebnisse in zukünftigen Arbeiten für die Realisierung zusätzlicher Funktionalität genutzt werden können, zeigt der Ausblick in Abschnitt 6.3.

6.1 Literaturvergleich

Während Kapitel 4 GRAS3 und CoObRA auf Basis der realisierten Versionsmodelle mit den Anforderungen von DRAGOS verglichen hat, stellt dieser Abschnitt die fertige Realisierung den Vergleichssystemen gegenüber. Zunächst beschäftigt sich Abschnitt 6.1.1 mit dem Vergleich zu GRAS3, bevor im darauf folgenden Abschnitt 6.1.2 CoObRA betrachtet wird.

6.1.1 Vergleich mit GRAS3

Tabelle 6.1 fasst die wichtigsten Unterschiede zwischen den in GRAS3 und DRAGOS realisierten Funktionen zur Versionskontrolle zusammen. Ein grundsätzlicher Unterschied ergibt sich durch die unterschiedliche Wahl der versionierten Objekte. Während in GRAS3 nur Graphen auf oberster Ebene versioniert werden, besitzt in DRAGOS jeder Graph einen eigenen Versionsraum. Die dadurch erreichte höhere Flexibilität bietet sich der Anwendung schon bei der Modellierung ihrer Daten. Bei GRAS3 muss je nach Anwendung darauf geachtet werden, dass getrennt zu bearbeitende Daten in getrennten Graphen modelliert werden, um separate Undo/Redo-Funktionalität bereitzustellen. In DRAGOS kann die Versionierung hingegen ausser Acht gelassen werden, da getrennte Versionsräume auf allen Ebenen der Graph-Hierarchie existieren.

	GRAS3	**DRAGOS**
Versionierte Objekte	Graphen auf oberster Ebene	Alle Graphen
Beziehungen zwischen versionierten Objekten	Log-Gruppen zur Verwaltung graphübergreifender Kanten	• Versionierung von gemeinsamen Obergraphen • Versionsgruppen
Undo/Redo	unterstützt	durch Versionierung realisierbar
Konfigurationen	nicht unterstützt	unterstützt, versionierbar
Checkin/Checkout	unterstützt	durch Versionierung realisierbar
Anwendungsspezifische Informationen im Versionsraum	nicht unterstützt	• Versionsbezeichner • Kommentare und Definition spezifischer Kommandos

Tabelle 6.1: Unterschiede zwischen GRAS3 und DRAGOS

Durch das entsprechend angepasste Logging-Konzept wird sichergestellt, dass bei der Versionierung eines Graphen der Zustand der Untergraphen mit erfasst wird. Wird demnach nur auf oberster Ebene versioniert, so ist wie bei GRAS3 die Konsistenz aller graphübergreifender Verbindungen zwischen den Untergraphen gewährleistet. Der Vorteil von DRAGOS ist, dass die Wahl der Versionierungsebene die Versionierbarkeit der tieferliegenden Graphen nicht einschränkt. So ist es möglich, auf hoher Ebene Versionen von ganzen Dokumenten zu speichern, und gleichzeitig auf niedrigerer Ebene einzelne Dokumentteile zu versionieren.

Existieren graphübergreifende Beziehungen, hat die Anwendung für die Konsistenz der beteiligten Graphen zu sorgen. In GRAS3 existieren Grenzen zwischen versionierten Zuständen nur auf oberster Graphebene. Durch den Einsatz von Log-Gruppen kann der versionierte Zustand auf mehrere Graphen ausgedehnt und so die Konsistenz aller graphübergreifenden Kanten sichergestellt werden. Zu diesem Zweck realisieren Log-Gruppen ein gemeinsames Log für alle enthaltenen Graphen. Ein neues Mitglied kann nur dann in eine Gruppe aufgenommen werden, wenn sein Log leer ist. Durch die Flexibilität, die DRAGOS bezüglich der Versionierungsebene bietet, entstehen an beliebiger Stelle Übergänge zwischen versionierten Zuständen, die konsistent zu halten

sind. Versionsgruppen sind an diese dynamischen Verhältnisse angepasst und erlauben flexibles Hinzufügen und Entfernen von Mitgliedsgraphen, ohne dass deren Log-Informationen verändert werden oder verloren gehen. Die Bedingung, dass ein Graph nur aufgenommen werden kann, wenn keiner seiner direkten oder indirekten Ober- oder Untergraphen bereits Mitglied der Versionsgruppe ist, kann durch die in Abschnitt 5.1.3 vorgestellten Verfahren jederzeit erfüllt werden. Auch können Graphen gleichzeitig Mitglied in mehreren Versionsgruppen sein. Versionsgruppen sind für die langfristige Versionskontrolle geeignet, da sie intern wie Obergraphen ihrer Mitglieder behandelt werden. Im Unterschied zu Obergraphen stellen sie jedoch eine flexible Versionierungsebene dar, die dynamisch erzeugt und verändert werden kann.

Undo/Redo-Funktionalität ist in GRAS3 direkt nutzbar, da alle graphverändernden Kommandos in einem Log gespeichert sind. In DRAGOS werden graphverändernde Operationen hingegen nur im Log des Vatergraphen protokolliert. Die für ein Undo/Redo notwendigen Informationen können also auf die Logs mehrerer Untergraphen verteilt sein. Eine Undo/Redo-Erweiterung für DRAGOS kann aber basierend auf der Versionierungserweiterung implementiert werden. Im Ausblick, den Abschnitt 6.3 liefert, wird darauf detailierter eingegangen.

Ein Konstrukt zur Verwaltung von Konfigurationen bietet GRAS3 nicht an. Insbesondere in Bezug auf die langfristige Versionsverwaltung ist das Speichern konsistenter Kombinationen von Dokument-Versionen jedoch eine wichtige Funktionalität. Die in DRAGOS realisierten Konfigurationen legen der Anwendung keinerlei Einschränkungen bei der Definition von Versionskonstellationen auf. Zusätzlich wird mit der *apply*-Funktion ein Mechanismus zur Verfügung gestellt, mit dem der Gesamtzustand einer Konfiguration im Graphpool hergestellt werden kann.

Die Mehrbenutzerfähigkeit ist in GRAS3 durch die Client/Server-Verteilung gegeben. Clients beziehen ihre Daten mittels Check-Out von einem Server und übertragen die lokalen Änderungen durch Ausführung einer Check-In-Operation. In DRAGOS ist diese Art der Verteilung bislang nicht realisiert. Durch Konzepte wie Graph-Replikation und Zusammenführung von Versionen kann jedoch eine zukünftige Realisierung auf Basis der Versionskontrolle geschaffen werden.

GRAS3 bietet der Anwendung keine Möglichkeiten, den Versionierungsprozess anwendungsspezifisch zu erweitern. Versionen werden in Form von Checkpoints nur anhand einer Nummerierung unterschieden. DRAGOS hingegen unterstüzt beliebige serialisierbare Objekte als Versionsbezeichner. Darüber hinaus kann die Anwendung den Logging-Prozess durch das Einfügen von

Kommentaren oder eigenen Kommandos erweitern. Dies kann zukünftig neue Applikationsmöglichkeiten eröffnen.

Zusammenfassung Als Nachfolger von GRAS3 soll DRAGOS dessen Einsatzmöglichkeiten nicht einschränken sondern erweitern. Dies impliziert, dass die realisierte Versionskontrolle alle Funktionalität bietet, die auch durch die Änderungsverwaltung von GRAS3 zur Verfügung steht. Alle Funktionen, die nicht direkt im Umfang diese Arbeit implementiert wurden, können später auf Basis der realisierten Konzepte umgesetzt werden. Durch die Versionierung von Graphen auf beliebiger Ebene wird zudem eine Flexibilität erreicht, die neue Anwendungsmöglichkeiten für die Versionskontrolle erschließt.

6.1.2 Vergleich mit CoObRA

CoObRA verfolgt einen allgemeinen Ansatz, der Persistenz, Mehrbenutzerfähigkeit und Undo/Redo für beliebige Objektstrukturen zur Verfügung stellt. Die Ziele bei der Entwicklung von CoObRA unterscheiden sich damit wesentlich von denen dieser Arbeit, Versionskontrolle für ein bestimmtes Datenmodell zu realisieren. Dieser grundsätzliche Unterschied ist beim Vergleich der Systeme zu beachten. Tabelle 6.2 stellt die wichtigsten abweichenden Merkmale gegenüber.

Da die verwalteten Objektstrukturen vom Anwendungsfall abhängig sind, stehen CoObRA keine Informationen über die Kompositionsstruktur der Objekte zur Verfügung. Folglich kann nur die Gesamtheit aller Objekte auf Repository-Ebene versioniert werden. Diese Art der Versionierung ist nicht geeignet für die Verwaltung großer Projekte, bei denen jeder Entwickler nur mit einem Teil der Dokumente arbeitet. Die Änderungen von Teilen der Datenbasis können in CoObRA nicht separat behandelt werden. DRAGOS bietet diesbezüglich mehr Flexibilität. Zudem kann auch das Versionieren des gesamten Graphpools simuliert werden, indem ein einziger Graph auf oberster Ebene angelegt wird und alle anderen Graphen seine Untergraphen darstellen.

Der Versionsraum eines CoObRA-Repositories ist linear strukturiert. Durch das Ableiten mehrerer Arbeitskopien aus dem Server-Repository können zwar Varianten entstehen, jedoch ist innerhalb eines Repositories keine Verzweigung möglich. Beim Check-In wird der lokale Entwicklungszweig wieder mit dem des Servers zusammengeführt. DRAGOS erlaubt hingegen beliebiges Verzweigen des Versionsraumes.

	CoObRA	**DRAGOS**
Versionierte Objekte	gesamtes Repository	Graphen (Versionierung des gesamten Graphpools simulierbar)
Parallele Entwicklung	nur Arbeitskopien möglich, keine parallelen Entwicklungszweige	Baumartige Verzweigung der Entwicklung
Zusammenführen von Versionen	Merge-Operation für Arbeitskopien	Realisierung vorgesehen
Mehrbenutzerbetrieb	Verteiltes Arbeiten durch Server-Repository und lokale Repositories	Verteilung noch nicht realisiert. Später durch Versionierung realisierbar
Anwendungsspezifische Informationen im Versionsraum	Änderungsgründe	• Versionsbezeichner • Kommentare und Definition spezifischer Kommandos

Tabelle 6.2: Unterschiede zwischen CoObRA und DRAGOS

CoObRA realisiert einen Merge-Mechanismus, der beim Check-In die lokalen Änderungen in das Server-Repositories überträgt. Dabei auftretende Konflikte werden entweder selbstständig aufgelöst, oder es werden mögliche Lösungsmöglichkeiten erkannt und der Anwendung zur Auswahl präsentiert. Für DRAGOS ist ein ähnlicher Mechanismus vorgesehen, der zum Zusammenführen beliebiger Versionen genutzt werden kann. Er ist basierend auf den vorhandenen Delta-Informationen und den Eigenschaften des Versionsraumes zukünftig realisierbar. Wie im vorherigen Abschnitt erwähnt wurde, gilt gleiches auch für das verteilte Arbeiten mit getrennten Repositories. Bei CoObRA ist die Client/Server-Verteilung grundlegender Bestandteil der Realisierung.

CoObRA speichert für jede Änderung einen Änderungsgrund. In der kurzfristigen lokalen Änderungsverwaltung wird dieser von der Anwendung gesetzt, die dadurch den Versionraum durch spezifische Informationen strukturieren kann. So können beispielsweise Gruppen von Änderungen zusammengefasst werden. Das Markieren von bestimmten Zuständen beziehungsweise Versionen ist nicht explizit möglich. Auch das Erweitern der Änderungslisten um anwendungsspezifische Kommandos ist nicht vorgesehen. DRAGOS bietet diesbezüglich umfassendere Möglichkeiten. Das Gruppieren von Änderungen lässt sich mit Hilfe von Kommentar-Kommandos erreichen.

Zusammenfassung CoObRA implementiert eine vollständige Client/Server-Verteilung, welche die parallele Arbeit mehrerer Benutzer an einer Datenbasis ermöglicht. Solche Funktionalität wird in DRAGOS erst in Zukunft integriert werden. Bezogen auf die Versionierungsfunktionen bietet DRAGOS jedoch weitaus flexiblere Einsatzmöglichkeiten. Durch das Versionieren auf Repository-Ebene sowie die fehlende Unterstützung baumartiger Entwicklungen ist das Anwendungsspektrum von CoObRA auf die Kernfunktionalität begrenzt. Die Versionskontrolle stellt eine Vereinfachung des in DRAGOS realisierten Modells dar und kann mit diesem nachgebildet werden.

6.2 Zusammenfassung

DRAGOS besitzt den Anspruch, die Integration beliebiger Datenmodelle in eine gemeinsame Datenbank zu ermöglichen. Werkzeuge aus dem Bereich der Softwareentwicklung stellen dabei das Haupteinsatzgebiet dar. Der allgemeine Anspruch von DRAGOS spiegelt sich auch im entwickelten Versionsmodell wieder. Es ist geeignet, ein großes Spektrum unterschiedlicher Anforderungen zu erfüllen. Die realisierte Versionskontrolle stellt umfangreiche Basismechanismen zur Verfügung, die sich individuell an spezielle Anwendungsbedürfnisse anpassen lassen. Dabei wurde eine vollständige Integration von kurzfristiger und langfristiger Änderungsverwaltung in ein gemeinsames Modell vollzogen. So ist persistente Verwaltung unterschiedlicher Entwicklungszweige ebenso möglich, wie die Bereitstellung von Undo/Redo-Funktionalität für eine Benutzeroberfläche.

Bei der Entwicklung des Versionsmodells wurde das Ziel verfolgt, der Anwendung ein Maximum an Flexibilität zu ermöglichen. In besonderem Maße trägt die Versionierbarkeit jedes einzelnen Graphen zu dieser Flexibilität bei. Durch Beziehungen zwischen getrennt versionierten Graphen kann die Konsistenz des Graphpools beeinträchtigt werden. Das grundsätzliche Verhindern von Konsistenzverletzungen wäre nur dadurch zu erreichen, dass der Anwendung Einschränkungen bei der Nutzung der Versionierungsfunktionen auferlegt werden. Dieser Ansatz ist jedoch nicht sinnvoll, da oftmals die Semantik der Anwendung für Konsistenz zwischen den versionierten Graphen sorgt. Der Ansatz von DRAGOS bietet der Anwendung umfassende Unterstützung, um die Konsistenz ihrer Datenstrukturen dort sicherzustellen, wo sie tatsächlich gefährdet ist. Solche kritischen Stellen sind auf Datenbank-Ebene nicht erkennbar.

Konfigurationen stellen ein zusätzliches Konstrukt dar, das DRAGOS-basierten Anwendungen zur Verfügung steht. Sie sind nicht nur für die klassische Konfigurationsverwaltung einsetzbar, sondern stellen ein universelles Hilfsmittel zur Verwaltung von Versionsmengen dar.

Die Realisierung der Versionskontrolle basiert auf einem Logging-Mechanismus, der sowohl Graphveränderungen als auch Versionsveränderungen protokolliert. Die entstehenden Kommandolisten enthalten ausreichende Informationen, um als Grundlage für weitergehende Konzepte zu dienen. Die komplexe Versionierungsfunktionalität kann so durch wenige Basis-Konzepte realisiert werden. Viel Wert wurde zudem auf die transparente Integration der Versionierungserweiterung in das universelle Graphmodell von DRAGOS gelegt. Durch das Wrapper-Konzept können zusätzliche Erweiterungen sowohl die Versionierungsfunktionalität nutzen als auch deren Möglichkeiten erweitern. Durch die klare Trennung der Erweiterung von der Graphmodellfunktionalität wird eine gute Wartbarkeit erreicht.

Ein erster Einsatz der Versionskontrolle in der Implementierung der PROGRES-Codegenerierung für DRAGOS zeigt ihre Anwendungstauglichkeit. Den mit PROGRES spezifizierten Anwendungen wird damit ermöglicht, DRAGOS als Graphenspeicher zu verwenden. Um den Backtracking-Mechanismus von PROGRES zu realisieren, wird Versionskontrolle eingesetzt.

Der folgende Abschnitt gibt einen Überblick über Optimierungen und Weiterentwicklungen der Versionierungserweiterung, deren Umsetzung Ziel zukünftiger Arbeiten sein könnte.

6.3 Ausblick

Die bestehende Implementierung der Versionierungserweiterung ist an einigen Stellen optimierbar. So könnte der Speicherplatzverbrauch reduziert werden, indem die Algorithmen zur Delta-Optimierung erweitert werden. Zudem sind die Möglichkeiten zum nachträglichen Löschen von Versionen zu verbessern. Hier wird der Anwendung bislang wenig Unterstützung bei der Konsistenzsicherung geboten.

Basierend auf den Konzepten, die in dieser Arbeit entwickelt wurden, sind zukünftige Erweiterungen denkbar. Undo/Redo-Operationen stellen eine wichtige Funktionalität dar, die ohne großen Aufwand realisierbar ist. Für flache

Graphen ist Undo/Redo direkt implementierbar, indem eine entsprechende Erweiterung dafür sorgt, dass zu bestimmten Zeitpunkten neue Versionen des betrachteten Graphen erzeugt werden. Indem diese mit erweiterungsspezifischen Bezeichnern markiert werden, sind sie von anderen Versionen unterscheidbar. Eine Undo- oder Redo-Operation stellt den Zustand der letzten beziehungsweise nächsten so markierten Version her. Dabei ist die Erkennung eines eindeutigen Redo-Pfades vorausgesetzt. Eine neue Versionserzeugung kann nach jeder atomaren Graphveränderung, nach jedem Transaktionsende oder frei von der Anwendung initiiert werden – je nachdem, wie feingranular Undo-Schritte möglich sein sollen.

Undo/Redo-Operationen, die sich auf Graphveränderungen an geschachtelten Graphen beziehen, müssen berücksichtigen, dass die rückgängig zu machenden Kommandos auf mehrere Deltas verteilt sind. Dies ist dadurch erklärt, dass die Änderungen an tieferliegenden Graphen im Log ihres Vater-Graphen protokolliert werden. Es muss also sichergestellt werden, dass die Erzeugung der Undo-relevanten Versionen auf der Ebene erfolgt, die über Undo/Redo-Funktionen verfügen soll. Dies könnte beispielsweise die oberste Graphebene sein, wenn dort einzelne Dokumente abgelegt sind.

DRAGOS soll in absehbarer Zeit um Mehrbenutzerfähigkeit erweitert werden, wozu die Versionierungskonzepte verwendet werden können. Wenn verschiedene Entwickler lokal auf eigenen Graphpools arbeiten, ist es notwendig, Arbeitskopien von Graphstrukturen erstellen zu können. Die vorgestellte Replikation von Graphen kann als Ausgangspunkt für die weitere Entwicklung verwendet werden. Für das Check-In von Änderungen, die an lokalen Arbeitskopien getätigt wurden, sind Mechanismen erforderlich, die das Zusammenführen von Versionen ermöglichen. Zukünftige Arbeiten müssen sich diesbezüglich damit beschäftigen, wie auftretende Konflikte erkannt werden, und ob sie automatisch beseitigt werden können. Ist letzteres nicht der Fall, muss ein Weg gefunden werden, die Konfliktsituation an die Anwendung weiterzuleiten und ihr entsprechende Reaktionsmöglichkeiten zu offerieren.

In dieser Arbeit wurde vorausgesetzt, dass das Graphschema konstant ist. Längerfristig kann die Versionierungserweiterung um Aspekte der Schema-Evolution erweitert werden. So sollte die nachträgliche Modifikation von Schemaklassen von der Versionierung erfasst werden. Zwei unterschiedliche Vorgehensweisen sind dabei grundsätzlich denkbar: Zum einen könnte das Graphschema mit versioniert werden, dass heisst eine vergangene Version eines Graphen instanziiert auch weiterhin die historische Version der Klasse. Zum anderen könnte auch eine Konvertierung aller Versionen auf die neue Klasse vorgenommen werden. Generell können durch nachträgliche Schemaveränderungen

Konflikte im aktuellen Zustand des Graphpools auftreten. Bei der zweiten Vorgehensweise muss eine Konflikterkennung jedoch auf alle Kommandos ausgeweitet werden, die in existierenden Deltas vorhanden sind. Es wird zu untersuchen sein, welche Anforderungen Schema-Evolution an die Versionskontrolle stellt.

Die vorliegende Arbeit zeigt, wie ein graph-basiertes Datenbanksystem um Versionierungsfunktionen erweitert werden kann. Die erarbeiteten Konzepte werden dem universellen Anspruch von DRAGOS gerecht und sind gut geeignet, die Anforderungen zukünftiger Anwendungen zu erfüllen.

Literaturverzeichnis

[Bau99] BAUMANN, ROLAND: *Ein Datenbankmanagementsystem für verteilte, integrierte Software-Entwicklungsumgebungen*. Dissertation, RWTH Aachen, 1999.

[Ber90] BERLINER, B.: *CVS II: Parallelizing Software Development*. In: *Proceedings of the USENIX Winter 1990 Technical Conference*, Seiten 341–352, Berkeley, CA, 1990. USENIX Association. `citeseer.ist.psu.edu/berliner90cvs.html`.

[Böh04] BÖHLEN, BORIS: *Specific Graph Models and Their Mappings to a Common Model*. In: PFALTZ, JOHN L., MANFRED NAGL und BORIS BÖHLEN (Herausgeber): *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003*, Band 3062 der Reihe *Lecture Notes in Computer Science*, Seiten 45–60. Springer-Verlag, Heidelberg, 2004.

[BK91] BARGHOUTI, NASER S. und GAIL E. KAISER: *Concurrency Control in Advanced Database Applications*. ACM Computing Surveys, 23(3):269–317, 1991. `citeseer.ist.psu.edu/barghouti94concurrency.html`.

[CS02] COLLINS-SUSSMAN, BEN: *The subversion project: buiding a better CVS*. Linux J., 2002(94):3, 2002.

[CW98] CONRADI, REIDAR und BERNHARD WESTFECHTEL: *Version models for software configuration management*. ACM Comput. Surv., 30(2):232–282, 1998.

[EKMR99] EHRIG, H., H.-J. KREOWSKI, U. MONTANARI und G. ROZENBERG (Herausgeber): *Handbook of graph grammars and computing by graph transformation: vol. 3: concurrency, parallelism, and distribution*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[GRA] *DRAGOS Web Site*. `http://www-i3.informatik.rwth-aachen.de/grasgxl`, Stand 2005-10-20.

[GXL03] *GXL Web Site*, 2003. `http://www.gupro.de/GXL`, Stand 2005-10-20.

[Ham97] HAMILTON, GRAHAM: *Sun Microsystems: JavaBeans Specification*. 1997. `http://java.sun.com/products/javabeans/docs/spec.html`.

[JEACS84] JAMES E. ARCHER, JR., RICHARD CONWAY und FRED B. SCHNEIDER: *User Recovery and Reversal in Interactive Systems*. ACM Trans. Program. Lang. Syst., 6(1):1–19, 1984.

[Kle93] KLEIN, PETER: *Die Graphcodemaschine PROGRES*. Diploma thesis, Lehrstuhl für Informatik III, RWTH Aachen, 1993.

[KNNZ99] KLEÍN, THOMAS, ULRICH A. NICKEL, JÖRG NIERE und ALBERT ZÜNDORF: *From UML to Java And Back Again*. Technischer Bericht tr-ri-00-216, University of Paderborn, Paderborn, Germany, September 1999.

[KSW95] KIESEL, NORBERT, ANDY SCHÜRR und BERNHARD WESTFECHTEL: *GRAS, a Graph-Oriented (Software) Engineering Database System*. Information Systems, 20(1):21–51, 1995.

[Mom00] MOMJIAN, BRUCE: *PostgreSQL: Introduction and Concepts*. Addison Wesley, Reading, MA, USA, 2000.

[Mor96] MORSE, TOM: *CVS*. Linux J., 1996(21es):3, 1996.

[Nag96] NAGL, MANFRED (Herausgeber): *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, Band 1170 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 1996.

[Opd92] OPDYKE, WILLIAM F.: *Refactoring Object-Oriented Frameworks*. Doktorarbeit, Urbana-Champaign, IL, USA, 1992. `citeseer.ist.psu.edu/opdyke92refactoring.html`.

[Roo02] ROOS, ROBIN: *Java Data Objects*. Pearson Education, 2002.

[Sch91] SCHÜRR, ANDY: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Dissertation, RWTH Aachen, 1991.

[Sch03] SCHNEIDER, C.: *CASE Tool Unterstützung für die Delta-basierte Replikation und Versionierung komplexer Objektstrukturen*. Diplomarbeit, Corolo Wilhelmina zu Braunschweig, 2003. `http://www.se.eecs.uni-kassel.de/coobra`.

[SUN] SUN MICROSYSTEMS INC.: *The Java Tutorial, The Reflection API*. `http://java.sun.com/docs/books/tutorial/reflect/TOC.html`, Stand 2006-01-22.

[Vin97] VINOSKI, STEVE: *CORBA: integrating diverse applications within distributed heterogeneous environments*. IEEE Communications Magazine, 14(2), 1997. `citeseer.ist.psu.edu/vinoski97corba.html`.

[Vit84] VITTER, JEFFREY SCOTT: *US&R: A new framework for redoing (Extended Abstract)*. In: *SDE 1: Proceedings of the first ACM SIGSOFT/-SIGPLAN software engineering symposium on Practical software development environments*, Seiten 168–176, New York, NY, USA, 1984. ACM Press.

[Vos99] VOSSEN, G.: *Datenbankmodelle, Datenbanksprachen und Datenbankmanagement-Systeme*. R. Oldenbourg Verlag, München, 3. Auflage, 1999.