UNIVERSITÄT KOBLENZ · LANDAU

ist

Diplomarbeit
zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

Transaktionskonzept für die TGraphenbibliothek JGraLab

vorgelegt von

José Angel Monte Barreto <monte@uni-koblenz.de>
Mat.-Nr. 203210027

am 4. März 2009

Betreuer

Prof. Dr. Jürgen Ebert <ebert@uni-koblenz.de>
Dr. Volker Riediger <riediger@uni-koblenz.de>
Daniel Bildhauer <dbildh@uni-koblenz.de>

Erklärung

Hiermit versichere ich, wie in §10 Abschnitt 6.2 der Diplomprüfungsordnung für Studierende der Informatik an der Universität Koblenz-Landau gefordert, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung im Internet stimme ich ebenfalls zu.

Danksagung

Bedanken möchte ich mich bei Prof. Dr. Jürgen Ebert, Dr. Volker Riediger und Daniel Bildhauer für die kompetente Betreuung, die vielen hilfreichen Ratschläge und eine angenehme Arbeitsatmosphäre. Die regelmäßigen Treffen, in denen die jeweiligen Zwischenergebnisse der Diplomarbeit ausführlich diskutiert und kommentiert wurden, empfand ich als sehr hilfreich.

Abstract

Im Rahmen dieser Diplomarbeit wird ein Transaktionskonzept für die aktuelle Implementationsversion der TGraphenbibliothek *JGraLab Carnotaurus* umgesetzt.

Nach einer grundlegenden Einführung in das Konzept der *TGraphen* werden die relevanten Implementationsdetails der TGraphenbibliothek erläutert. Anschließend erfolgt ein konzeptueller Entwurf, in dem die formalen Grundlagen des Transaktionskonzepts beschrieben werden. Das aus der Datenbankwelt bekannte *ACID*-Paradigma für Transaktionen dient dabei als wissenschaftliche Grundlage.

In einem nächsten Schritt erfolgt der objektorientierte Feinentwurf der Integration des zu entwickelnden Transaktionskonzepts in das vorhandene Gesamtsystem, anhand dessen die Implementation durchgeführt wird.

Eine Analyse und Bewertung des umgesetzten Transaktionskonzepts (vor allem im Hinblick auf den Speicherverbrauch und das Laufzeitverhalten) schließen die Arbeit ab.

Inhaltsverzeichnis

1 Einführung

Dieses Kapitel dient als Einführung in das Thema der Diplomarbeit. In einem ersten Schritt wird die Motivation zur Entwicklung eines Transaktionskonzepts für das Graphenlabor erläutert (siehe Abschnitt 1.1). Anschließend werden grundlegende Definitionen, Begriffe und Konzepte im Kontext der TGraphen erläutert (siehe Abschnitt 1.2). In einem letzten Teil wird ein Überblick über die Struktur und den Inhalt der Diplomarbeit geliefert (siehe Abschnitt 1.3).

1.1 Motivation

Die *Graphentechnologie* ist ein Forschungsschwerpunkt der Arbeitsgruppe Ebert im Institut für Softwaretechnik der Universität Koblenz-Landau[1]. Sie dient der Entwicklung von Anwendungssoftware, welche durch die Methoden der *Graphentheorie* und die Verwendung geeigneter *Graphenalgorithmen* realisiert wird und *Graphen* als interne Datenstruktur zur Darstellung von (komplexen) strukturierten Informationen einsetzt.

Die Arbeitsgruppe Ebert verwendet dabei die sogenannten *TGraphen* (siehe Abschnitt 1.2) als generelle Graphenrepräsentation. TGraphen sind *gerichtete*, *typisierte*, *attributierte* und *angeordnete* Graphen. Die Eigenschaften von TGraphen erlauben es, verschiedenste Graphenarten mit unterschiedlicher Mächtigkeit darzustellen. Die „Besonderheit“ der TGraphen liegt vor allem in der Verwendung eines Typsystems, wonach den einzelnen *Graphenelementen* (*Knoten* und *Kanten*) ein definierter Typ zugewiesen wird. Die *(Mehrfach-)Vererbung* zwischen den *Graphenelementtypen* wird ebenfalls unterstützt. Zudem können die Graphenelemente von ihrem Typ abhängige Attribute besitzen.

Um letztlich Anwendungssoftware mit Hilfe von TGraphen erstellen zu können, wurde im Institut für Softwaretechnik eine Klassenbibliothek entwickelt, welche als *Graphenlabor* (abgekürzt mit *GraLab*) bezeichnet wird (siehe Kapitel 2). Das Graphenlabor ermöglicht es, TGraphen als (effiziente) interne Datenstruktur abzubilden. Es bietet zudem eine API zur Erstellung von TGraphen, aber auch zur Manipulation und zur Traversierung von TGraphen zur Laufzeit an.

[1] `http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/MainResearch/Graphentechnologie/Graphentechnologie` (abgerufen am 4. März 2009)

Im Rahmen einer im Jahre 2006 abgeschlossenen Diplomarbeit [Kah06] wurde eine auf der Programmiersprache Java basierende Version des Graphenlabors (genannt *JGraLab*) umgesetzt. Diese bietet als wichtigste Neuerung eine objektorientierte Manipulations- und Zugriffsschicht für die TGraphen und deren Elemente an. Somit wurde vor allem die Benutzerfreundlichkeit beim Umgang mit TGraphen verbessert. Allerdings unterstützt die aktuellste Version JGraLab *Carnotaurus* nicht die gleichzeitige (parallele) koordinierte Manipulation von TGraphen durch mehrere Prozesse. Folglich wird nur der *Einprozessbetrieb* realisiert.

Wünschenswert wäre jedoch die Umsetzung eines *Mehrprozessbetriebs*, welcher es erlaubt, einzelne (Änderungs-)Operationen auf einem TGraphen *atomar* und *thread-safe* auszuführen. Parallele Prozesse, die Änderungen an einem TGraphen durchführen, sollen ohne gegenseitige Beeinflussung (*isoliert*) ablaufen und nach deren Beendigung einen konsistenten Zustand des TGraphen hinterlassen. Diese Prozesse sollten also im kontrollierten Rahmen eines *Transaktionskonzepts* ablaufen.

Ziel dieser Diplomarbeit ist die *Konzeption*, die *Modellierung* und die anschließende *Implementierung* eines *Transaktionskonzepts* für die aktuelle Java-Version des Graphenlabors, welches sich möglichst nahtlos in das Gesamtsystem integrieren lassen soll. Dabei sollen Transaktionen, die im Rahmen des zu entwickelnden Transaktionskonzepts ablaufen, (möglichst) alle Eigenschaften von *ACID*-Transaktionen besitzen (siehe Kapitel 3). Die für die Einhaltung der ACID-Eigenschaften verwendeten Methoden in der Datenbankwelt dienen dabei als wissenschaftliche Grundlage. Diese müssen zunächst im Hinblick auf ihre Tauglichkeit im Kontext der Graphentechnologie untersucht werden, um anschließend gegebenenfalls in angepasster Form im Graphenlabor umgesetzt werden zu können.

1.2 TGraphen

Graphen bilden in der Informatik eine grundlegende Datenstruktur, welche effizient nutzbar ist und mit formalen Methoden untersucht werden kann. Viele der auftretenden algorithmischen Probleme lassen sich mit Hilfe von Graphen modellieren und anschaulich und strukturiert darstellen [DW98]. Dabei liefern Graphen eine abstrakte Darstellung von *Objekten* und deren *Beziehungen* (zueinander) [Sed02].

Im Kontext des Graphenlabors (siehe Kapitel 2) dienen die TGraphen als Graphenrepräsentation und bilden die interne Datenstruktur, auf der das Graphenlabor operiert. Um daher die vom Graphenlabor bereitgestellte Funktionalität besser verstehen und einordnen zu können, müssen zunächst die grundlegenden Begriffe und Konzepte im Zusammenhang mit TGraphen erläutert werden.

1.2.1 Graphen allgemein

Ein *Graph* besteht aus einer endlichen Menge von *Knoten* und *Kanten*. In ihrer grafischen Darstellung werden Knoten meist als *Punkte* und Kanten als verbindende *Linien* zwischen Knoten repräsentiert. Bei der Interpretation eines Graphen ist dabei nur entscheidend, welche Knoten durch welche Kanten im Graphen verbunden sind [Sed02]. Die räumliche Position und Anordnung der Knoten und Kanten ist demnach irrelevant.

1.2.2 Charakteristische Eigenschaften von TGraphen

Im Kontext dieser Diplomarbeit werden die sogenannten *TGraphen* betrachtet. TGraphen sind durch die folgenden Eigenschaften charakterisiert [DW98]:

Gerichtet — TGraphen werden prinzipiell als gerichtete Graphen modelliert. Kanten in TGraphen sind demnach immer gerichtet und haben einen definierten *Start*- und *Endknoten*.

Allerdings können TGraphen bei Bedarf (zum Beispiel zur Traversierung) problemlos als ungerichtete Graphen betrachtet werden, ohne Veränderungen an deren internen Darstellung vornehmen zu müssen.

Typisiert — Jedem Graphenelement und dem Graphen selbst werden ein bestimmter definierter Typ zugewiesen.

„Zusammengehörende“ Graphenelemente (mit gemeinsamen Eigenschaften) werden durch die Zuweisung des gleichen Typs klassifiziert. Es wird dabei zwischen *Knoten*-, *Kanten* und *Graphentypen* unterschieden.

Attributiert — Knoten, Kanten und Graphen können *Attribut-Wert-Paare* besitzen.

Die einzelnen Graphenelemente und der Graph selbst können Attribute besitzen, welchen wiederum ein Wert zugewiesen werden kann.

Angeordnet — Diese Eigenschaft besagt, dass die einzelnen Graphenelemente in einer totalen Anordnung zueinander im Graphen stehen.

Die jeweiligen Anordnungen werden in *injektiven Sequenzen* (*iseq*) gehalten. Dies sind Listen, in denen ein Graphenelement jeweils höchstens einmal vorkommen darf.

1.2.3 Beispiel für einen TGraphen

Abbildung 1.1 zeigt ein Beispiel für einen TGraphen TG_1.

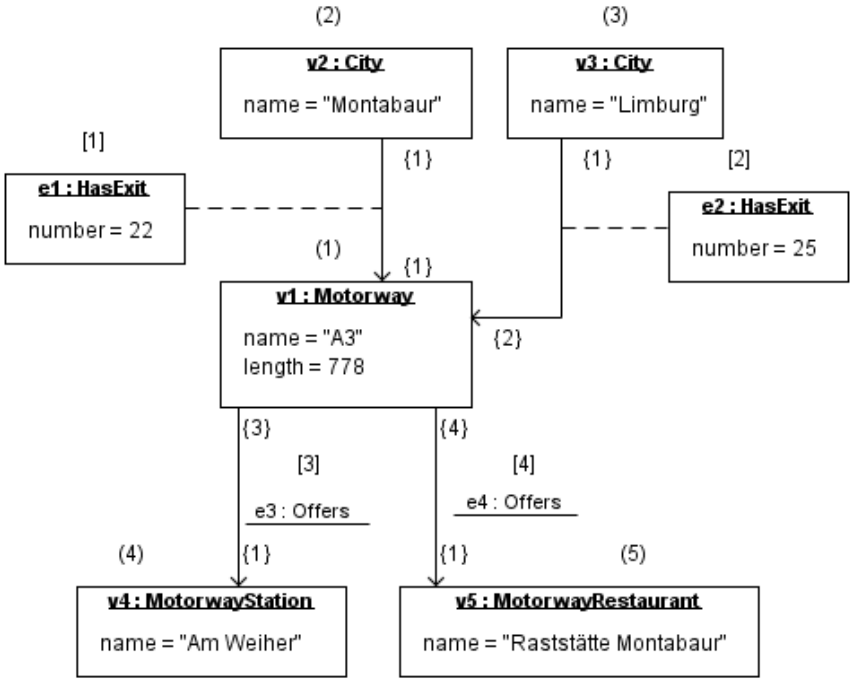


Abbildung 1.1: Beispiel für einen TGraphen TG_1

Zur Notation wird *GrUML* (*Graph Unified Modelling Language*) verwendet, eine auf UML basierende Modellierungssprache für TGraphen [BER08]. Der dargestellte TGraph stellt eine fiktive Autobahnkarte dar, in der für eine Autobahn (*motorway*) sowohl Rastplätze (*motorway stations*) und Raststätten (*motorway restaurants*) als auch Autobahnausfahrten (*exits*) zu Ortschaften (*cities*) angezeigt werden.

In TG_1 sind alle Kanten gerichtet, wobei die Richtung in der GrUML-Notation durch die Pfeilspitzen angedeutet ist. Der Startknoten liegt demnach am Richtungsanfang und der Endknoten an der Pfeilspitze der jeweiligen Kante.

Jedem Graphenelement in TG_1 ist ein Typ zugewiesen; beispielsweise ist der Knoten *v1* vom Typ *Motorway* (*v1 : Motorway*), während die Kante *e1* mit *HasExit* (*e1 : HasExit*) typisiert ist. Ein Knoten vom Typ *Motorway* repräsentiert dabei eine Autobahn, welcher durch die Kante vom Typ *HasExit* mit einem Knoten vom Typ *City* verbunden ist, um eine Autobahnausfahrt zu einer Ortschaft zu signalisieren. Kanten vom Typ *Offers* verbinden Knoten vom Typ *Motorway* mit Knoten vom Typ *MotorwayStation* oder vom Typ *MotorwayRestaurant*, um auszudrücken, dass eine Autobahn einen Rastplatz bzw. eine Raststätte anbietet. *City*, *Motorway*, *MotorwayStation* und *MotorwayRestaurant* stellen die *Knotentypen* dar, während *HasExit* und *Offers* die *Kantentypen* repräsentieren.

In TG_1 besitzen beispielsweise alle Knoten vom Typ *Motorway* die Attribute *name* und *length* (um den Namen einer Autobahn bzw. die Autobahnlänge darzustellen), während alle Kanten vom Typ *HasExit* mit *number* attributiert sind (um die Nummer der Autobahnausfahrt anzugeben).

Die (globale) Anordnung der Knoten in TG_1 ist in Abbildung 1.1 durch die Zahlen in runden Klammern angedeutet, die (globale) Anordnung der Kanten durch die Zahlen in eckigen Klammern und die (lokale) Anordnung der Verbindungspunkte eines Knotens mit Kanten durch die Zahlen in geschweiften Klammern.

1.2.4 Formale Definition von TGraphen

Definition 1.1 (TGraph)

Seien

- *Vertex* das Universum der **Knoten**,
- *Edge* das Universum der **Kanten**,
- *TypeID* das Universum der **Typbezeichner**,
- *AttrID* das Universum der **Attributbezeichner** und
- *Value* das Universum der **Attributwerte**.

Ferner seien

- $V \subseteq Vertex$ eine endliche Menge von **Knoten** und
- $E \subseteq Edge$ eine endliche Menge von **Kanten**.

Dann ist $G = (Vseq, Eseq, \Lambda seq, type, value)$ ein **TGraph**, falls gilt:

- $Vseq \in \text{iseq } V$ ist eine **Anordnung von V**,
- $Eseq \in \text{iseq } E$ ist eine **Anordnung von E**,
- $\Lambda\text{seq} : V \rightarrow \text{iseq}(E \times \{\text{in, out}\})$ ist eine **Inzidenzabbildung**, für die gilt:
  $\forall\, e \in E\ \exists_1\, v, w \in V : (e, out) \in \text{ran } \Lambda\text{seq(v)} \wedge (e, in) \in \text{ran } \Lambda\text{seq(w)}$,
- $type : V \cup E \rightarrow TypeID$ ist eine **Typisierung**,
- $value : V \cup E \rightarrow (AttrID \nrightarrow Value)$ ist eine **Attributierung** und
- $\forall\, x, y \in V \cup E : type(x) = type(y) \Rightarrow dom(value(x)) = dom(value(y))$.

Definition 1.1 zeigt die formale Definition von TGraphen, welche auf der in [Ste06] eingeführten Definition aufbaut. Diese wird im Folgenden natürlichsprachlich erläutert und an geeigneten Stellen mit Hilfe des Beispielgraphen TG_1 verdeutlicht.

Zunächst werden die Universen *Vertex*, *Edge*, *TypeID*, *AttrID* und *Value* definiert. Das Universum *Vertex* der Knoten und das Universum *Edge* der Kanten stellen die Graphenelemente dar. Zur Umsetzung der Typisierung dient das Universum *TypeID* der Typbezeichner. Um die Attributierung umsetzen zu können, werden jeweils das Universum *AttrID* von Attributbezeichnern und das Universum *Value* von Attributwerten eingeführt.

Wie jede andere Graphenart auch, setzen sich TGraphen aus einer endlichen Menge *V* von *Knoten* und einer endlichen Menge *E* von *Kanten* zusammen. TG_1 besteht aus der Knotenmenge *V* = {v1, v2, v3, v4, v5} und aus der Kantenmenge *E* = {e1, e2, e3, e4}.

Ein TGraph *G* wird als Fünftupel *G = (Vseq, Eseq, Λseq, type, value)* definiert.

Dabei stellt die injektive Sequenz *Vseq* eine zu einem Zeitpunkt gültige Auflistung der im Graphen *G* auftretenden Knoten dar. *Vseq* repräsentiert daher die *Anordnung* der Menge *V* im Graphen *G*. In TG_1 gilt *Vseq* = <v1, v2, v3, v4, v5>.

Die injektive Sequenz *Eseq* ist analog eine zu einem Zeitpunkt gültige Auflistung der im Graphen *G* auftretenden Kanten. *Eseq* entspricht also der *Anordnung* der Menge *E* im Graphen *G*. In TG_1 gilt *Eseq* = <e1, e2, e3, e4>.

In der Inzidenzabbildung wird jedem in *G* auftretenden Knoten $v \in V$ eine injektive Sequenz von mit *v* verbundenen Kanten aus der Menge *E* zugeordnet. Dabei wird eine Kante $e \in E$ als *inzident* zu einem Knoten $v \in V$ bezeichnet, falls *e* mit *v* verbunden ist. Der Verbindungspunkt wird als *Inzidenz* des Knotens *v* bezeichnet. Für jede mit einem Knoten *v* verbundene Kante *e* wird unterschieden, ob sie eingehend (*in*) oder ausgehend (*out*) ist. Bei eingehenden Kanten repräsentiert der Knoten *v* den *Zielknoten* von *e*, während bei ausgehenden Kanten der Knoten *v* den *Startknoten* von *e* darstellt. Zudem wird festgelegt, dass für eine Kante $e \in E$ nur genau zwei Knoten $v, w \in V$ existieren, so dass *e* eine ausgehende Kante (*e,out*) bezüglich *v* und eine eingehende Kante (*e,in*) bezüglich *w* darstellt. Mit anderen Worten muss eine Kante *e* in einem Graphen *G* immer genau *einen* Startknoten mit genau *einem* Endknoten verbinden. Dabei ist laut Definition nicht ausgeschlossen, dass Start- und Endknoten identisch sein können. *Λseq(v)* liefert also die Anordnung aller eingehenden und ausgehenden Kanten eines Knotens *v*. In TG_1 gilt für den Knoten *v1* die Inzidenzabbildung *Λseq(v1)* = <(e1,in), (e2,in), (e3,out), (e4,out)>.

In der Funktion *type* wird jedem Graphenelement in *G* genau ein Typbezeichner aus dem Universum *TypeID* zugeordnet. *type* setzt also die *Typisierung* in TGraphen um. In TG_1 ist zum Beispiel *type(v1) = Motorway* und *type(e1) = HasExit*.

Die Funktion *value* ordnet jedem Graphenelement in *G* genau eine (gegebenenfalls leere) Menge von Attributen zu. Einem Attributbezeichner aus *AttrID* kann dabei höchstens ein Attributwert aus *Value* zugewiesen werden. Diese Funktion setzt die *Attributierung* in TGraphen um. In TG_1 gilt beispielsweise *value(v1)* = {(name, A3), (length, 778)} und *value(e1)* = {(number, 22)}.

Im letzten Punkt der Definition wird festgelegt, dass für zwei Graphenelemente x und y des gleichen Typs immer gelten muss, dass ihre jeweiligen Mengen an Attributbezeichnern identisch sind. Dies entspricht dem Grundsatz der Objektorientierung, welcher besagt, dass Objekte des gleichen Typs immer die gleichen Attribute besitzen.

1.2.5 Weiterführende Begriffe und Definitionen

Aufbauend auf der formalen Definition von TGraphen werden nachfolgend weitere Begriffe und Definitionen eingeführt [DEL94].

Ein Knoten v, welcher keine Inzidenzen besitzt - für den also $\Lambda seq(v) = <>$ gilt - wird als *isoliert* bezeichnet. Die Funktionen $\alpha, \omega : E \rightarrow V$ liefern zu einer Kante $e \in E$ den dazugehörigen Start- (α) bzw. Endknoten (ω). Eine Kante $e \in E$ wird als *Schlinge* bezeichnet, falls $\alpha(e) = \omega(e)$ gilt. Besitzen zwei voneinander verschiedene Kanten $e1$, $e2$ ($e1 \neq e2$) den gleichen Start- und Endknoten, werden sie als *Mehrfachkanten* bezeichnet ($\alpha(e1) = \alpha(e2)$ und $\omega(e1) = \omega(e2)$).

Bei der Inzidenzabbildung kann zwischen der Anordnung von *eingehenden* und *ausgehenden* Kanten unterschieden werden. $\Lambda seq^{-}(v)$ liefert die Sequenz, welche nur diejenigen Paare enthält, deren zweite Komponente *in* ist. Λseq^{-} ist definiert als $\Lambda seq^{-} : V \rightarrow iseq(E \times \{in\})$ und berücksichtigt also lediglich Inzidenzen eines Knotens $v \in V$ mit eingehenden Kanten. $\Lambda seq^{+}(v)$ liefert seinerseits die Sequenz, welche nur Paare enthält, deren zweite Komponente *out* ist. Λseq^{+} ist definiert als $\Lambda seq^{+} : V \rightarrow iseq(E \times \{out\})$ und berücksichtigt also lediglich Inzidenzen eines Knotens $v \in V$ mit ausgehenden Kanten. Wichtig ist zu erwähnen, dass die Inzidenzabbildung $\Lambda seq(v)$ nicht zwischen eingehenden und ausgehenden Kanten unterscheidet, und somit eine Schlinge e für einen Knoten $v \in$ V zweimal aufgeführt wird, nämlich einmal für die eingehende Richtung *(e,in)* und einmal für die ausgehende Richtung *(e,out)*. In TG_1 gilt $\Lambda seq^{+}(v1) =$ <(e3,out), (e4,out)> und $\Lambda seq^{-}(v1) =$ <(e1,in), (e2,in)>.

Für einen Knoten $v \in V$ wird die Anzahl der mit v verbundenen Inzidenzen als der *Grad* δ des Knotens v bezeichnet. Nach dieser Definition ist demnach $\delta(v) = \# \Lambda seq(v)$. Für einen Knoten, welcher nur mit einer Schlinge verbunden ist, gilt δ(v) = 2. δ(v) umfasst alle Inzidenzen eines Knotens, unabhängig davon, ob es sich dabei um Verbindungspunkte mit eingehenden oder ausgehenden Kanten handelt. Zur Differenzierung kann für jeden Knoten $v \in V$ der *Innengrad* δ^{-}(v) und der *Außengrad* δ^{+}(v) berechnet werden. Der Innengrad eines Knotens v entspricht dabei der Anzahl der Verbindungspunkte von v mit eingehenden Kanten ($\delta^{-}(v) = \# \Lambda seq^{-}(v)$). Der Außengrad entspricht analog dazu der Anzahl der Verbindungspunkte von v mit ausgehenden Kanten ($\delta^{+}(v) = \# \Lambda seq^{+}(v)$). In TG_1 gilt δ(v1) = 4, δ^{+}(v1) = 2 und δ^{-}(v1) = 2.

1.2.6 Schemata als Mittel zur Umsetzung der Typisierung und Attributierung

Die Typisierung und die Attributierung der darzustellenden TGraphen wird mit dem *Konzept der Schemata* umgesetzt. Die Idee dabei ist, dass ein TGraph nur diejenigen Knoten- und Kantentypen (mit entsprechenden Attributen) enthalten darf, welche in seinem zugehörigen Schema definiert worden sind. Das Schema legt zudem fest, welche Knoten- und Kantentypen miteinander in Verbindung stehen dürfen. Die Struktur und die Elemente eines solchen Schemas werden wiederum durch ein *Meta-Schema* definiert.

1.2.6.1 Das Konzept der Metamodellierung

Die *Metamodellierung* ist demnach ein zentrales Mittel für den Umgang mit TGraphen. Zur allgemeinen Beschreibung der Metamodellierung wird in der Regel das *4-Schichten-Modell* [RHQ05] angewandt (siehe Abbildung 1.2). Dieses definiert von unten nach oben die Schichten *M0* (*Laufzeitschicht* oder *Realität*), *M1* (*Modellschicht*), *M2* (*Metamodellschicht*) und *M3* (*Meta-Metamodellschicht*). Dabei legen Modelle einer oberen Schicht die Elemente und die Struktur der darunter liegenden Schicht fest (*istModellFür*), während die untere Schicht eine gültige Instanziierung der darüber liegenden Schicht darstellt (*istInstanzVon*).

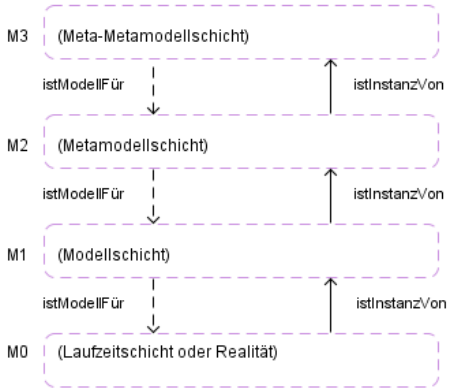


Abbildung 1.2: Das 4-Schichten-Modell der Metamodellierung

Im Folgenden werden die im Kontext der TGraphen definierten Schichten - welche dabei als *Ebenen* bezeichnet werden - aufgelistet und erläutert [Kah06].

M3, Meta-Schemaebene — Das in dieser Ebene definierte Meta-Schema bestimmt die Struktur und die Elemente eines Schemas der darunterliegenden Ebene *M2*. Zu beachten ist dabei, dass *Typen* hier (in Anlehnung an die Objektorientierung) als *Klassen* bezeichnet werden. Es werden demnach (unter anderem) *Graph*-, *Vertex*- und *EdgeClass* als Elemente eingeführt (siehe Abbildung 1.3).

M2, Schemaebene — In dieser Ebene werden Schemata erstellt, die den Aufbau und die Struktur von TGraphen der Ebene *M1* bestimmen. Sie bilden gültige Instanziierungen des definierten Modells (Meta-Schemas) in *M3*, so dass sie konkrete Instanzen von *Graph-*, *Vertex-* und *EdgeClass* beinhalten. Ein Schema kann dabei aus zwei verschiedenen Blickwinkeln interpretiert werden:

- als Instanz des Meta-Schemas der *M3*-Ebene oder
- als Metamodell und somit als Schema für die *M1*-Ebene.

M1, Graphenebene — Auf dieser Ebene werden die eigentlichen TGraphen dargestellt. TGraphen sind Instanzen von ihrem in *M2* definierten zugehörigem Schema. Sie enthalten nur Knoten und Kanten, welche gültige Instanziierungen ihrer jeweiligen im Schema definierten Knoten- und Kantenklassen darstellen. Gültige TGraphen werden auch als *schemakonform* bezeichnet.

1.2.6.2 Das Meta-Schema der Ebene M3

Abbildung 1.3 zeigt eine unvollständige Version des für TGraphen definierten Meta-Schemas (M3), welches nur die zum groben Verständnis notwendigen Elemente darstellt.

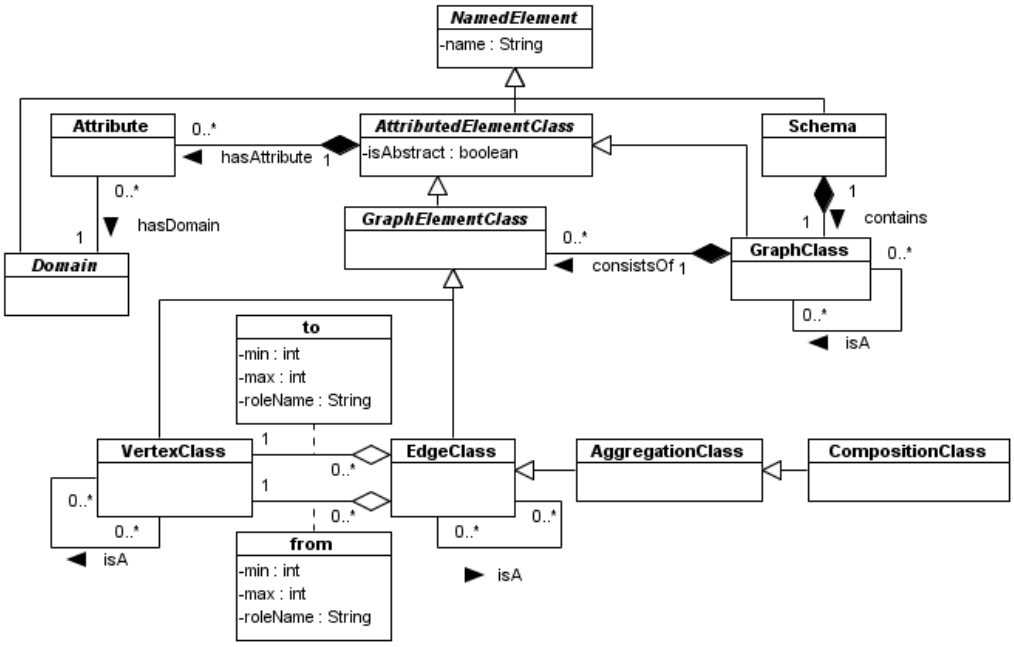


Abbildung 1.3: Meta-Schema der M3-Ebene nach [BER08]

Ein *Schema* beinhaltet (*contains*) eine Graphenklasse (*GraphClass*). Eine Graphenklasse besteht (*consistsOf*) ihrerseits aus beliebig vielen Knoten- (*VertexClass*) und Kantenklassen (*EdgeClass*). Durch die Assoziation *isA* wird die Vererbunghierarchie jeweils für Graphen-, Knoten- und Kantenklassen definiert, wobei Mehrfachvererbung erlaubt ist.

Mit den (Unter-)Klassen *AggregationClass* und *CompositionClass* lassen sich in einem Schema aus M2 die aus der UML bekannten Konzepte der *Aggregation* bzw. der *Komposition* anwenden.

Die Verbindungen zwischen Knoten- und Kantenklassen werden durch die Assoziationsklassen *to* und *from* ausgedrückt. Eine Verbindung kann durch die Attribute *min* und *max* eine Kardinalitätsangabe besitzen, welche im Schema die Definition von *Constraints* für die minimale bzw. maximale Anzahl der in Beziehung stehenden Knoten- und Kantenklassen erlaubt. Zudem bietet das Attribut *roleName* die Möglichkeit an, das jeweilige Ende einer Verbindung mit einem Rollennamen zu versehen.

Jede Graphen-, Knoten- oder Kantenklasse hat (*hasAttribute*) beliebig viele Attribute (*Attribute*), die jeweils einen bestimmten Wertebereich (*Domain*) besitzen (*hasDomain*).

Durch das boolesche Attribut *isAbstract* in der Klasse *AttributedElementClass* wird die Definition von abstrakten Graphen-, Knoten- und Kantenklassen in einem Schema der Ebene *M2* ermöglicht.

Alle in einem Schema der Ebene *M2* auftretenden Elemente besitzen (durch das Attribut *name* der Klasse *NamedElement*) einen (eindeutigen) Namen.

1.2.6.3 Beispiel für ein Schema der Ebene M2

Abbildung 1.4 zeigt ein Beispiel für ein Schema der Ebene *M2*. Es stellt dabei das zu dem TGraphen TG_1 in Abbildung 1.1 zugehörige Schema in GrUML-Notation dar.

Das Schema *MotorwayMapSchema* definiert die Elemente der Graphenklasse *MotorwayMap*. Diese enthält eine Knotenklasse *Motorway* mit den Attributen *name* vom Typ *String* und *length* vom Typ *double*. Eine weitere Knotenklasse *City* ist mit *name* attributiert. Die abstrakte Knotenklasse *StoppingPlace* mit dem *String*-Attribut *name* verallgemeinert die Knotenklassen *MotorwayStation* und *MotorwayRestaurant*. Die Kantenklasse (genauer die Kompositionsklasse) *Offers* verbindet die Knotenklasse *Motorway* mit der Knotenklasse *StoppingPlace*. Die Kardinalitäten geben an, wieviele Instanzen von *Motorway* mit Instanzen von *StoppingPlace* verbunden werden können und definieren demnach weitere *Constraints* für einen TGraphen der Klasse *MotorwayMap*. Die Kantenklasse (genauer die Aggregationsklasse) *HasExit* verbindet die Knotenklasse *City* mit der Knotenklasse *Motorway*. Die Kantenklasse *HasExit* ist ihrerseits mit dem Attribut *number* vom Typ *int*

attributiert. Die Kardinalitäten geben hier an, wieviele Instanzen von *City* mit Instanzen von *Motorway* verbunden werden können.

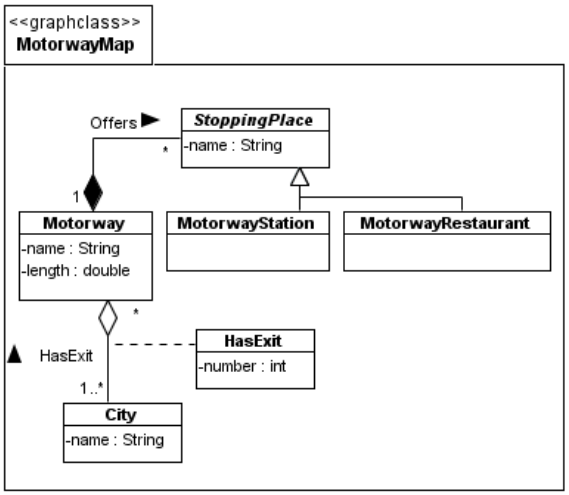


Abbildung 1.4: Das Schema *MotorwayMapSchema* in GrUML-Notation

1.2.6.4 Beispiel für einen TGraphen der Ebene M1

Als Beispiel für einen TGraphen der Graphenebene *M1* sei auf den bereits erläuterten und in Abbildung 1.1 dargestellten TGraphen TG_1 verwiesen. Dieser stellt eine gültige und demnach schemakonforme Instanz des Schemas in Abbildung 1.4 dar.

1.3 Überblick über die Diplomarbeit

Nachdem die grundlegenden Begriffe und Definitionen für die im Kontext dieser Diplomarbeit relevanten TGraphen im vorangegangenen Abschnitt 1.2 erläutert worden sind, folgt in Kapitel 2 zunächst ein Überblick über das Graphenlabor, welches TGraphen als interne Datenstruktur einsetzt. Hierbei werden die im Kontext der Diplomarbeit relevanten Aspekte der Implementation von JGraLab *Carnotaurus* erläutert.

Kapitel 3 behandelt das *ACID*-Paradigma für Transaktionen aus der Sicht der Datenbankwelt. Es werden verschiedene Synchronisations- und Fehlerbehandlungsmethoden vorgestellt und erläutert, mit denen die Einhaltung der Eigenschaften des *ACID*-Paradigmas für einzelne Transaktionen garantiert werden können.

Die Erkenntnisse aus Kapitel 3 dienen als wissenschaftliche Grundlage für Kapitel 4. Hier werden die formalen Grundlagen für das Transaktionskonzept im JGraLab erarbeitet. Zudem werden die Unterschiede zwischen Transaktionen im Graphenlabor und Transaktionen in Datenbanken aufgezeigt.

In Kapitel 5 werden zunächst die Bereiche der aktuellen JGraLab-Implementierung identifiziert, welche durch die Integration des in Kapitel 4 ausgearbeiteten Transaktionskonzepts angepasst und/ oder erweitert werden müssen. In einem zweiten Schritt erfolgt der objektorientierte Feinentwurf der Integration des Transaktionskonzepts im Gesamtsystem.

In Kapitel 6 wird das umgesetzte Transaktionskonzept vor allem im Hinblick auf den Speicherverbrauch und das Laufzeitverhalten analysiert und bewertet.

Kapitel 7 fasst die in dieser Diplomarbeit ausgearbeiteten Ergebnisse zusammen und liefert einen Überblick über noch ausstehende oder zu bearbeitende Frage- und Problemstellungen im Kontext des entwickelten Transaktionskonzepts.

2 Das Graphenlabor JGraLab

Das *Graphenlabor* ist eine Klassenbibliothek, welche TGraphen als interne Datenstruktur darstellen und verarbeiten kann. Sie erlaubt sowohl die Erstellung von TGraphen als auch deren Manipulation und Traversierung zur Laufzeit [DW98]. Die in der Programmiersprache Java umgesetzte und im Kontext dieser Diplomarbeit relevante Version des Graphenlabors wird als *JGraLab* bezeichnet. Diese bietet im Gegensatz zur C++-Variante des Graphenlabors (*GraLab*) eine objektorientierte Zugriffsschicht auf die einzelnen Graphenelemente [Kah06].

Dieses Kapitel liefert einen Überblick über die für diese Diplomarbeit relevanten Aspekte der Implementierung der aktuellen JGraLab-Version *Carnotaurus*[2].

2.1 Implementierung von JGraLab

Dieser Abschnitt behandelt die Implementierung von JGraLab. Hierbei werden nur die im Rahmen der Diplomarbeit relevanten Aspekte detaillierter erläutert. Zunächst wird eine Übersicht über die Paketstruktur im JGraLab geliefert (siehe Abschnitt 2.1.1). Anschließend wird auf die Umsetzung der einzelnen in Abschnitt 1.2.6.1 (Seite 8) definierten Ebenen eingegangen, hier vor allem mit dem Fokus auf der *Graphenebene* (siehe Abschnitt 2.1.3). Abschließend wird die interne Repräsentation von TGraphen im JGraLab analysiert (siehe Abschnitt 2.1.4).

2.1.1 Die Paketstruktur

JGraLab *Carnotaurus* weist die in Abbildung 2.1 dargestellte Paketstruktur auf[3]. Die Pakete *de.uni_koblenz.jgralab.utilities* und *de.uni_koblenz.jgralab.xmlrpc* spielen im Kontext der Diplomarbeit keine Rolle und werden nicht näher erläutert. Das Paket *de.uni_koblenz.jgralab.greql2* ist für die Konsistenzüberprüfung relevant (siehe Abschnitt 4.4).

[2] `http://userpages.uni-koblenz.de/~ist/JGraLab_Download#JGraLab_Carnotaurus` (abgerufen am 4. März 2009)

[3] `http://userpages.uni-koblenz.de/~ist/jgralab/api/` (abgerufen am 4. März 2009)

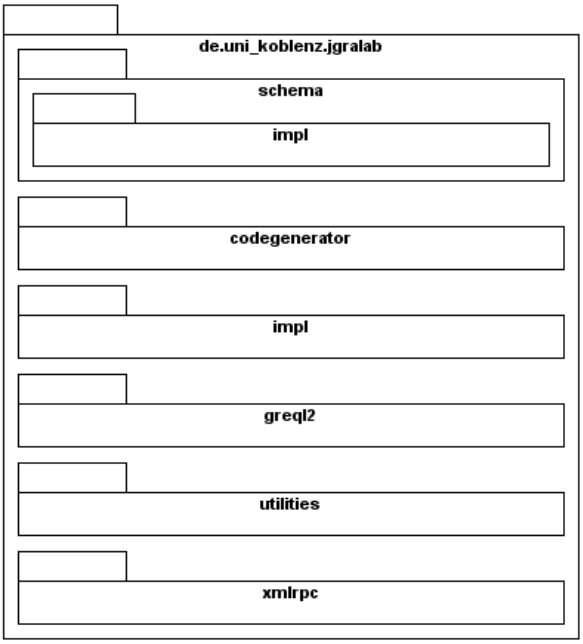


Abbildung 2.1: UML-Paketdiagramm für JGraLab

2.1.2 Implementation der Meta-Schema- und Schema-Ebene

Die Meta-Schema- und Schema-Ebene werden durch die Java-Pakete *de.uni_koblenz.jgralab.schema* und *de.uni_koblenz.jgralab.schema.impl* umgesetzt. Hier wird eine Trennung zwischen den Interfaces (in *de.uni_koblenz.jgralab.schema*) und der eigentlichen Implementierung (in *de.uni_koblenz.jgralab.schema.impl*) vorgenommen. Die Anwender von JGraLab sehen dabei lediglich die in den Interfaces definierte Funktionalität; die Details der eigentlichen Implementierung bleiben verborgen.

Das Paket *de.uni_koblenz.jgralab.schema* setzt das für TGraphen gültige Meta-Schema der *M3*-Ebene um und definiert die notwendige Funktionalität zur Erstellung von Schemata für konkrete TGraphen. Das Paket *de.uni_koblenz.jgralab.schema.impl* bietet dem JGraLab-Anwender eine API an, mit der Schemata erstellt werden können, welche gültige Instanziierungen des Meta-Schemas darstellen. Die erstellten Schemata werden in dem eigens für das JGraLab entwickelte *.TG*-Format[4] persistent gespeichert [Kah06]. Alternativ kann der Anwender die Schemata mit einem Texteditor manuell erstellen.

Da das in dieser Diplomarbeit zu entwickelnde Transaktionskonzept nicht für die Schemaebene und somit für die Erstellung von Schemata umgesetzt werden soll, wird an dieser Stelle nicht auf die weiteren Details der Implementierung eingegangen.

[4] `http://userpages.uni-koblenz.de/~ist/TG_Format` (abgerufen am 4. März 2009)

2.1.3 Implementation der Graphenebene

Abbildung 2.2 stellt die Umsetzung der Graphenebene im Form eines UML-Klassendiagramms dar. Sie zeigt die Pakete *de.uni_koblenz.jgralab* und *de.uni_koblenz.jgralab.impl*. Es handelt sich hierbei um eine vereinfachte, unvollständige und auf das Wesentliche beschränkte Darstellung der Implementationsdetails der Graphenebene.

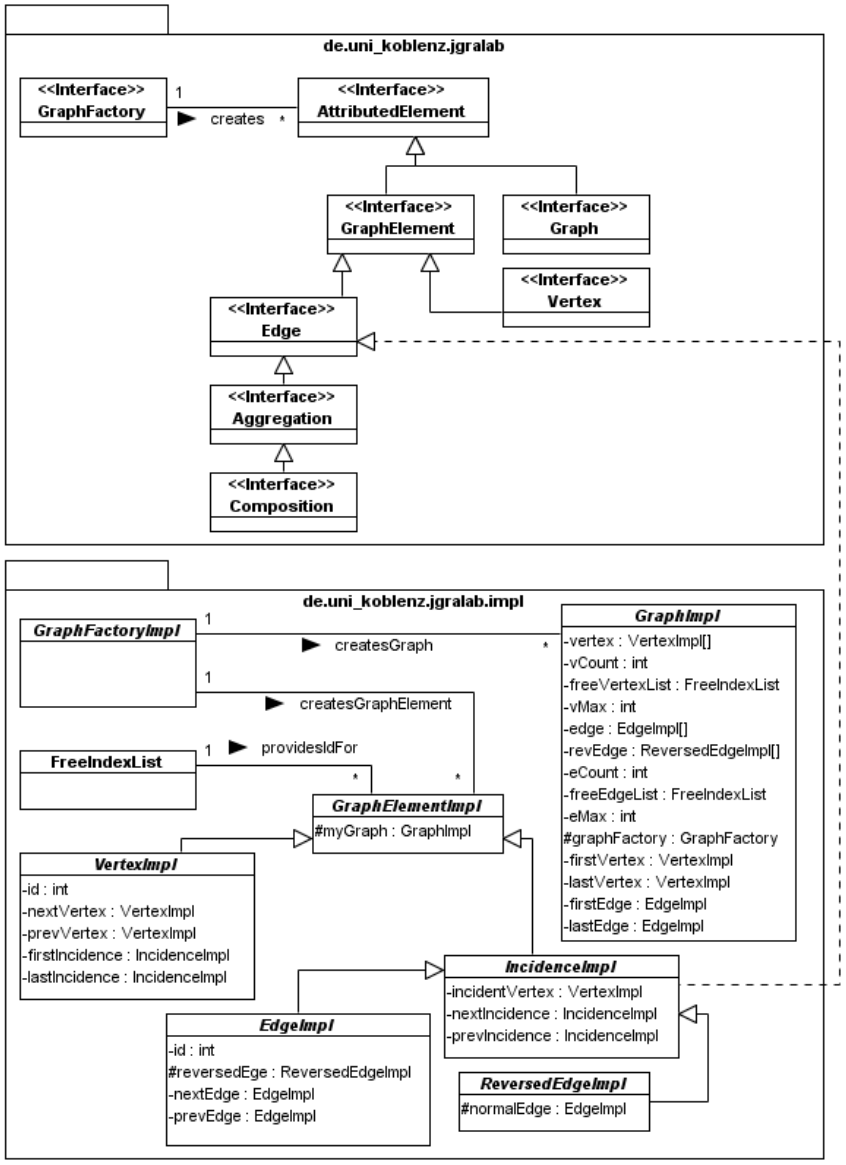


Abbildung 2.2: UML-Klassendiagramm zur Umsetzung der Graphenebene

2.1.3.1 Definition der Interfaces

Im Paket *de.uni_koblenz.jgralab* erfolgt die Definition der Interfaces. Ein Graph (*Graph*) und ein Graphenelement (*Graphelement*) stellen Elemente dar, die Attribute besitzen können (*AttributedElement*). Ein Graphenelement kann sowohl eine Kante (*Edge*) als auch ein Knoten (*Vertex*) sein. Zudem werden die Aggregations- (*Aggregation*) und die Kompositionskante (*Composition*) umgesetzt.

Das definierte Interface *GraphFactory* wird in Abschnitt 2.3 erläutert.

2.1.3.2 Die Implementationsklassen

Das Paket *de.uni_koblenz.jgralab.impl* enthält die Implementationsklassen der Graphenebene. In Abbildung 2.2 wird die explizite Beziehungsangabe (gestrichelte Linie mit Pfeilspitze) zwischen den Interfaces und den zugehörigen Implementationsklassen aus Übersichtlichkeitsgründen nur dann angegeben, wenn diese nicht durch die Namensgebungen der jeweiligen Interfaces und Klassen offensichtlich erscheint. So implementieren beispielsweise die Klassen *EdgeImpl* und *ReversedEdgeImpl* das Interface *Edge*.

Für die (leeren) Interfaces *Aggregation* und *Composition* existieren keine zugehörigen Implementationsklassen im Paket *de.uni_koblenz.jgralab.impl*. *Aggregation* und *Composition* werden als *Marker-Interfaces* verwendet. Diese werden von entsprechenden (aus einem Schema generierten) Kantenklassen implementiert (siehe Abschnitt 2.2), um diese als Aggregations- bzw. Kompositionskanten identifizieren zu können.

In der Implementation der Graphenebene treten weitere Elemente auf, die nicht (bereits) durch die Definition der Interfaces eingeführt werden. So repräsentiert die Klasse *ReversedEdgeImpl* eine Kante in ihrer umgekehrten Richtung. Diese enthält mit dem Attribut *normalEdge* vom Typ *EdgeImpl* eine Referenz auf die entsprechende Kante mit normaler Ausrichtung. Die Klasse *EdgeImpl* besitzt ihrerseits mit dem Attribut *reversedEdge* vom Typ *ReversedEdgeImpl* eine Referenz auf die zugehörige Kante in umgekehrter Richtung.

Sowohl eine Kante mit normaler Ausrichtung (*EdgeImpl*) als auch ihre entsprechende Darstellung in umgekehrter Richtung (*ReversedEdgeImpl*) werden als *Inzidenzen* modelliert. Eine Inzidenz wird durch die Klasse *IncidenceImpl* repräsentiert. Jede Inzidenz besitzt durch das Attribut *incidentVertex* vom Typ *VertexImpl* eine Referenz auf den zugehörigen (verbundenen) Knoten. Für einen Knoten v stellt eine mit v verbundene Kante e_1 vom Typ *EdgeImpl* eine Inzidenz mit einer *ausgehenden*, eine mit v verbundene Kante e_2 vom Typ *ReversedEdgeImpl* eine Inzidenz mit einer *eingehenden* Kante dar.

Jedes Graphenelement hält durch das Attribut *myGraph* des Typs *GraphImpl* eine Referenz auf den zugehörigen Graphen.

Die Klassen *GraphFactoryImpl* und *FreeIndexList* werden in Abschnitt 2.3 bzw. in Abschnitt 2.5 erläutert.

2.1.4 Interne Repräsentation von TGraphen

Im Folgenden wird die interne Repräsentation von TGraphen im JGraLab erläutert.

2.1.4.1 Die Menge V

Die Menge V wird durch das Array *vertex* in der Klasse *GraphImpl* umgesetzt. Es enthält Referenzen auf alle zu einem Zeitpunkt gültigen Knoteninstanzen. Dabei entspricht der Arrayindex der jeweiligen Knoten-ID. Hier sei vorab erwähnt, dass im JGralab jede Knoteninstanz jeweils eine eindeutige ID > 0 besitzen muss, so dass zu einem Zeitpunkt keine zwei Knoteninstanzen mit gleicher ID existieren dürfen. Das *int*-Feld *vCount* in der Klasse *GraphImpl* speichert die aktuelle Knotenanzahl im Graphen.

2.1.4.2 Die Menge E

Die Arrays *edge* und *revEdge* in *GraphImpl* setzen die Menge E um und enthalten Referenzen auf alle zu einem Zeitpunkt gültigen Kanteninstanzen in normaler bzw. in umgekehrter Richtung. Der Arrayindex entspricht hier (dem absoluten Wert) der jeweiligen Kanten-ID. Die Eindeutigkeit von Kanten-IDs gilt analog wie bei den Knoteninstanzen. Anzumerken bleibt, dass eine Instanz der Klasse *ReversedEdgeImpl* die gleiche ID wie die zugehörige Instanz der Klasse *EdgeImpl* besitzt, allerdings mit zusätzlichem negativem Vorzeichen. Das *int*-Feld *eCount* in *GraphImpl* speichert die aktuelle Kantenanzahl im Graphen.

2.1.4.3 Die Anordnung der Knoten Vseq

Die Informationen zu *Vseq* verteilen sich in mehreren Klassen. So enthält die Klasse *GraphImpl* mit den Attributen *firstVertex* und *lastVertex* vom Typ *VertexImpl* Referenzen auf den ersten bzw. letzten Knoten in *Vseq*. Jeder Knoten kennt zudem seinen jeweiligen Nachfolger- bzw. Vorgängerknoten in *Vseq*. Hierfür besitzt die Klasse *VertexImpl* die Attribute *nextVertex* bzw. *prevVertex* vom Typ *VertexImpl*.

2.1.4.4 Die Anordnung der Kanten Eseq

Ähnlich verhält es sich bei *Eseq*. In der Klasse *GraphImpl* werden durch die Attribute *firstEdge* und *lastEdge* vom Typ *EdgeImpl* Referenzen auf die erste bzw. die letzte Kante in *Eseq* gehalten. Zusätzlich besitzt die Klasse *EdgeImpl* mit den Attributen *nextEdge* und *prevEdge* Referenzen auf die jeweilige Nachfolge- bzw. Vorgängerkante in *Eseq*.

2.1.4.5 Die Inzidenzliste Λseq(v) eines Knotens *v*

Die Klasse *VertexImpl* hält durch die Attribute *firstIncidence* und *lastIncidence* vom Typ *IncidenceImpl* Referenzen auf die erste bzw. die letzte Inzidenz der zugehörigen Inzidenzliste. Jede Inzidenz kennt zudem seine Nachfolge- und Vorgängerinzidenz in der zugehörigen Inzidenzliste Λ*seq*(v) eines Knotens *v*. Hierfür besitzt die Klasse *IncidenceImpl* die Attribute *nextIncidence* bzw. *prevIncidence* vom Typ *IncidenceImpl*.

2.2 Automatisierte Generierung der objektorientierten Zugriffsschicht

Das Paket *de.uni_koblenz.jgralab.impl* beinhaltet die Implementation für die Erstellung und Manipulation von Graphen und Graphenelementen unabhängig von den in einem Schema definierten Graphen-, Knoten- und Kantenklassen. Da die jeweiligen Klassen jedoch als *abstrakt* deklariert sind, lassen sich diese nicht direkt instanziieren. Damit ein Anwender von JGraLab auf TGraphen einer speziellen Graphenklasse arbeiten kann, muss daher zunächst eine (nur für die speziellen TGraphen verwendbare) *objektorientierte Zugriffsschicht* aus dem zugehörigen Schema generiert werden. Die Funktionalität zur Codegenerierung wird im Java-Paket *de.uni_koblenz.jgralab.codegenerator* umgesetzt.

Abbildung 2.3 zeigt die aus dem Schema *MotorwayMapSchema* (siehe Abbildung 1.4, Seite 11) generierten Pakete *de.uni_koblenz.motorwaymap* und *de.uni_koblenz.motorwaymap.impl* und deren Beziehungen zu den in Abbildung 2.2 aufgeführten Paketen.

2.2.1 Die generierten Interfaces

Sowohl für die im Schema *MotorwayMapSchema* definierte Graphenklasse (*MotorwayMap*) als auch für die auftretenden Knoten- (*City*, *Motorway*, *StoppingPlace*, *MotorwayStation* und *MotorwayRestaurant*) und Kantenklassen (*HasExit* und *Offers*) werden entsprechende Interfaces im Paket *de.uni_koblenz.motorwaymap* generiert. Die Interfaces

werden von den zugehörigen Interfaces im Paket *de.uni_koblenz.jgralab* abgeleitet, also von den Interfaces *Graph*, *Vertex* und *Aggregation* bzw. *Composition*.

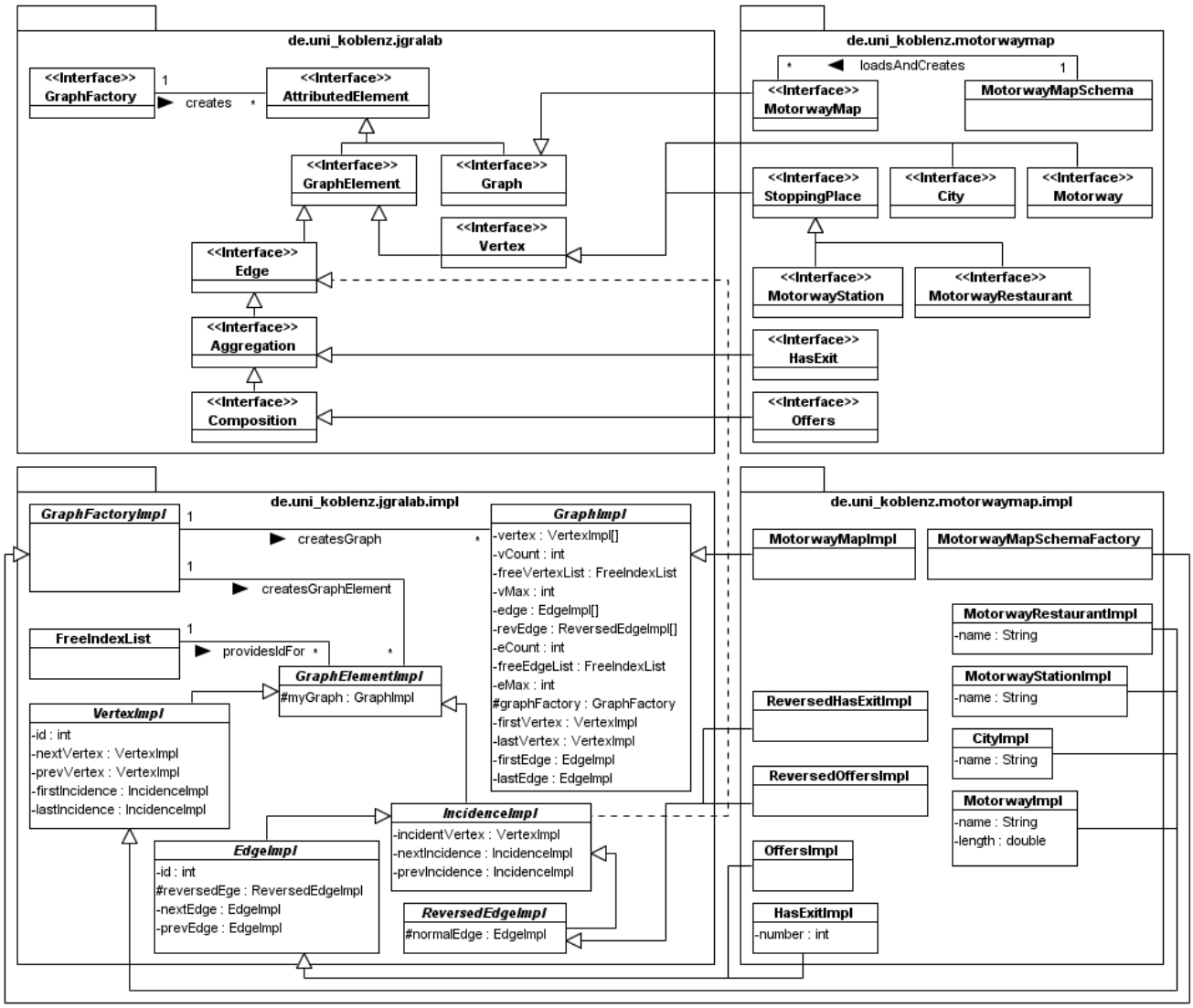


Abbildung 2.3: Erweitertes UML-Klassendiagramm mit generierten Paketen

2.2.2 Die generierten Implementationsklassen

Das Paket *de.uni_koblenz.motorwaymap.impl* enthält die konkreten (instanziierbaren) Klassen mit ihren entsprechenden Attributen, welche die jeweiligen generierten Interfaces aus dem Paket *de.uni_koblenz.motorwaymap* implementieren. Zu dem Interface der im Schema als abstrakt deklarierten Knotenklasse *StoppingPlace* wird keine zugehörige Im-

plementationsklasse generiert. Aufgrund der fehlenden Vererbungshierarchie wird das String-Attribut *name* jeweils direkt in die Unterklassen *MotorwayStationImpl* und *MotorwayRestaurantImpl* integriert. Entscheidend ist hierbei, dass die generierten Klassen von den im Paket *de.uni_koblenz.jgralab.impl* zugehörigen abstrakten Klassen abgeleitet sind. So sind die Graphenklasse *MotorwayMapImpl* von *GraphImpl*, die Knotenklassen *CityImpl*, *MotorwayImpl*, *MotorwayStationImpl* und *MotorwayRestaurantImpl* von *VertexImpl* und die Kantenklassen *HasExitImpl*, *OffersImpl*, *ReversedHasExitImpl* und *ReversedOffersImpl* von *EdgeImpl* bzw. *ReversedEdgeImpl* abgeleitet.

Die generierte objektorientierte Zugriffsschicht dient dem Zugriff und der Manipulation von TGraphen, welche in einem zugehörigen Schema definiert worden sind. Sie stellt sicher, dass nur die im Schema definierten (nicht abstrakten) Graphen-, Knoten- und Kantenklassen instanziiert werden können. Die Einhaltung der im Schema definierten Kardinalitäten wird dabei nicht sichergestellt.

2.2.3 Erzeugen und Laden von Graphen

Um neue Graphen vom Typ *MotorwayMap* zu erstellen oder bereits existierende zu laden, wird die Klasse *MotorwayMapSchema* aus dem Paket *de.uni_koblenz.motorwaymap* benötigt (siehe Abbildung 2.4). *MotorwayMapSchema* ist als *Singleton*-Klasse modelliert, dessen einzige Instanz (*theInstance*) nur mittels der statischen Methode *instance* ermittelt werden kann.

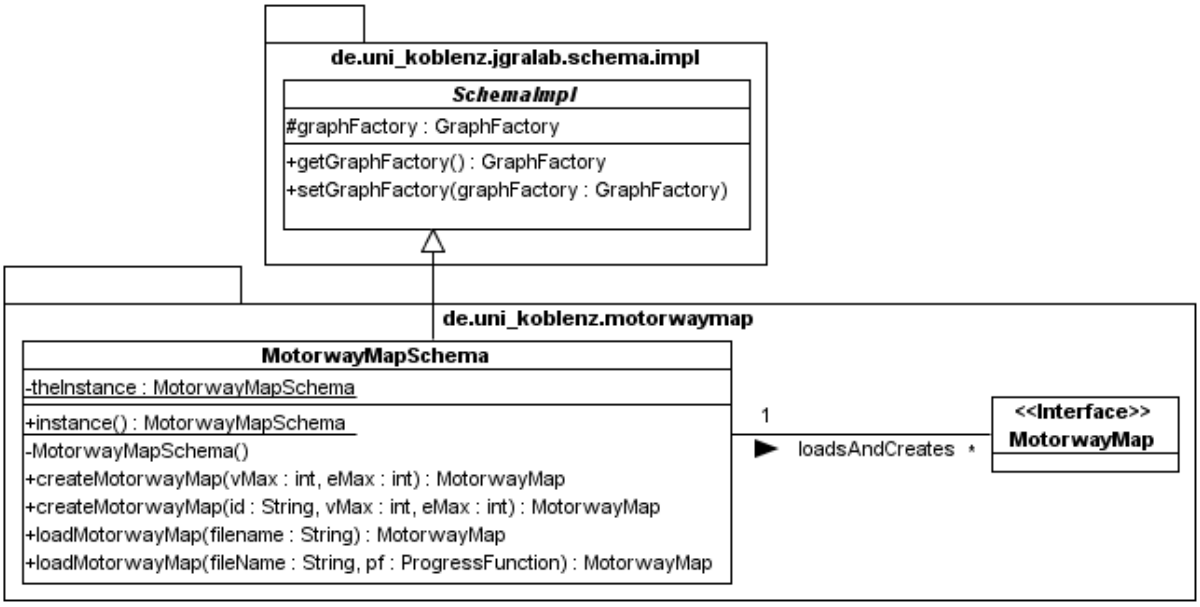


Abbildung 2.4: Die Klasse *MotorwayMapSchema*

Mit der Methode *createMotorwayMap* kann ein neuer Graph vom Typ *MotorwayMap* erzeugt werden. Die *int*-Parameter *vMax* und *eMax* geben die maximale Anzahl von Knoten bzw. Kanten an, die innerhalb des zu erzeugenden Graphen (zunächst) maximal gespeichert werden können.

Mit der Methode *loadMotorwayMap* kann ein bereits bestehender und in einer entsprechenden *.TG*-Datei gespeicherter Graph vom Typ *MotorwayMap* geladen werden.

2.3 Factory zur Instanziierung von Graphen und Graphenelementen

Für die Instanziierung von Graphen und Graphenelementen zur Laufzeit ist im JGraLab die sogenannte *GraphFactory* verantwortlich. Abbildung 2.5 zeigt dessen Umsetzung im Zusammenhang mit Graphen vom Typ *MotorwayMap*.

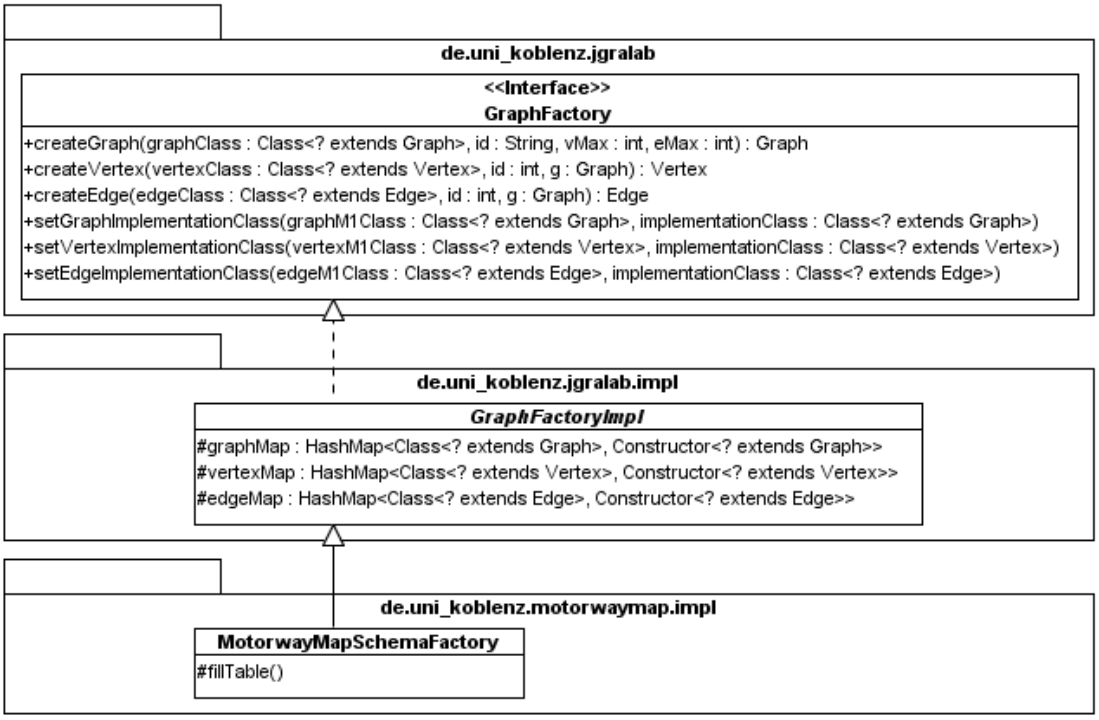


Abbildung 2.5: Die Umsetzung der *GraphFactory*

Das Interface *GraphFactory* definiert zum einen die Methoden *createGraph*, *createVertex* und *createEdge*, die zur Instanziierung von Graphen-, Knoten- bzw. Kanteninstanzen zur Laufzeit dienen.

Mit den Methoden *setGraphImplementationClass*, *setVertexImplementationClass* und *setEdgeImplementationClass* können die Zuordnungen zwischen den generierten Graphen-, Knoten- und Kanteninterfaces und den entsprechenden generierten Implementationsklassen festgelegt werden. So wird in der für Graphen des Typs *MotorwayMap* zugehörigen Factory-Implementierung *MotorwayMapSchemaFactory* innerhalb der Methode *fillTable* die zur Festlegung der gültigen Zuordnungen notwendigen Methodenaufrufe durchgeführt (siehe Listing 2.1). Dadurch wird beispielsweise festgelegt, dass für Knoten vom Typ *Motorway* die Klasse *MotorwayImpl* aus dem Paket *de.uni_koblenz.motorwaymap.impl* zur Instanziierung verwendet werden soll (Zeilen 12 - 14).

```
1  public class MotorwayMapSchemaFactory extends GraphFactoryImpl {
2
3    public MotorwayMapSchemaFactory() {
4      super();
5      fillTable();
6    }
7
8    protected void fillTable() {
9      setGraphImplementationClass(
10       de.uni_koblenz.motorwaymap.MotorwayMap.class,
11       de.uni_koblenz.motorwaymap.impl.MotorwayMapImpl.class);
12     setVertexImplementationClass(
13       de.uni_koblenz.motorwaymap.Motorway.class,
14       de.uni_koblenz.motorwaymap.impl.MotorwayImpl.class);
15     setVertexImplementationClass(
16       de.uni_koblenz.motorwaymap.MotorwayStation.class,
17       de.uni_koblenz.motorwaymap.impl.MotorwayStationImpl.class);
18     setVertexImplementationClass(
19       de.uni_koblenz.motorwaymap.MotorwayRestaurant.class,
20       de.uni_koblenz.motorwaymap.impl.MotorwayRestaurantImpl.class);
21     setVertexImplementationClass(
22       de.uni_koblenz.motorwaymap.City.class,
23       de.uni_koblenz.motorwaymap.impl.CityImpl.class);
24     setEdgeImplementationClass(
25       de.uni_koblenz.motorwaymap.HasExit.class,
26       de.uni_koblenz.motorwaymap.impl.HasExitImpl.class);
27     setEdgeImplementationClass(
28       de.uni_koblenz.motorwaymap.Offers.class,
29       de.uni_koblenz.motorwaymap.impl.OffersImpl.class);
30   }
31 }
```

Listing 2.1: Die Klasse *MotorwayMapSchemaFactory*

Die getätigten Zuordnungen werden in den *Maps graphMap*, *vertexMap* und *edgeMap* innerhalb der abstrakten Klasse *GraphFactoryImpl* gespeichert. Als Schlüssel dient das *Class*-Objekt des jeweiligen Graphen-, Knoten- bzw. Kanteninterfaces. Der zugehörige Wert repräsentiert das entsprechende *Constructor*-Objekt der Implementationsklasse. Zur Instanziierung der Graphen und Graphenelemente zur Laufzeit wird die *Java-Reflection-API* verwendet.

Die Klasse *MotorwayMapSchema* aus Abbildung 2.4 besitzt durch das Attribut *graphFactory* der Oberklasse *SchemaImpl* eine Referenz auf die zugehörige *GraphFactory*-Instanz. Jede Grapheninstanz erhält ihrerseits eine entsprechende Referenz auf die sie erzeugende *GraphFactory*-Instanz, welche mit Hilfe der Methode *getGraphFactory* der Klasse *SchemaImpl* ermittelt werden kann. Diese Referenz wird in dem Attribut *graphFactory* der Klasse *GraphImpl* (siehe Abbildung 2.2) gespeichert und zur Instanziierung von Graphenelementen zur Laufzeit für die jeweilige Grapheninstanz verwendet.

Durch die Methode *setGraphFactory* der Klasse *SchemaImpl* ist es möglich, eine andere (als die generierte und damit vorgegebene) *GraphFactory*-Implementationsklasse als zu verwendende *GraphFactory* festzulegen. So lassen sich in einer solchen (neuen) *GraphFactory*-Implementationsklasse andere (als die generierten und damit vorgegebenen) Implementationsklassen zur Instanziierung von Graphen und Graphenelementen angeben.

2.4 Beispiel zur Verwendung der generierten API

Listing 2.2 zeigt, wie mit der generierten API aus Abbildung 2.3 der Beispielgraph aus Abbildung 1.1 (Seite 4) erzeugt werden kann.

```
1  MotorwayMapSchema schema = MotorwayMapSchema.instance();
2  // create TGraph of class MotorwayMap with maximum 5 vertices and 4 edges
3  MotorwayMap motorwayMap = schema.createMotorwayMap("motorwaymap", 5, 4);
4
5  // create vertex v1 : Motorway
6  Motorway motorway = motorwayMap.createMotorway();
7  motorway.setLength(778);
8  motorway.setName("A3");
9
10 // create vertex v2 : City
11 City city1 = motorwayMap.createCity();
12 city1.setName("Montabaur");
13 //create vertex v3 : City
14 City city2 = motorwayMap.createCity();
15 city2.setName("Limburg");
16
17 // create edge e1 : HasExit
18 HasExit hasExit1 = motorwayMap.createHasExit(city1, motorway);
19 hasExit1.setNumber(22);
20 // create edge e2 : HasExit
21 HasExit hasExit2 = motorwayMap.createHasExit(city2, motorway);
22 hasExit2.setNumber(25);
```

```
23
24 // create vertex v4 : MotorwayStation
25 MotorwayStation motorwayStation = motorwayMap.createMotorwayStation();
26 motorwayStation.setName("Am Weiher");
27 // create vertex v5 : MotorwayRestaurant
28 MotorwayRestaurant motorwayRestaurant =
29         motorwayMap.createMotorwayRestaurant();
30 motorwayRestaurant.setName("Raststätte Montabaur");
31
32 // create edge e3 : Offers
33 Offers offers1 = motorwayMap.createOffers(motorway, motorwayStation);
34 // create edge e4 : Offers
35 Offers offers2 = motorwayMap.createOffers(motorway, motorwayRestaurant);
```

Listing 2.2: Erzeugung des Beispielgraphen mit der generierten API

2.5 Freispeicherliste für Knoten und Kanten

Werden einem Graphen (zur Laufzeit) neue Graphenelemente hinzugefügt, muss die Vergabe von jeweils eindeutigen IDs für Knoten und Kanten vom System gewährleistet werden. Die Verwaltung dieser IDs wird in sogenannten *Freispeicherlisten* vorgenommen. Diese werden beim Erstellen oder Laden eines Graphen automatisch initialisiert und beim Hinzufügen und Löschen von Graphenelementen entsprechend aktualisiert.

2.5.1 Die Klasse FreeIndexList

Abbildung 2.6 zeigt die Klasse *FreeIndexList*, welche eine Freispeicherliste umsetzt. Die Klasse *GraphImpl* (siehe Abbildung 2.2) besitzt die Attribute *freeVertexList* und *freeEdgeList* vom Typ *FreeIndexList*, welche die Freispeicherliste der Knoten bzw. der Kanten für den jeweiligen Graphen repräsentieren.

Beim *Hinzufügen* von neuen Graphenelementen in einem Graphen wird mit Hilfe der Methode *allocateIndex* die nächste freie ID ermittelt, welche dem jeweiligen Graphenelement zugeordnet wird. Beim *Löschen* von existierenden Graphenelementen wird die jeweilige ID durch die Übergabe als Parameter (*index*) an die Methode *freeIndex* (zur erneuten Vergabe) wieder freigegeben.

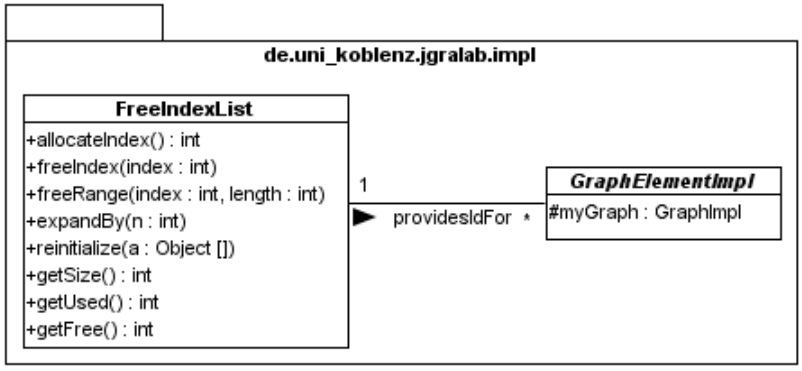


Abbildung 2.6: Die Klasse *FreeIndexList*

2.5.2 Vergrößerung der Graphenkapazitäten

Wie bereits in Abschnitt 2.2.3 erläutert, wird bei der Erzeugung einer neuen Grapheninstanz eine maximale Anzahl von Knoten und Kanten angegeben, die (zunächst) in dem jeweiligen Graphen gespeichert werden können. Die zu einem Zeitpunkt gültige maximale Anzahl von Knoten und Kanten wird in den Feldern *vMax* bzw. *eMax* der Klasse *GraphImpl* gespeichert (siehe Abbildung 2.2, Seite 15). Für den Fall, dass die maximale Anzahl von Knoten in einem Graphen bereits erreicht wurde und ein neuer Knoten hinzugefügt werden soll, müssen die *Kapazitäten* zur Aufnahme von Knoten in dem zugehörigen Graphen erweitert werden. Analoges gilt für die Kanten.

So liefert die Methode *allocateIndex* den (für die ID eines Graphenelements ungültigen) Wert 0, falls (je nach Kontext) keine weiteren Knoten oder Kanten (ohne Kapazitätserweiterungen) hinzugefügt werden können. Tritt diese Situation beispielsweise für die Knoten auf, müssen das Array *vertex* und die Freispeicherliste der Knoten *freeVertexList* in der Klasse *GraphImpl* um einen (intern mit Hilfe von *vMax* berechneten) Faktor einheitlich erweitert werden. Die Erweiterung der Freispeicherliste wird durch die Methode *expandBy* angestossen, welcher als Parameter (*n*) der entsprechende Erweiterungsfaktor übergeben wird. Nach der Kapazitätsvergrößerung liefert die Methode *allocateIndex* wieder eine gültige ID > 0. Analoges gilt für die Kanten und somit für die Arrays *edge* und *revEdge* und die Freispeicherliste der Kanten *freeEdgeList* in der Klasse *GraphImpl*.

2.6 Persistente Speicherung von TGraphen

Zur Laufzeit werden die TGraphen im Hauptspeicher (*Java-Memory*) abgebildet. Beim Neustart des Systems sind alle relevanten Informationen (zunächst) nicht mehr im Hauptspeicher vorhanden. Daher können TGraphen im JGraLab (wie Schemata auch) zur Lauf-

zeit *persistent* in Dateien auf dem Hintergrundspeicher gespeichert werden. Diese Dateien entsprechen ebenfalls dem *.TG*-Format. Die persistente Speicherung muss dabei vom Anwender von JGraLab angestossen werden. Zur Laufzeit besteht die Möglichkeit, die in *.TG*-Dateien abgespeicherten TGraphen wieder in den Hauptspeicher zu laden.

2.7 Einschränkung auf Einprozessbetrieb

In der aktuellen Version des JGraLabs ist die korrekte (koordinierte und konfliktfreie) Durchführung von Änderungen an einem TGraphen nur für den Fall garantiert, dass zu einem gegebenen Zeitpunkt höchstens ein Prozess gleichzeitig aktiv ist. Demnach wird nur der *Einprozessbetrieb* unterstützt. Im Rahmen dieser Diplomarbeit soll ein *Mehrprozessbetrieb* entworfen und umgesetzt werden, welcher es erlaubt, dass mehrere Prozesse gleichzeitig (parallel) und koordiniert Änderungen an einem TGraphen durchführen können. Der *Mehrprozessbetrieb* soll dabei im Rahmen eines *Transaktionskonzepts* ablaufen.

3 ACID-Transaktionen

Bevor ein Transaktionskonzept für das Graphenlabor konzipiert werden kann, müssen zunächst die Grundlagen im Zusammenhang mit Transaktionen erläutert werden. Transaktionen finden in vielen Bereichen der Informatik Verwendung (zum Beispiel in Betriebssystemen und in der Programmierung). Erforscht wurden diese aber vor allem im Kontext der Datenbanken [KE04]. Die folgenden Abschnitte orientieren sich aus diesem Grund an den Ergebnissen der Datenbankforschung.

Zunächst werden die grundlegenden Begriffe und Eigenschaften von *ACID*-Transaktionen erläutert, um ein allgemeines Verständnis für Transaktionen zu vermitteln. Anschließend werden Mechanismen vorgestellt, welche die im *ACID*-Paradigma (siehe Abschnitt 3.1) definierten Eigenschaften für Transaktionen sicherstellen sollen. Hier wird speziell auf die *Fehlerbehandlung* (siehe Abschnitt 3.5) und die *Synchronisation* (siehe Abschnitt 3.6) im Kontext von Transaktionen eingegangen.

3.1 Das ACID-Paradigma

Das *ACID-Paradigma* legt vier grundlegende Eigenschaften fest, welche von jeder in einem System auszuführenden Transaktion erfüllt sein müssen [KE04]:

Atomicity (Atomarität) — Diese Eigenschaft besagt, dass entweder alle durch eine Transaktion durchgeführten Änderungen in den Daten gespeichert werden oder gar keine davon („Alles-oder-nichts“-Prinzip). Eine Transaktion wird somit als kleinste, nicht weiter zerlegbare Einheit betrachtet.

Consistency (Konsistenz) — Es wird gefordert, dass eine Transaktion zu Beginn ihrer Ausführung einen konsistenten Datenzustand vorfindet. Nachdem die Transaktion beendet wurde, muss ebenfalls ein konsistenter Datenzustand vorherrschen. Die Zwischenzustände während der Transaktionsausführung dürfen durchaus inkonsistent sein. Entscheidend ist letztlich nur, dass der durch die Transaktionsausführung herbeigeführte Endzustand alle definierten (systemabhängigen) Konsistenzbedingungen erfüllt. Ist dies nicht der Fall, muss die Transaktion in der Regel nach dem „ Alles-oder-nichts“-Prinzip komplett zurückgesetzt werden.

Isolation — Diese Eigenschaft legt fest, dass sich mehrere parallel ablaufende Transaktionen nicht gegenseitig beeinflussen dürfen. Eine Transaktion muss (aus logischer Sicht) unabhängig von allen anderen aktiven Transaktionen ausgeführt werden. Es soll demnach simuliert werden, dass eine Transaktion während ihrer Ausführung die einzige zu einem Zeitpunkt im System aktive Transaktion darstellt.

Durability (Dauerhaftigkeit oder Persistenz) — Die Änderungen einer erfolgreich abgeschlossenen Transaktion müssen (auch nach einem System- oder Hardwarefehler) dauerhaft in den Daten erhalten bleiben.

3.2 Flache Transaktionen

Nachdem die Eigenschaften von Transaktionen durch das *ACID*-Paradigma festgelegt wurden, werden im Folgenden die sogenannten *flachen Transaktionen* erläutert.

3.2.1 Formale Definition von flachen Transaktionen

Definition 3.1 (Flache Transaktion)

Seien

- D das Universum der **unteilbaren und disjunkten Datenobjekte**,
- OP das Universum der **Operationen** und
- $IDENT$ das Universum der **Bezeicher**.

Ferner sei

- $TOP \subseteq OP$ eine endliche Menge von **Transaktionsoperationen** mit $TOP = \{BOT, COMMIT, ABORT, r(x), w(x), dsp(m), rsp(m)\}$ für $x \in D \wedge m \in IDENT$.

Dann ist eine **flache Transaktion** T definiert als eine **Sequenz von Transaktionsoperationen** $T = <op_0, op_1, ..., op_n>$ mit

- $n < \infty$,
- $op_0 = BOT$,
- $op_i \in \{r(x), w(x), dsp(m), rsp(m)\}$ für $1 \leq i < n$ und
- $op_n \in \{COMMIT, ABORT\}$

Definition 3.1 zeigt die formale Definition einer *flachen Transaktion* nach [Vos00]. Diese ist notwendig, da alle in diesem Kapitel im Folgenden angesprochenen Aspekte auf flachen Transaktionen basieren werden. Der Begriff *flach* soll dabei ausdrücken, dass eine Transaktion selbst keine weiteren (Sub-)Transaktionen beinhalten kann. Die Definition wird nachfolgend vor allem im Hinblick auf ihre Semantik erläutert. Zur Veranschaulichung dient die Beispieltransaktion

T_1 = <BOT, r(x), w(x), dsp(1), r(y), w(y), rsp(1), r(z), w(z), COMMIT>.

Zunächst wird das Universum *D* eingeführt. Angelehnt an das *Read-Write-Modell* [Vos00] erfolgt hiermit eine Abstraktion von dem zugrundeliegenden Datenmodell mit Hilfe von *unteilbaren* und *disjunkten* Datenobjekten, auf denen nur Lese- und Schreiboperationen *op* ∈ *OP* ausgeführt werden können. Dies ermöglicht die Einführung einer allgemein gültigen und vom Datenmodell unabhängig verwendbaren Sichtweise auf Transaktionen[5]. Zur allgemeinen Erläuterung von (ACID-)Transaktionen wird das Read-Write-Modell im Rahmen dieses Kapitels als zugrundeliegendes Modell angenommen.

Zur Vergabe von eindeutigen Bezeichnern für Transaktionen und für *Sicherungspunkte* innerhalb einer Transaktion wird das Universum *IDENT* definiert. Sicherungspunkte werden im späteren Verlauf dieses Abschnitts erläutert.

Die Menge *TOP* definiert die innerhalb einer flachen Transaktion ausführbaren Operationen.

Eine flache Transaktion T_i besteht formal aus einer Sequenz von Operationen. Die erste Operation op_0 von T_i ist immer *BOT* (Begin Of Transaction). Diese signalisiert, dass eine neue Transaktion T_i gestartet wird und alle nun folgenden Operationen im Kontext von T_i ausgeführt werden. Jede Transaktion erhält bei ihrem Start einen im System eindeutigen Bezeichner *t* ∈ *IDENT*. In T_1 markiert die erste Operation *BOT* den Transaktionsbeginn. *1* stellt dabei die Transaktions-ID dar.

Operationen zwischen der ersten (op_0) und der letzten Operation (op_n) einer Transaktion können einerseits entweder eine Lese- (*r(x)*, r für *read*) oder eine Schreiboperation (*w(x)*, w für *write*) auf einem Datenobjekt *x* ∈ *D* darstellen.

Aufbauend auf der Definition 3.1 werden die Mengen *rSet* und *wSet* für jede Transaktion T_i eingeführt. Die Menge *rSet* enthält alle Datenobjekte *x* ∈ D, für die *r(x)* ∈ ran T_i gilt. Die Menge *wSet* enthält analog alle Datenobjekte *x* ∈ D, für die *w(x)* ∈ ran T_i gilt. Treten in einer Transaktion T_i Lese- aber keine Schreiboperationen auf (ist wSet = {}), wird T_i als *Read-Only*-Transaktion bezeichnet. Sobald mindestens eine Schreiboperation ausgeführt wird, ist T_i eine sogenannte *Read-Write*-Transaktion (wSet ≠ {}). In T_1 wird

[5]Diese Sicht ist im Kontext des Graphenlabors in dieser vereinfachten Form nicht anwendbar und wird daher in Kapitel 4 revidiert und erweitert (siehe Abschnitt 4.1.3.1).

in der 2. Operation eine Leseoperation auf dem Datenobjekt *x* ausgeführt. Im Folgenden wird davon ausgegangen, dass beim Einlesen eines Datenobjekts durch eine Transaktion T_i der zugehörige Wert in einer (nur für T_i gültigen) lokalen Variablen gespeichert wird. Im 3. Schritt führt T_1 eine Schreiboperation auf dem gleichen Datenobjekt *x* aus. Hier wird also der Wert einer lokalen Variablen als neuer Wert für *x* gespeichert. Analog dazu werden im 5. und 6. Schritt eine Lese- bzw. Schreiboperation auf dem Datenobjekt *y* ausgeführt, während die 8. und 9. Operation eine Lese- bzw. Schreiboperation auf dem Datenobjekt *z* darstellen. Zudem gilt rSet = {x,y,z} und wSet = {x,y,z}. T_1 ist eine *Read-Write*-Transaktion.

Die Operationen *dsp(m) und rsp(m)* erweitern das klassische Konzept der flachen Transaktionen um das *Konzept der Sicherungspunkte*, deren Verwendung vor allem bei Transaktionen von langer Dauer sinnvoll sein können [KE04]. Mit der Operation *dsp*(m) (define savepoint) kann während der Ausführung einer Transaktion T_i jederzeit ein (lokaler) Sicherungspunkt für T_i angelegt werden. Ein Sicherungspunkt „merkt“ sich alle bis dahin durchgeführten Operationen und somit den derzeit gültigen Datenzustand für die zugehörige Transaktion. Jedem Sicherungspunkt wird eine eindeutige ID $m \in IDENT$ zugewiesen. Möchte man während der Ausführung einer Transaktion T_i zu einem existierenden Sicherungspunkt zurückspringen und den damit verbundenen Zustand für T_i wiederherstellen, muss die Operation *rsp*(m) (restore savepoint) verwendet werden. *m* ist dabei die (gültige) ID eines zuvor angelegten Sicherungspunkts. Wird eine ungültige ID angegeben, wird die Operation ignoriert und übersprungen. In T_1 wird als 4. Operation ein Sicherungspunkt mittels *dsp(1)* angelegt. Der Sicherungspunkt erhält die ID *1*. Dieser repräsentiert den bis dahin gültigen Datenzustand. Im 8. Schritt wird *rsp(1)* ausgeführt, wodurch T_1 wieder auf den Datenzustand zurückgesetzt werden muss, welcher mit dem Sicherungspunkt *1* verbunden ist. Anders ausgedrückt, muss die im 6. Schritt ausgeführte Schreiboperation *w(y)* (in ihrer Wirkung) rückgängig gemacht werden.

Die letzte Operation op_n einer Transaktion T_i ist entweder die *COMMIT*- oder die *ABORT*-Operation. Erstere signalisiert, dass alle Datenänderungen von T_i dauerhaft gespeichert werden sollen. Durch die *ABORT*-Operation wird T_i abgebrochen und komplett zurückgesetzt (rückgängig gemacht). Das komplette Zurücksetzen einer Transaktion wird auch als *Rollback* bezeichnet. In T_1 ist *COMMIT* die letzte Operation.

Gilt für eine Transaktion T_i, dass die letzte Operation $op_n \in$ {COMMIT, ABORT} ist, so wird T_i als *vollständig* bezeichnet. Ist dies nicht der Fall, wird T_i als *unvollständige* Transaktion betrachtet. Eine *unvollständige* Transaktion kann beispielsweise durch einen unerwarteten Abbruch der Transaktion aufgrund eines Systemabsturzes entstehen. T_1 ist eine vollständige Transaktion.

3.2.2 Notation von flachen Transaktionen

In diesem Abschnitt werden zwei (alternative) Notationen für flache Transaktionen eingeführt, welche je nach Kontext eine übersichtlichere Darstellung von (parallel ablaufenden) Transaktionen erlauben. Die erste sieht für die Transaktion T_1 wiefolgt aus:

$T_1 = BOT_1 \rightarrow r_1(x) \rightarrow w_1(x) \rightarrow dsp_1(1) \rightarrow r_1(y) \rightarrow w_1(y) \rightarrow rsp_1(1) \rightarrow r_1(z) \rightarrow w_1(z) \rightarrow COMMIT_1$

Dabei werden Pfeile verwendet, die den Übergang einer Operation zur nächsten Operation symbolisieren. Zudem werden an die einzelnen Operationen Indizes notiert, welche mit dem Indexwert der zugehörigen Transaktion übereinstimmen.

Eine zweite Notation in Tabelle 3.1 stellt die Transaktion T_1 in Tabellenform dar. In der linken Spalte wird die Schrittzahl notiert. Die zweite Spalte enthält als Überschrift den Namen der Transaktion (T_1) und in den nachfolgenden Zeilen die zu den Schrittzahlen in der linken Spalte zugehörigen Operationen.

Schritt	T_1
1.	BOT
2.	r(x)
3.	w(x)
4.	dsp(1)
5.	r(y)
6.	w(y)
7.	rsp(1)
8.	r(z)
9.	w(z)
10.	COMMIT

Tabelle 3.1: Notation einer flachen Transaktion in Tabellenform

3.2.3 Mögliche Abschlüsse einer Transaktion

Nachfolgend werden die Varianten für den Abschluss einer Transaktion erläutert [KE04].

3.2.3.1 Erfolgreicher Abschluss durch ein COMMIT

Eine Transaktion T_i wird erfolgreich abgeschlossen, falls seine letzte Operation *COMMIT* ist und keine Konsistenzverletzungen durch die in T_i ausgeführten Operationen auf-

treten. Die in T_i durchgeführten Änderungen an Datenobjekten $x \in D$ werden dauerhaft gespeichert.

Geht man für die Transaktion T_1 in Tabelle 3.1 davon aus, dass nach dem *COMMIT* keine Konsistenzverletzungen festgestellt werden, stellt T_1 ein Beispiel für eine erfolgreich abgeschlossene Transaktion dar. Die Änderungen an den Datenobjekten *x*, *y* und *z* werden dauerhaft gespeichert.

3.2.3.2 Erfolgloser Abschluss nach einem COMMIT

Wenn jedoch nach dem *COMMIT* einer Transaktion T_i Konsistenzverletzungen auftreten, muss T_i abgebrochen und nach dem ACID-Paradigma (siehe Abschnitt 3.1) komplett zurückgesetzt werden. Das *Rollback* impliziert, dass die durch T_i durchgeführten Änderungen rückgängig gemacht werden müssen.

Geht man für die Transaktion T_1 in Tabelle 3.1 davon aus, dass nach dem *COMMIT* Konsistenzverletzungen festgestellt werden, stellt T_1 ein Beispiel für eine erfolglos abgeschlossene Transaktion nach einem COMMIT durch Konsistenzverletzungen dar. Die Änderungen an den Datenobjekten *x*, *y* und *z* müssen rückgängig gemacht werden.

3.2.3.3 Erfolgloser Abschluss durch ein ABORT

Eine Transaktion kann jederzeit mittels der *ABORT*-Operation abgebrochen werden, wodurch ein (explizites) *Rollback* der Transaktion ausgelöst wird.

Schritt	T_2
1.	BOT
2.	r(x)
3.	w(x)
4.	dsp(1)
5.	r(y)
6.	w(y)
7.	rsp(1)
8.	r(z)
9.	w(z)
10.	ABORT

Tabelle 3.2: Erfolgloser Abschluss einer Transaktion nach einem ABORT

Tabelle 3.2 zeigt ein zugehöriges Beispiel einer Transaktion T_2. Die Änderungen an den Datenobjekten *x*, *y* und *z* müssen rückgängig gemacht werden.

3.2.3.4 Erfolgloser Abschluss durch einen Systemfehler

Während der Ausführung einer Transaktion T_i können *Systemfehler* (Software- oder Hardwarefehler) auftreten, die einen Systemabsturz hervorrufen. *Systemfehler* sei dabei eine Pseudo-Operation *pop* ∈ *OP*. Beim Neustart des Systems muss sichergestellt sein, dass die durch die erfolglose und unvollständige Transaktion T_i durchgeführten Änderungen nicht in den jeweiligen Datenobjekten übernommen wurden.

Schritt	T_3
1.	BOT
2.	r(x)
3.	w(x)
4.	**Systemfehler**

Tabelle 3.3: Erfolgloser Abschluss einer Transaktion nach einem Systemfehler

In Tabelle 3.3 tritt nach der Ausführung der 3. Operation *w(x)* der Transaktion T_3 im 4. Schritt ein *Systemfehler* auf, der zum Systemabsturz führt. Es muss sichergestellt werden, dass beim Systemneustart das Datenobjekt *x* den Wert besitzt, welcher vor der Ausführung von T_3 gültig war, da T_3 durch den Systemfehler keine *COMMIT*-Operation ausführen konnte.

3.3 Weitere Transaktionsarten

Das in Abschnitt 3.2.1 eingeführte *Konzept der Sicherungspunkte* ermöglicht einer flachen Transaktion T_i die Speicherung von Zwischenzuständen, auf welche T_i während ihrer Ausführung jederzeit zurückgesetzt werden kann. Nachfolgend werden alternative Transaktionsarten vorgestellt, die ebenfalls das Speichern von Zwischenzuständen während der Transaktionsausführung erlauben ohne dabei Sicherungspunkte zu verwenden [GR94]. Die folgenden Transaktionsarten basieren auf der Definition 3.1 einer flachen Transaktion. Es wird jedoch davon ausgegangen, dass das Konzept der Sicherungspunkte wegfällt.

3.3.1 Verkettete Transaktionen

Im Kontext der *verketteten Transaktionen* (engl. *chained transactions*) wird zunächst eine neue Operation *CHAIN* ∈ *OP* eingeführt, so dass op_n ∈ {COMMIT, ABORT, CHAIN} gilt. *CHAIN* setzt sich dabei aus den Operationen *COMMIT* und *BOT* zusammen und wird als Einheit (atomar) ausgeführt. Wenn eine Transaktion T_{1a} die *CHAIN*-Operation

ausführt, werden also zunächst alle bis dahin durchgeführten Änderungen durch das *COMMIT* vorläufig gespeichert (falls keine Konsistenzverletzungen vorliegen). Danach wird durch das *BOT* eine neue Transaktion T_{1b} gestartet. T_{1b} kann dann wiederum mittels der *CHAIN*-Operation eine weitere Transaktion T_{1c} starten und so weiter. Dabei ist entscheidend, dass alle entstehenden Transaktionen im Kontext einer großen Transaktion T_1 ablaufen, die in unserem Beispiel mit T_{1a} beginnt und (erfolgreich) beendet wird, sobald eine Transaktion in der Kette nur das *COMMIT* als letzte Operation ausführt. Das *ABORT* einer Transaktion in der Kette führt dazu, dass nur diese eine Transaktion zurückgesetzt wird. Dabei wird der Zustand wiederhergestellt, der nach dem Ausführen der *CHAIN*-Operation der vorangegangenen Transaktion gültig war. Die Ausführung der *CHAIN*-Operation simuliert also das Anlegen eines Sicherungspunkts. Während die flachen Transaktionen beim *Konzept der Sicherungspunkte* auf beliebige Sicherungspunkte zurückgesetzt werden können, kann bei verketteten Transaktionen nur der zuletzt angelegte Sicherungspunkt wiederhergestellt werden [GR94].

Um die gesamte Transaktionskette als Einheit rückgängig machen zu können, bedarf es einer weiteren Operation *ABORTALL* $\in$ *OP*. *ABORTALL* bewirkt, dass alle Änderungen der einzelnen Transaktionen in der Kette von hinten nach vorne rückgängig gemacht werden. Es gilt $op_n \in$ {COMMIT, ABORT, CHAIN, ABORTALL}.

Ein Systemfehler führt zum Abbruch und zur Ungültigkeit der gesamten Transaktionskette.

Die Beispieltransaktion T_1 aus Abschnitt 3.2.1 kann mit dem Konzept der verketteten Transaktionen folgendermaßen umgesetzt werden:

$T_{1a} = BOT_{1a} \rightarrow r_{1a}(\mathrm{x}) \rightarrow w_{1a}(\mathrm{x}) \rightarrow CHAIN_{1a} \cdots T_{1b} = r_{1b}(\mathrm{y}) \rightarrow w_{1b}(\mathrm{y}) \rightarrow ABORT_{1b}$
$\rightarrow r_{1b}(\mathrm{z}) \rightarrow w_{1b}(\mathrm{z}) \rightarrow COMMIT_{1b}$

Die Notation $\cdots$ soll andeuten, dass die Transaktionen T_{1a} und T_{1b} miteinander verkettet sind und eine Einheit bilden.

3.3.2 Geschachtelte Transaktionen

In *geschachtelten Transaktionen* (engl. *nested transactions*) wird eine *Hierarchie* von Transaktionen gebildet. Auf oberster Ebene existiert eine *Supertransaktion* (engl. *toplevel transaction*), die wiederum aus beliebig vielen Subtransaktionen (engl. *subtransactions*) bestehen kann. Diese können ihrerseits wiederum beliebig viele Subtransaktionen enthalten. Dieser Mechanismus kann über beliebig viele Ebenen weitergeführt werden. Letztlich entsteht ein Baum von verschachtelten Transaktionen. Die einzelnen Transaktionen im Baum werden dabei parallel, aber immer im Kontext der Supertransaktion ausgeführt [GR94].

Das Konzept der geschachtelten Transaktionen wird durch die folgenden Regeln bestimmt [GR94]:

Commit-Regel — Ein (lokales) *COMMIT* einer Subtransaktion T_{1a} führt nicht automatisch zu einer dauerhaften Speicherung der von T_{1a} durchgeführten Änderungen. Erst wenn auch alle Vorfahrtransaktionen von T_{1a} bis zur Wurzel (zur Supertransaktion) ebenfalls ihr *COMMIT* erfolgreich ausführen, erfolgt eine dauerhafte Speicherung der durch T_{1a} verursachten Änderungen. Per Induktion wird das *COMMIT* von T_{1a} erst dann wirksam, nachdem die zugehörige Wurzeltransaktion ihre *COMMIT*-Operation erfolgreich ausführt.

Rollback-Regel — Wenn eine (Sub-)Transaktion ein *ABORT* ausführt, müssen alle zugehörigen (direkten und indirekten) Subtransaktionen rekursiv in der Hierarchie abgebrochen und zurückgesetzt werden. Wenn die Supertransaktion eine *ABORT*-Operation ausführt, erfolgt ein *Rollback* aller Subtransaktionen in der Hierarchie.

Sichtbarkeits-Regel — Durch das (lokale) *COMMIT* einer Subtransaktion werden ihre durchgeführten (aber noch nicht dauerhaft gespeicherten) Änderungen ausschließlich für die direkte Vatertransaktion sichtbar.

Während die Supertransaktion alle Eigenschaften des ACID-Paradigmas erfüllt, können Subtransaktionen nur die *Atomaritäts*-, die *Konsistenz*- und die *Isolationseigenschaft* sicherstellen. Da ein (lokales) *COMMIT* einer Subtransaktion durch ein *ABORT* einer Vorfahrtransaktion aufgehoben werden kann, kann die Eigenschaft der *Dauerhaftigkeit* nicht erfüllt werden [GR94].

Um das Konzept der verschachtelten Transaktionen umsetzen zu können, wird eine neue Operation *INTR(m)* (invoke transaction) ∈ *OP* eingeführt, wobei *m* ∈ *IDENT* ist und die ID einer (Sub-)Transaktion darstellt. Es gilt also, dass $op_i \in$ {r(x), w(x), INTR(m)}. Ruft eine Transaktion T_1 die Operation *INTR(1a)* auf, wird eine neue Subtransaktion T_{1a} erzeugt. T_1 wird zur *Vatertransaktion* von T_{1a}, während T_{1a} als *Kindtransaktion* von T_1 bezeichnet wird [GR94].

Abbildung 3.1 zeigt wie die Beispieltransaktion T_1 als geschachtelte Transaktion durchgeführt werden kann. Die Pfeile deuten an, dass durch den Aufruf der *INTR*-Operation durch die Supertransaktion T_1 die jeweilige Subtransaktion auf der rechten Seite erzeugt wird.

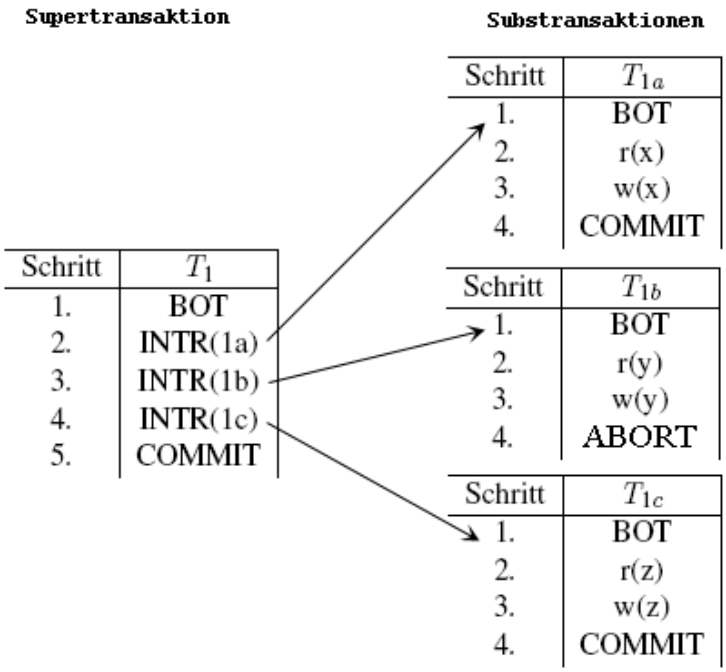


Abbildung 3.1: Beispieltransaktion T_1 als geschachtelte Transaktion

3.4 Der Transaktionsmanager

In einem System, welches ein Transaktionskonzept umsetzt, ist die Verwaltung von parallel laufenden Transaktionen Aufgabe des sogenannten *Transaktionsmanagers*. Dieser muss vor allem sicherstellen, dass jede im System ausgeführte Transaktion die Eigenschaften des *ACID*-Paradigmas erfüllt.

Die Atomarität und die Dauerhaftigkeit wird durch eine Komponente des Transaktionsmanagers gewährleistet, die im Folgenden als *Recovery*-Komponente bezeichnet wird. Die *Synchronisation*-Komponente ist für die Isolation zuständig. Die *Consistency*-Komponente ist für die Gewährleistung der Konsistenz und somit für die Erkennung von Konsistenzverletzungen beim *COMMIT* einer Transaktion verantwortlich. Da Konsistenzbedingungen abhängig vom Anwendungsgebiet (möglichst genau und vollständig) definiert werden müssen, werden diese erst in Abschnitt 4.4 behandelt.

3.5 Fehlerbehandlung durch die Recovery-Komponente

Die *Recovery*-Komponente ist in einem Transaktionskonzept für die korrekte Behandlung aller auftretenden Fehlerfälle verantwortlich. Als Fehlerfall wird in diesem Kontext ein Systemabsturz, das Ausführen der *ABORT*-Operation und das erfolglose *COMMIT* beim Auftreten von Konsistenzverletzungen bezeichnet.

3.5.1 Fehlerklassen

Dabei lassen sich im Kontext der Datenbanken die folgenden Fehlerklassen unterscheiden [KE04].

Lokaler Fehler einer Transaktion — Fehler, die zum Abbruch einer Transaktion T_i führen, werden als *lokale Fehler* bezeichnet. Ein lokaler Fehler tritt auf, falls

- T_i ihre *ABORT*-Operation ausführt,
- nach dem *COMMIT* von T_i eine Konsistenzverletzung festgestellt wird oder
- falls ein systemgesteuerter Abbruch einer Transaktion zur Behebung einer *Verklemmung* (engl. *deadlock*) stattfindet (siehe Abschnitt 3.6.4).

Fehler mit Hauptspeicherverlust — In Datenbanksystemen existiert eine Speicherhierarchie, die sich aus dem (persistenten) *Hintergrundspeicher* und (dem temporären) *Hauptspeicher* (DBMS-Puffer) zusammensetzt. Dabei werden einzelne Datenobjekte sowohl von dem Hintergrundspeicher in den Hauptspeicher eingelagert als auch von dem Hauptspeicher in den Hintergrundspeicher ausgelagert. Wenn ein Fehler mit Hauptspeicherverlust eintritt, kann es zum Beispiel vorkommen, dass Änderungen von erfolgreich abgeschlossenen Transaktionen noch nicht dauerhaft in den Hintergrundspeicher übertragen wurden. Die Aufgabe der Recovery-Komponente ist es in diesem Fall sicherzustellen, dass nach dem Systemneustart die fehlenden Änderungen in den Hintergrundspeicher übertragen werden.

Fehler mit Hintergrundspeicherverlust — Durch einen Fehler im Hintergrundspeicher (zum Beispiel einem *head crash* der Festplatte) werden alle oder Teile der persistenten Daten vernichtet. Aufgabe der Recovery-Komponente ist in diesem Fall die Wiederherstellung des Datenzustands anhand von entsprechenden Archivdateien.

3.5.2 Behebung von lokalen Fehlern

Zur Behebung von lokalen Fehlern wird im Kontext der Datenbanken der gleiche Mechanismus angewendet, welcher auch zur Behebung von Fehlern mit Hauptspeicher- und Hintergrundspeicherverlust zum Einsatz kommt. Die Idee dabei ist, dass alle relevanten durch Transaktionen durchgeführten Operationen in einer *Log*-Datei mitprotokolliert werden. Diese *Log*-Datei wird im Hauptspeicher gehalten und kann bei Bedarf in den Hintergrundspeicher rausgeschrieben werden. In jedem Protokolleintrag für eine Schreiboperation wird festgehalten, wie die verursachte Änderung rückgängig gemacht werden kann (*Undo*) und bei Bedarf reproduziert wird (*Redo*). Das Rollback einer Transaktion wird im Folgenden als *lokales Undo* bezeichnet (siehe Abschnitt 3.5.4). Ein teilweises Zurücksetzen einer Transaktion ist ein *partielles Undo* (siehe Abschnitt 3.5.5).

3.5.3 Protokollierung von Operationen

In diesem Abschnitt wird beispielhaft auf die Protokollierung von flachen Transaktionen mit Sicherungspunkten eingangen. Dabei wird von einer *logischen Protokollierung* ausgegangen, so dass als *Undo*- und *Redo*-Information jeweils (vom Transaktionskonzept unabhängige) Operationen angegeben werden. Als Alternative kann die *physische Protokollierung* verwendet werden, bei der als *Undo*- und *Redo*-Information der Wert des Datenobjekts vor bzw. nach der Operationsausführung protokolliert wird [KE04].

Zusätzlich wird im Folgenden gefordert, dass ein Protokolleintrag vor der zugehörigen Operationsausführung angelegt wird. Vorab sei erwähnt, dass bei Read-Only-Transaktionen die Protokollierung nicht erforderlich ist.

3.5.3.1 Allgemeine Struktur von Log-Einträgen

Für jede von einer Transaktion auf einem Datenobjekt x durchgeführten Schreiboperation werden die folgenden Protokollinformationen benötigt [KE04]:

1. Eine *Redo-Information*, die zur Reproduktion der Schreiboperation dient und

2. eine *Undo-Information*, die angibt, wie die Schreiboperation rückgängig gemacht werden kann.

Desweiteren sind die folgenden zusätzlichen Informationen interessant:

- Eine *LSN* (Log Sequence Number), welche eine eindeutige ID für den Log-Eintrag darstellt. LSNs sollten dabei monoton aufsteigend vergeben werden, damit eine chronologische Reihenfolge von Logeinträgen entsteht.
- Die *TID* (Transaction-ID) der Transaktion, welche die Änderung durchgeführt hat.
- Eine *DOID* (Data Object-ID), welche anzeigt, auf welchem Datenobjekt die Schreiboperation ausgeführt wurde.
- Die *PrevLSN* (Previous Log Sequence Number) erhält als Wert die LSN des vorhergehenden Log-Eintrags der jeweiligen Transaktion. Die PrevLSN wird vor allem zur effizienteren Bearbeitung der Log-Datei beim Undo einer Transaktion angelegt (siehe Abschnitt 3.5.4 und Abschnitt 3.5.5).

Jeder Log-Eintrag einer Schreiboperation hat den folgenden Aufbau:

[LSN, TID, DOID, Redo-Information, Undo-Information, PrevLSN]

Die Operation *BOT* erfordert einen Log-Eintrag mit folgendem Aufbau:

[LSN, TID, 'BOT', 0]

Für die Operation *dsp(m)* gilt der folgende Aufbau:

[LSN, TID, 'dsp', m, PrevLSN]

Leseoperationen und die Operationen *COMMIT*, *ABORT* und *rsp(m)* werden (im Kontext von lokalen Fehlern) nicht mitprotokolliert.

3.5.3.2 Protokollierung einer Beispieltransaktion

Tabelle 3.4 zeigt in ihrer Spalte *Log-Datei* die Protokollierung der in Tabelle 3.1 dargestellten Beispieltransaktion T_1 in leicht abgewandelter Form. Hierbei wird (zunächst) die Operation rsp(1) weggelassen und *ABORT* als letzte Operation angenommen.

Schritt	T_1	Log-Datei
1.	BOT	[#1, T_1, BOT, 0]
2.	r(x)	
3.	w(x)	[#2, T_1, x, x -= 50, x += 50, #1]
4.	dsp(1)	[#3, T_1, dsp, 1, #2]
5.	r(y)	
6.	w(y)	[#4, T_1, y, y -= 20, y += 20, #3]
7.	r(z)	
8.	w(z)	[#5, T_1, z, z -= 30, z += 30, #4]
9.	ABORT	

Tabelle 3.4: Protokollierung der Beispieltransaktion T_1

Es wird der Einfachheit halber angenommen, dass die Datenobjekte *x*, *y* und *z* numerische Werte besitzen und somit (in lokalen Variablen) einfache mathematische Operationen (+ und -) auf den Datenobjekten ausgeführt werden können. Beispielsweise wird im 3. Schritt protokolliert, dass durch die Ausführung der Schreiboperation *w(x)* der Wert des Datenobjekts *x* um den Wert 50 erniedrigt wird (*x -= 50*, Redo-Information). Zudem wird in der zugehörigen *Undo*-Information vermerkt, dass der gültige Wert von *x* vor dem Ausführen der Schreiboperation *w(x)* mit der (inversen) Operation *x += 50* wiederhergestellt werden kann.

3.5.4 Lokales Undo einer Transaktion

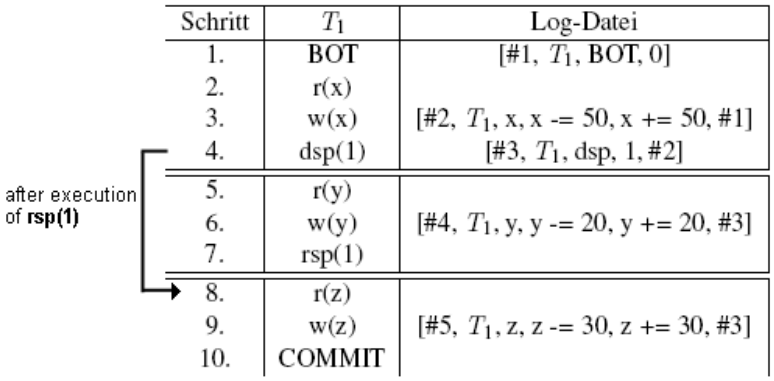
Ein lokales Undo (Rollback) einer Transaktion T_i kann mit Hilfe der zugehörigen Protokolleinträge leicht durchgeführt werden [KE04]. Zunächst muss die LSN des zuletzt angelegten Protokolleintrags für T_i ermittelt werden. Mit Hilfe von *PrevLSN* können die restlichen Protokolleinträge effizient im umgekehrter Reihenfolge durchlaufen werden. Änderungen von Schreiboperationen werden mit Hilfe der entsprechenden protokollierten Undo-Information rückgängig gemacht. Das lokale Undo ist beendet, sobald der erste Eintrag (das BOT) für T_i in der Log-Datei gefunden wurde. Alle für T_i angelegten Protokolleinträge können daraufhin aus der Log-Datei gelöscht werden.

Für das Beispiel in Tabelle 3.4 müssen alle Protokolleinträge für die Transaktion T_1 in umgekehrter Reihenfolge durchlaufen werden (also von #5 bis #1). Bei einer Schreiboperation wird die entsprechende Operation der *Undo*-Information ausgeführt und der daraus resultierende Wert für das entsprechende Datenobjekt neu gesetzt. Ist man beim ersten Protokolleintrag der Transaktion T_1 angekommen, wurden alle durch T_1 durchgeführten Änderungen erfolgreich rückgängig gemacht. Die für T_1 angelegten Protokolleinträge (#1 bis #5) können anschließend aus der Log-Datei entfernt werden.

3.5.5 Partielles Undo einer Transaktion

Wurde während der Ausführung einer Transaktion T_i mindestens ein Sicherungspunkt angelegt, kann ein partielles Undo von T_i durch die Operation *rsp(m)* auf den Sicherungspunkt mit der ID *m* angestossen werden. Der letzte Protokolleintrag von T_i vor dem Ausführen der Operation *rsp(m)* wird als Startpunkt gesetzt. Mit Hilfe von *PrevLSN* können alle zu T_i gehörenden Protokolleinträge in umgekehrter Reihenfolge durchlaufen werden. Dabei müssen alle Änderungen von Schreiboperationen (analog zum lokalen Undo) mittels der Undo-Information zurückgesetzt werden. Trifft man auf einen Protokolleintrag für einen angelegten Sicherungspunkt, wird dieser (und damit auch der Sicherungspunkt selbst) ebenfalls aus der Log-Datei entfernt. Das partielle Undo endet, sobald der Protokolleintrag für den Sicherungspunkt mit der ID *m* erreicht wurde. Alle bis dahin durchlaufenden Protokolleinträge können aus der Log-Datei gelöscht werden.

Je nach Umsetzung des Konzepts der Sicherungspunkte kann auch der Protokolleintrag für den Sicherungspunkt mit der ID *m* entfernt werden, falls der zugehörige Sicherungspunkt ebenfalls mitgelöscht werden soll. Alternativ bleibt der Protokolleintrag in der Log-Datei erhalten, so dass die Transaktion T_i in ihrer weiteren Ausführung wieder auf den zugehörigen Sicherungspunkt zurückgesetzt werden kann.



Schritt	T_1	Log-Datei
1.	BOT	[#1, T_1, BOT, 0]
2.	r(x)	
3.	w(x)	[#2, T_1, x, x -= 50, x += 50, #1]
4.	dsp(1)	[#3, T_1, dsp, 1, #2]
5.	r(y)	
6.	w(y)	[#4, T_1, y, y -= 20, y += 20, #3]
7.	rsp(1)	
8.	r(z)	
9.	w(z)	[#5, T_1, z, z -= 30, z += 30, #3]
10.	COMMIT	

Abbildung 3.2: Beispiel für ein partielles Undo

Abbildung 3.2 zeigt die vollständige Protokollierung der in Tabelle 3.1 (Seite 31) dargestellten Beispieltransaktion T_1. Im 4. Schritt wird ein Sicherungspunkt mit der ID *1* angelegt. Im 7. Schritt wird die Operation *rsp(1)* ausgeführt, wodurch das Zurücksetzen auf den im 4. Schritt angelegten Sicherungspunkt mit der ID 1 signalisiert wird. Die *Recovery*-Komponente muss in diesem Fall alle Protokolleinträge für T_1 vor dem 7. Schritt rückwärts durchlaufen und alle Schreiboperationen mittels der *Undo*-Information zurücksetzen. Trifft die *Recovery*-Komponente auf den Protokolleintrag, welcher den Sicherungspunkt mit der ID 1 repräsentiert (im Beispiel der Protokolleintrag *#3*), ist das partielle Undo von T_1 auf den gewünschten Sicherungspunkt erfolgreich abgeschlossen. Im Beispiel wird dabei der Protokolleintrag *#4* aus der Log-Datei entfernt. Der Log-Eintrag

für den Sicherungspunkt mit der ID 1 (und somit der Sicherungspunkt selbst) bleibt hier erhalten. Dies kann man daran erkennen, dass bei dem Log-Eintrag #5 als Wert für die *PrevLSN* der Wert #3 und nicht #2 eingetragen wurde. Der dargestellte Pfeil deutet an, dass nach der Durchführung des partiellen Undos mit dem 8. Schritt fortgefahren wird.

3.6 Synchronisation von parallel laufenden Transaktionen

Bei mehreren in einem System parallel ablaufenden Transaktionen muss für jede Transaktion T_i die Isolations-Eigenschaft des ACID-Paradigmas sichergestellt werden. Diese lässt sich im einfachsten Fall mit dem *Eintransaktionsbetrieb* erzielen, in dem alle Transaktionen nacheinander (*sequentiell*) ausgeführt werden. Diese Lösung liefert implizit die *vollständige Isolation*, wodurch keine Synchronisation zwischen den einzelnen Transaktionen stattfinden muss. Da jedoch beim *Eintransaktionsbetrieb* die Vorteile der parallelen Ausführung von Transaktionen (höhere Ausführungsgeschwindigkeit und besserer Ressourcenverbrauch) unberücksichtigt bleiben, wird in der Regel ein *Mehrtransaktionsbetrieb* angestrebt [KE04]. Vorab muss erwähnt werden, dass eine Synchronisation nicht notwendig ist, wenn zu einem Zeitpunkt nur eine Transaktion im System aktiv ist oder alle aktiven Transaktionen ausschließlich Read-Only-Transaktionen sind.

3.6.1 Mögliche Fehler beim unkontrollierten Mehrtransaktionsbetrieb

Im Folgenden werden mögliche Fehler erläutert, die beim *unkontrollierten* Mehrbenutzerbetrieb auftreten können.

3.6.1.1 Lost Update

Tabelle 3.5 zeigt zwei Transaktionen T_1 und T_2, die Operationen auf einem numerischen Datenobjekt *x* ausführen. Die mit - markierten Schritte signalisieren, dass die Transaktionen vom Transaktionskonzept unabhängige Operationen auf *x* ausführen, deren Ergebnisse in den lokalen Variablen x_1 bzw. x_2 gespeichert werden. Die Werte der lokalen Variablen werden dann durch die Schreiboperationen im 5. bzw. 6. Schritt im Datenobjekt *x* gespeichert. Das Datenobjekt *x* trage zu Beginn den Wert 10, der im 3. und 4. Schritt jeweils für T_1 bzw. T_2 eingelesen wird. Vor der Ausführung des 5. Schritts erwartet man, dass x = 13 gilt. Da T_1 und T_2 unabhängig voneinander auf dem Ursprungswert von *x* (10) arbeiten, wird zunächst im 5. Schritt der Wert 11 von T_1 gespeichert, welcher im folgenden Schritt von T_2 (durch den Wert 12) überschrieben wird. Die Änderung von T_1 geht demnach verloren. Dieses Phänomen wird als *Lost Update* bezeichnet.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	r(x)	
4.		r(x)
-	x_1 = x + 1	
-		x_2 = x + 2
5.	w(x)	
6.		w(x)
7.	COMMIT	
8.		COMMIT

Tabelle 3.5: Lost Update nach [Vos00]

3.6.1.2 Dirty Read

Wenn eine Transaktion den Wert eines Datenobjekts liest, welcher so niemals in der konsistenten Datenbasis vorkommen wird, spricht man von der *Abhängigkeit von einer nicht freigegebenen Änderung* [KE04]. Diese Situation wird auch als *Dirty Read* bezeichnet. Tabelle 3.6 zeigt ein entsprechendes Beispiel zur Verdeutlichung des Problems.

Schritt	T_1	T_2
1.	BOT	
2.	r(x)	
3.	w(x)	
4.		BOT
5.		r(x)
6.		w(x)
7.		COMMIT
8.	ABORT	

Tabelle 3.6: Dirty Read nach [KE04]

In der dargestellten Ausführungsreihenfolge liest T_2 im 5. Schritt den aktuellen Wert für das Datenobjekt *x* ein, führt lokal eine Berechnung auf diesem aus und schreibt den neuen Wert für *x* im 6. Schritt in den Speicher. Dieser neue Wert ist jedoch ungültig. Die zunächst das Datenobjekt *x* im 3. Schritt verändernde Transaktion T_1 wird im 8. Schritt mittels *ABORT* zum Rollback gezwungen. Somit muss der Wert von *x* vor der Ausführung von T_1 wiederhergestellt werden. Die Transaktion T_2 wurde also auf der Basis von inkonsistenten Daten (engl. *dirty data*) durchgeführt.

3.6.1.3 Inconsistent Read

Tabelle 3.7 zeigt das Problem des *Inconsistent Read* (auch als *Unrepeatable Read* bezeichnet). Die Transaktion T_1 habe die Aufgabe die Summe der numerischen Werte für die Datenobjekte *x* und *y* zu ermitteln. Vor dem Beginn von T_1 seien x = 20 und y = 30. Nach dem 9. Schritt ermittelt T_1 für *x* + *y* = 40, obwohl durch die Änderungen von T_2 eigentlich *x* (30) + *y* (20) = 50 gelten müsste. Die von T_1 durchgeführte Leseoperation im 2. Schritt ist ein *Inconsistent Read*, da der hier eingelesene Wert für *x* im späteren Verlauf der Transaktion veraltet und ungültig ist.

Schritt	T_1	T_2
1.	BOT	
2.	r(x)	
3.		BOT
4.		r(y)
-		y_1 = y - 10
5.		w(y)
6.		r(x)
-		x_1 = x + 10
7.		w(x)
8.		COMMIT
9.	r(y)	
10.	COMMIT	

Tabelle 3.7: Inconsistent Read nach [Vos00]

3.6.1.4 Das Phantomproblem

Das sogenannte *Phantomproblem* wird mit dem Szenario aus Tabelle 3.8 erläutert. Eine Transaktion T_1 ermittelt zunächst alle Knoten (die Menge *V*) in einem Graphen *G*. Anschließend führt T_1 einen Transformationsalgorithmus auf dieser ermittelten Menge aus. Bevor dieser gestartet wird, fügt eine parallel laufende Transaktion T_2 einen neuen Knoten *v1* in die Menge *V* ein. *v1* wird jedoch nicht von T_1 beim Transformationsalgorithmus mitberücksichtigt. Aus der Sicht von T_1 stellt der Knoten *v1* ein sogenanntes Phantom dar.

T_1	T_2
Menge *V* im Graphen *G* ermitteln	Neuen Knoten *v1* in *V* einfügen
Transformation auf der gesamten Menge *V*	

Tabelle 3.8: Phantomproblem

3.6.2 Theorie der Serialisierbarkeit

Die im vorigen Abschnitt erläuterten Probleme beim unkontrollierten Mehrtransaktionsbetrieb lassen sich am simpelsten durch den bereits angesprochenen Eintransaktionsbetrieb lösen. Diese Lösung ist jedoch vor allem aus Performanzgründen ungeeignet. Aufgrund dessen wird das *Konzept der Serialisierbarkeit* [KE04] eingeführt, welches den Vorteil der seriellen (sequentiellen) Transaktionsausführung (die vollständige Isolationsgarantie) mit der erhöhten Ausführungsgeschwindigkeit des Mehrtransaktionsbetriebs kombiniert [KE04].

3.6.2.1 Historie

Ein Ziel der Serialisierbarkeit ist es, die Reihenfolge von Operationen verschiedener parallel laufender Transaktionen so festzulegen, dass ein möglichst hoher Grad an Parallelität gewährleistet wird. Die entstehende Operationsreihenfolge soll dabei in ihrer Wirkung einer sequentiellen Ausführung der Transaktionen entsprechen, um die in Abschnitt 3.6.1 angesprochenen Fehlersituationen zu vermeiden. Die Ausführungsreihenfolge von Operationen mehrerer parallel ablaufender Transaktionen wird in einer *Historie* (engl. *schedule*) notiert [KE04]. Im Folgenden werden nur Historien von flachen Transaktionen betrachtet, wobei das Konzept der Sicherungspunkte in diesem Kontext keine Rolle spielt. Eine Historie *H* für das Beispiel aus Tabelle 3.7 wird wiefolgt notiert:

$H = BOT_1 \rightarrow r_1(x) \rightarrow BOT_2 \rightarrow r_2(y) \rightarrow w_2(y) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow COMMIT_2 \rightarrow r_1(y) \rightarrow COMMIT_1$

3.6.2.2 Bestimmung der Ausführungsreihenfolge bei Konfliktsituationen

Die Ausführungsreihenfolge muss vor allem für sogenannte *Konfliktoperationen* festgelegt werden. Als Konfliktoperationen werden dabei diejenigen Operationen bezeichnet, welche bei unkontrollierter paralleler Ausführung zu Inkonsistenzen führen können. Dabei ist es irrelevant, ob letztlich eine Inkonsistenz zustande kommt oder nicht. Lediglich die potentielle Gefahr der Inkonsistenz ist entscheidend.

Um die möglichen Konfliktsituationen zu verdeutlichen wird im Folgenden von zwei Transaktionen T_i und T_j ausgegangen, die beide auf das gleiche Datenobjekt *x* zugreifen wollen. Die folgenden Kombinationsmöglichkeiten müssen unterschieden werden:

1. r_i**(x) und** r_j**(x)**: solange beide Operationen gleichzeitig nur lesend auf das gleiche Datenobjekt *x* zugreifen möchten, ist deren Reihenfolge in der Historie zueinander irrelevant. Diese Kombination stellt demnach kein Konflikt dar.

2. r_i**(x) und** w_j**(x)**: es handelt sich um eine Konfliktsituation, da T_i entweder den alten oder den neuen Wert von *x* liest. Es muss also entweder r_i(x) vor w_j(x) oder w_j(x) vor r_i(x) als Reihenfolge in der Historie festgelegt werden.

3. w_i**(x) und** r_j**(x)**: analog zu Punkt 2.

4. w_i**(x) und** w_j**(x)**: auch bei dieser Kombination ist die Reihenfolge der Ausführung für die Konsistenzerhaltung entscheidend. Hier muss also festgelegt werden, ob w_i(x) vor w_j(x) oder w_j(x) vor w_i(x) gelten soll.

3.6.2.3 Die Forderung nach serialisierbaren Historien

Bei der Festlegung von Historien ist eine notwendige Forderung, dass sogenannte *serialisierbare* Historien entstehen. Eine Historie *H* wird als *serialiserbar* bezeichnet, wenn sie äquivalent zu einer seriellen Historie H_s ist. In einer seriellen Historie H_s für eine beliebige Historie *H* wird die Reihenfolge der einzelnen Operationen so festgelegt, dass die in *H* beteiligten Transaktionen sequentiell ausgeführt werden [KE04].

Um feststellen zu können, ob eine gegebene Historie *H* serialisierbar ist, kann ein sogenannter *Serialisierbarkeitsgraphen SG(H)* erzeugt werden. *SG(H)* enthält als Knotenmenge alle zur Zeit parallel ablaufenden Transaktionen (T_1, ..., T_n). Für je zwei in Konflikt stehende Operationen op_i, op_j aus der Historie *H* mit $op_i <_H op_j$ (op_i wird vor op_j in *H* ausgeführt) wird die Kante $T_i \rightarrow T_j$ (falls noch nicht vorhanden) in den Graphen *SG(H)* eingefügt. Nach dem *Serialisierbarkeitstheorem* ist eine Historie *H* genau dann serialisierbar, wenn der zugehörige Serialisierbarkeitsgraph *SG(H) azyklisch* ist (keine Zyklen enthält) [KE04].

Abbildung 3.3 zeigt ein Beispiel für eine nicht-serialisierbare Historie H_1, dessen Serialisierbarkeitsgraph *SG(*H_1*)* zyklisch ist. Die Kante von T_1 nach T_2 entsteht durch die Konfliktoperationen w_1(x) und r_2(x), die Kante von T_2 nach T_1 durch die Konfliktoperationen w_2(y) und r_1(y).

$$H_1 = \mathrm{BOT}_1 \rightarrow \mathrm{r}_1(\mathrm{x}) \rightarrow \mathrm{w}_1(\mathrm{x}) \rightarrow \mathrm{BOT}_2 \rightarrow \mathrm{r}_2(\mathrm{x}) \rightarrow \mathrm{w}_2(\mathrm{x}) \rightarrow \mathrm{r}_2(\mathrm{y}) \rightarrow \mathrm{w}_2(\mathrm{y}) \rightarrow \mathrm{COMMIT}_2 \rightarrow \mathrm{r}_1(\mathrm{y}) \rightarrow \mathrm{w}_1(\mathrm{y}) \rightarrow \mathrm{COMMIT}_1$$

$$SG(\mathrm{H}_1) = \mathrm{T}_1 \rightleftarrows \mathrm{T}_2$$

Abbildung 3.3: Beispiel einer nicht-serialisierbaren Historie [KE04]

3.6.2.4 Eigenschaften von Historien bezüglich der Fehlerbehandlung

Die angesprochene Serialisierbarkeit stellt die minimale Anforderung an Historien dar. Im Kontext der Fehlerbehandlung (Recovery) (siehe Abschnitt 3.5, Seite 37) werden zusätzliche Forderungen an die Eigenschaften von gültigen Historien gestellt. Diese Anforderungen werden nachfolgend erläutert [KE04].

Rücksetzbare Historien — Rücksetzbare Historien stellen die Minimalanforderung im Bezug auf die Fehlerbehandlung dar. Sie verlangt, dass Historien so aufgebaut werden müssen, dass eine aktive Transaktion jederzeit abgebrochen werden kann, ohne dass die Wirkungen von bereits abgeschlossenen Transaktionen zu Konsistenzverletzungen führen. Diese Forderung verhindert also das Problem des *Dirty Read* [KE04].

Um dies genauer beleuchten zu können, wird zunächst das Prinzip der *schreibenden* und der *lesenden* Transaktion erläutert [KE04]. In einer *Historie H* liest die Transaktion T_i von der Transaktion T_j falls gilt, dass

1. w_j(x) $<_H$ r_i(x) (T_j schreibt also zunächst ein Datenobjekt x, welches T_i nachfolgend liest),

2. $ABORT_j$ $\neg <_H$ r_i(x) (T_j wird nicht vor dem Lesevorgang von T_i zurückgesetzt) und

3. für jede weitere Transaktion T_k, durch die w_j(x) $<_H$ w_k(x) $<_H$ r_i(x) gilt, muss $ABORT_k$ $<_H$ r_i(x) gegeben sein.

Eine Historie wird als rücksetzbar bezeichnet, falls immer die schreibende Transaktion (T_j) vor der lesenden Transaktion (T_i) ihre *COMMIT*-Operation durchführt ($COMMIT_j$ $<_H$ $COMMIT_i$) [KE04].

Historien ohne kaskadierendes Rücksetzen — Eine weitere nicht notwendige - aber aus Gründen der Performanz nützliche - Forderung ist das Verhindern von sogenanntem *kaskardierendem Rücksetzen* von Transaktionen. In Tabelle 3.9 müssen (aufgrund der entstandenen Abhängigkeiten) nach dem *ABORT* von T_1 auch T_2 und T_3 zurückgesetzt werden, um Dirty Reads der Datenobjekte *x* und *y* zu verhindern. Eine „Lawine“ von Rollbacks ist die Folge [KE04].

Zur Vermeidung solcher Situationen wird verlangt, dass Änderungen an Datenobjekten erst nach einem *COMMIT* freigegeben werden. Dies bedeutet, dass $COMMIT_j$ $<_H$ r_i(A) gelten muss, falls T_i ein Datenobjekt *x* von T_j liest [KE04].

Schritt	T_1	T_2	T_3
1.	BOT		
2.	w(x)		
3.		BOT	
4.		r(x)	
5.		w(y)	
6.			BOT
7.			r(y)
8.			w(z)
9.	ABORT		
10.		...	...

Tabelle 3.9: Kaskadierendes Rücksetzen nach [KE04]

Strikte Historien — Strikte Historien stellen eine verschärfte Version der Historien ohne kaskadierendes Rücksetzen dar. Wann immer in der Historie $w_j(x) <_H op_i(x)$ mit $op_i = r_i$ oder $op_i = w_i$ auftritt, dann muss entweder

- $COMMIT_j <_H op_i(x)$ oder
- $ABORT_j <_H op_i(x)$ gelten.

Mit anderen Worten, darf ein von einer noch aktiven Transaktion verändertes Datenobjekt *x* von anderen Transaktionen weder gelesen noch überschrieben werden.

3.6.3 Dynamische Synchronisation durch den Scheduler

Die Erstellung von dynamischen Historien zur Laufzeit ist die Aufgabe der sogenannten *Scheduler*-Komponente des Transaktionsmanagers. Dieser empfängt als Eingabe Operationen von parallel ablaufenden Transaktionen in beliebiger Reihenfolge (*Input-Schedule*). Als Ausgabe muss der Scheduler Historien liefern, die als Mindestanforderung die Serialisierbarkeit erfüllen (*Output-Schedule*) [KE04]. In der Praxis werden aus Performanzgründen Historien ohne kaskadierendes Rücksetzen verlangt [Vos00]. Zur Realisierung eines solchen *Schedulers* können verschiedene Synchronisationsmethoden verwendet werden, welche die geforderten Eigenschaften (zum Teil) garantieren.

3.6.3.1 Klassifikation von Synchronisationsmethoden

Synchronisationsmethoden können in *pessimistische* und *optimistische* Verfahren unterteilt werden. Die *pessimistischen* (und präventiven) Verfahren basieren auf der Annah-

me, dass potentielle Konflikte zu einer nicht serialisierbaren Historie führen (was in der Praxis nicht immer der Fall ist). Das Auftreten von Synchronisationsproblemen wird also a priori verhindert. Pessimistische Verfahren können ihrerseits in *sperrbasierte* und *Zeitstempel-basierte* Synchronisationmethoden unterteilt werden. Bei den *optimistischen* Synchronisationsmethoden wird erst nach dem *COMMIT* einer Transaktion festgestellt, ob Konflikte aufgetreten sind. Anschließend wird basierend auf dem Ergebnis der Konfliktüberprü-fung entsprechend gehandelt [KE04].

Eine weitere Synchronisationsmethode ist die *Mehrversionensynchronisation*, bei der verschiedene Versionen von Datenobjekten verwaltet werden. Diese Methode wird in der Regel in Kombination mit einer anderen Synchronisationsmethode eingesetzt und ist davon abhängig entweder als *pessimistisch* oder *optimistisch* einzustufen.

3.6.4 Sperrbasierte Synchronisation

Bei der *sperrbasierten Synchronisation* darf eine Transaktion generell erst nach Erhalt einer entsprechenden Sperre auf ein Datenobjekt zugreifen.

3.6.4.1 Sperrmodi

Je nachdem, ob sich es um eine Lese- oder Schreiboperation handelt, werden zwei Sperrmodi unterschieden [KE04]:

- S (shared, read lock, Lesesperre): Wenn eine Transaktion T_i eine S-Sperre für das Datenobjekt *x* besitzt, darf sie die Operation r(x) ausführen. Das Anfordern einer Lesesperre für ein Datenobjekt *x* wird mit der Operation *lockS(x)*, das Aufheben der Sperre mit *unlockS(x)* durchgeführt.

- X (exclusive, write lock, Schreibsperre): Nur eine Transaktion T_i, welche eine X-Sperre auf einem Datenobjekt *x* besitzt, darf w(x) ausführen. Das Anfordern einer Schreibsperre für ein Datenobjekt *x* wird mit der Operation *lockX(x)*, das Aufheben der Sperre mit *unlockX(x)* durchgeführt.

Die sogenannte *Verträglichkeitsmatrix* in Tabelle 3.10 legt fest, welche Sperren unter welcher Voraussetzung für ein Datenobjekt *x* vergeben werden können. In der Waagerechten wird die existierende Sperre angegeben. *NL* steht dabei für *no lock* (keine Sperre). In der Senkrechten wird die Sperranforderung abgebildet. + steht für *Sperranforderung erlauben*, - für *Sperranforderung ablehnen*.

	NL	S	X
S	+	+	-
X	+	-	-

Tabelle 3.10: Verträglichkeitsmatrix von Sperranforderungen [KE04]

Wenn also (noch) keine Sperre auf einem Datenobjekt *x* existiert, kann sowohl die Lese- als auch die Schreibsperre gewährt werden. Falls eine Lesesperre für ein Datenobjekt *x* vorliegt, kann auch ein weitere Lesesperre erteilt werden. Alle anderen Kombinationen verhindern die Erteilung einer Sperre.

Beim lokalen Undo einer Transaktion T_i (siehe Abschnitt 3.5.4, Seite 40) werden alle von T_i angeforderten Sperren freigegeben. Beim partiellem Undo von T_i (siehe Abschnitt 3.5.5, Seite 41) dürfen nur diejenigen Sperren freigegeben werden, auf deren zugehörigen Datenobjekten vor dem Anlegen des jeweiligen Sicherungspunkts weder Lese- noch Schreiboperationen durch T_i ausgeführt wurden [KE04].

3.6.4.2 Zwei-Phasen-Sperrprotokoll

Setzt ein Scheduler das sogenannte Zwei-Phasen-Sperrprotokoll (engl. *two-phase locking*, 2PL) ein, ist die Serialisierbarkeit einer Historie garantiert. Hierbei werden von jeder Transaktion die folgenden Punkte verlangt [KE04]:

1. Jedes Datenobjekt muss vor dem (lesenden oder schreibenden) Zugriff gesperrt werden.

2. Eine Transaktion fordert Sperren, die sie schon besitzt, nicht neu an.

3. Das Anfordern von Sperren verläuft gemäß der Verträglichkeitsmatrix. Falls einer Transaktion (zunächst) eine Sperre verwährt bleibt, wird die Transaktion solange in eine Warteschlange eingereiht, bis die Sperre letztlich gewährt werden kann.

4. Jede Transaktion durchläuft zwei Phasen:

   - Eine *Wachstumsphase*, in der Sperren angefordert, aber nicht freigegeben werden dürfen und

   - eine *Schrumpfungsphase*, in der alle bis dahin erworbenen Sperren wieder freigegeben werden und keine neuen Sperren erworben werden können.

5. Bei Transaktionsende muss eine Transaktion all ihre Sperren zurückgeben.

3.6.4.3 Strenges Zwei-Phasen-Sperrprotokoll

Das bisher eingeführte 2PL-Protokoll garantiert zwar die Serialisierbarkeit, verhindert jedoch nicht das kaskadierende Zurücksetzen von Transaktionen und auch nicht-rücksetzbare Historien sind möglich. Die Lösung besteht darin, das sogenannte *strenge 2PL-Protokoll* zu verwenden, welches folgende Forderungen an eine Transaktion stellt [KE04]:

- Die Anforderungen 1. - 5. des bisher eingeführten 2PL-Protokolls bleiben erhalten,
- jedoch existiert keine Schrumpfungsphase mehr, so dass alle Sperren erst zum Ende der Transaktion freigegeben werden.

3.6.4.4 Gefahr durch Verklemmungen

Ein mögliches Problem bei den bisher erläuterten sperrbasierten Synchronisationsmethoden ist das Auftreten von *Verklemmungen* (engl. *deadlocks*). Eine solche Situation wird in Tabelle 3.11 beispielhaft demonstriert.

Schritt	T_1	T_2	Bemerkung
1.	BOT		
2.	lockX(x)		
3.		BOT	
4.		lockS(y)	
5.		r(y)	
6.	r(x)		
7.	w(x)		
8.	lockX(y)		T_1 muss auf T_2 warten
9.		lockS(x)	T_2 muss auf T_1 warten
10.			=> Deadlock

Tabelle 3.11: Beispiel für eine Deadlocksituation nach [KE04]

Nach dem 9. Schritt entsteht eine Verklemmung, da die Transaktion T_1 auf die Freigabe einer Sperre von T_2 wartet und vice versa.

Um das Problem von Verklemmungen beheben zu können, müssen entweder Methoden zur nachträglichen Erkennung von Verklemmungen oder Methoden zur präventiven Vermeidung von Verklemmungen eingesetzt werden [KE04]. Die in diesem Kontext verwendeten Mechanismen werden im Rahmen dieser Diplomarbeit nicht erläutert.

3.6.5 Zeitstempel-basierende Synchronisation

Mit Zeitstempeln kann ein Verfahren zur Synchronisation umgesetzt werden, welches vollständig auf Sperren verzichtet [KE04]. Dabei wird jeder Transaktion zu Ausführungsbeginn ein Zeitstempel *TS* zugeordnet. Hier gilt TS(T_i) < TS(T_j), falls T_i älter als T_j ist. Zudem werden jedem Datenobjekt *x* zwei Marken zugeordnet [KE04]:

- rTS(x), welcher den Zeitstempel der jüngsten Transaktion enthält, die *x* gelesen hat.
- wTS(x), welcher den Zeitstempel der jüngsten Transaktion enthält, die *x* geschrieben hat.

Für eine beliebige Transaktion T_i werden folgende Regeln festgelegt [KW06]:

- T_i will r_i(x) ausführen:

  Falls TS(T_i) < wTS(x) ist, dann muss T_i zurückgesetzt werden. Ist TS(T_i) ≥ wTS(x), kann T_i die Leseoperation durchführen und rTS(x) wird auf max(TS(T_i), rTS(x)) gesetzt.

- T_i will w_i(x) ausführen:

  Falls TS(T_i) < rTS(x) gilt, wird T_i zurückgesetzt. Falls TS(T_i) < wTS(x) gilt, wird die Schreiboperation von T_i ignoriert, die Ausführung von T_i jedoch fortgeführt. Treten die beiden Fälle nicht auf, wird die Schreiboperation für T_i ausgeführt und wTS(x) = TS(T_i) gesetzt.

Zurückgesetzte Transaktionen werden mit dem aktuellen Zeitstempel neugestartet. Diese Synchronisationsmethode liefert serialisierbare Historien, die äquivalent zu der seriellen Ausführung der beteiligten Transaktionen in der Reihenfolge ihrer Startzeitstempel ist. Deadlocks treten nicht auf. Ein Nachteil dieses Verfahrens ist die unter Umständen hohe Quote an zurückgesetzten und neugestarteten Transaktionen, welche die Systemperformanz beeinträchtigen kann [KE04].

3.6.6 Optimistische Synchronisation

Eine Alternative zu den bisher vorgestellten pessimistischen Synchronisationsverfahren ist die sogenannte optimistische Synchronisation. Eine Transaktion T_i wird dabei in die folgenden drei Phasen unterteilt [KE04][Wei88]:

1. **Lesephase**: In dieser Phase werden alle von T_i ausgeführten Lese- und Schreiboperationen auf einem nur für T_i gültigen *lokalen Puffer* ausgeführt. Die in der Lesephase durch T_i durchgeführten Schreiboperationen haben also (zunächst) keinerlei Wirkung auf die eigentlichen Daten.

2. **Validierungsphase**: Nachdem T_i ihre *COMMIT*-Operation ausführt, folgt die Überprüfung, ob durch die in T_i durchgeführten Änderungen Konflikte (Konsistenzverletzungen) auftreten. Werden Konflikte festgestellt, scheitert die Validierung und T_i muss zurückgesetzt werden.

3. **Schreibphase**: Verläuft die Validierung für T_i positiv (konnten also keine Konflikte festgestellt werden), werden alle durch T_i durchgeführten Änderungen in den eigentlichen Daten übernommen.

Bei der Umsetzung der Validierungsphase für T_i gibt es zwei Vorgehensweisen [Wei88]:

- **Backward Oriented Optimistic Concurrency-Control** (*BOCC*), bei der die Transaktion T_i gegenüber bereits abgeschlossenen (sich also nicht mehr in der Lesephase befindenden) Transaktionen T_j validiert wird und

- **Forward Oriented Optimistic Concurrency-Control** (*FOCC*), bei der T_i gegenüber noch laufenden (sich also noch in der Lesephase befindenden) Transaktionen T_j validiert wird.

Für das *BOCC*-Verfahren kann die Validierungsphase mit den folgenden Regeln umgesetzt werden [KE04]:

- Jede Transaktion T_i erhält beim Eintritt in die Validierungsphase einen Zeitstempel *TS*.

- Für die Validierung von T_i sind alle (aus Sicht von T_i) älteren Transaktionen T_j relevant, für die TS(T_j) < TS(T_i) gilt.

- Für jede Transaktion T_j wird bezogen auf T_i überprüft, ob einer der folgenden zwei Bedingungen erfüllt ist:

  1. T_j war zu Beginn der Transaktion T_i (mitsamt ihrer Schreibphase) abgeschlossen.

  2. Die Menge der Datenobjekte, auf denen T_j eine Schreiboperation durchgeführt hat und die Menge der Datenobjekte, auf denen T_i eine Leseoperation ausgeführt hat, sind disjunkt. Es gilt also wSet(T_j) $\cap$ rSet(T_i) = $\varnothing$.

Wenn für alle Transaktionen T_j mindestens eine der beiden Bedingungen erfüllt ist, kann T_i in die Schreibphase übergehen. Scheitert die Validierung, muss T_i zurückgesetzt und neugestartet werden.

Um die Korrektheit dieses Verfahrens zu gewährleisten, muss das System sicherstellen, dass sich zu einem Zeitpunkt nur höchstens eine Transaktion in der Validierungs- oder der Schreibphase befindet.

Für das *FOCC*-Verfahren gelten die folgenden Regeln [Wei88]:

- Jede Transaktion T_i, welche die Validierungsphase betritt, wird gegenüber allen sich zu diesem Zeitpunkt in der Lesephase aufhaltenden Transaktionen T_j validiert.

- Falls für jede sich in der Lesephase aufhaltenden Transaktion T_j die Bedingung wSet(T_i) ∩ rSet(T_j) = ∅ gilt, darf T_i in die Schreibphase übergehen. Ist diese Bedingung für mindestens eine Transaktion T_j nicht erfüllt, muss einer der folgenden Aktionen durchgeführt werden:

  1. T_i wird zurückgesetzt,

  2. die konfliktversursachenden(n) Transaktion(en) T_j wird (werden) zurückgesetzt oder

  3. T_i tritt in eine Warteschlange und durchläuft die Validierung zu einem späteren Zeitpunkt.

  Alle Transaktionen, die aufgrund der Durchführung der Validierungsphase von T_i zurückgesetzt werden, müssen neugestartet werden.

Um die Korrektheit des *FOCC*-Verfahrens zu gewährleisten, muss während der Validierungsphase der Transaktion T_i die Menge wSet(T_i) mit einer Schreibsperre geschützt werden.

Ähnlich der Zeitstempel-basierenden Synchronisation (siehe Abschnitt 3.6.5) kann bei der optimistischen Synchronisation das häufige Zurücksetzen und Neustarten von Transaktionen zu Performanzproblemen führen. Optimistische Synchronisationsverfahren sind aus diesem Grund vor allem bei einer hohen Quote an Read-Only-Transaktionen effizient nutzbar, da hierbei weniger Konflikte auftreten.

Der große Vorteil der optimistischen Synchronisation liegt in der einfachen Ausführung des lokalen Undos (siehe Abschnitt 3.5.4, Seite 40) einer Transaktion T_i. Da sich Schreiboperationen während der Lesephase von T_i lediglich auf den lokalen Puffer von T_i auswirken, entfällt das Zurücksetzen von Änderungen in den eigentlichen Daten.

3.6.7 Mehrversionen-Synchronisation

Bei der *Mehrversionen-Synchronisation* (engl. *multiversion concurrency control*) werden verschiedene Versionen von Datenobjekten im System verwaltet. Dabei erzeugt jede an einem Datenobjekt *x* durchgeführte Änderung eine neue Version x_i des betroffenen Datenobjekts, sofern sie durch ein erfolgreiches *COMMIT* einer Transaktion T_i entsteht [HR99].

Für diese Synchronisationsmethode werden unterschiedliche Vorgehensweisen für *Read-Only-* und *Read-Write-Transaktionen* definiert. Eine Read-Only-Transaktion T_i kann während ihrer Ausführung für jedes Datenobjekt *x* nur auf diejenige Version x_i zugreifen, welche zu ihrem *BOT*-Zeitpunkt gültig war. Für jedes Datenobjekt *x* gilt zudem, dass neue Versionen x_i, die während der Ausführung von T_i durch Änderungen parallel laufender Transaktionen T_j entstehen, nicht für T_i zugänglich sind. Da Read-Only-Transaktionen während ihrer gesamten Lebensdauer die gleiche (gültige und konsistente) Sicht auf die Datenobjekte behalten, ist für solche Transaktionen vor allem im Zusammenspiel mit anderen parallel laufenden Transaktionen keinerlei Synchronisation notwendig. Überwiegen also Read-Only-Transaktionen, wird durch die festgelegte Vorgehensweise der Synchronisationsaufwand erheblich reduziert. Dabei muss man in Kauf nehmen, dass Read-Only-Transaktionen nicht immer die aktuellsten Daten zu sehen bekommen [HR99].

Um die für eine *Read-Only-Transaktion* T_i gültigen Versionen für jedes Datenobjekt *x* ermitteln zu können, muss jede neue Version x_i mit einem Zeitstempel *cTS(x_i)* markiert werden. *cTS(x_i)* erhält als Wert den *COMMIT*-Zeitstempel der Transaktion T_j, welche durch ihr erfolgreiches *COMMIT* die neue Version x_i erzeugt hat. Jede gestartete *Read-Only-Transaktion* T_i wird bei Ausführung ihrer *BOT*-Operation der Zeitstempel *TS(T_i)* zugeordnet. Möchte T_i nun auf das Datenobjekt *x* zugreifen, muss die Version x_i ermittelt werden, für die cTS(x_i) ≤ TS(T_i) gültig ist, wobei es keine weitere Version x_j geben darf, für die cTS(x_i) < cTS(x_j) ≤ TS(T_i) gilt [HR99].

Eine *Read-Write-Transaktion* T_i greift während ihrer Ausführung für jedes Datenobjekt *x* immer auf die aktuell gültige Version x_i zu. Aus diesem Grund muss der Zugriff auf die Datenobjekte zwischen Read-Write-Transaktionen synchronisiert werden. Zur Synchronisation können dabei die bisher vorgestellten (pessimistischen und optimistischen) Synchronisationsverfahren eingesetzt werden [HR99].

Die unterschiedlichen Versionen der Datenobjekte werden in einem sogenannten *Versionen-Pool* verwaltet. Die Verwaltung verschiedener Versionen für die einzelnen Datenobjekte erhöht einerseits den Speicherplatzbedarf und kann andererseits zu Speicherplatzproblemen führen, falls nicht genügend Speicher zum Anlegen einer neuen Version vorhanden ist. Um diese Situation zu vermeiden, muss das System dafür sorgen, dass nicht mehr benötigte Versionen von Datenobjekten aus dem Versionen-Pool entfernt werden. Diese Aufräumarbeit (engl. *garbage collection*) sollte in regelmäßigen Abständen durchgeführt werden [HR99].

4 Entwurf eines Transaktionskonzepts für JGraLab

Nachdem in Kapitel 3 die grundlegenden Konzepte zu *ACID*-Transaktionen aus der Sicht der Datenbankforschung erläutert wurden, wird es in diesem Kapitel darum gehen, diese Erkenntnisse auf das Graphenlabor (und im Speziellen auf die aktuelle Java-Version JGraLab *Carnotaurus*) zu übertragen.

In einem ersten Schritt werden die Eigenschaften von Transaktionen im Graphenlabor (im Vergleich zu den Transaktionen in Datenbanken) erläutert (siehe Abschnitt 4.1). Im Anschluss folgt eine Analyse der im Graphenlabor bei der parallelen Ausführung von Transaktionen auftretenden Konfliktsituationen (siehe Abschnitt 4.2.1). Damit verbunden werden die in Abschnitt 3.6 (Seite 42) vorgestellten Synchronisationsverfahren im Hinblick auf ihre Tauglichkeit und Umsetzbarkeit im JGraLab bewertet (siehe Abschnitt 4.2.2). Anhand dieser Ergebnisse wird ein Konzept für ein Synchronisationsverfahren im JGraLab entwickelt (siehe Abschnitt 4.2.3). Anschließend wird auf die Umsetzung der Recovery-Komponente eingegangen (siehe Abschnitt 4.3). In einem letzten Schritt wird erläutert, inwiefern die Überprüfung von Konsistenzbedingungen im Kontext des Graphenlabors umgesetzt wird (siehe Abschnitt 4.4).

4.1 Transaktionen im Kontext des Graphenlabors

In diesem Abschnitt wird zunächst die umzusetzende *Transaktionsart* im JGraLab festgelegt (siehe Abschnitt 4.1.1). Anschließend folgt eine Analyse, inwiefern die Definition 3.1 für flache Transaktionen (Seite 28) im Kontext des Graphenlabors anwendbar und gültig ist. Daraus resultierend wird zum einen definiert, welche *Funktionalität* und *Eigenschaften* Transaktionen im Graphenlabor besitzen (siehe Abschnitt 4.1.2). Zum anderen werden die relevanten *Datenobjekte* und die darauf ausführbaren *Lese-* und *Schreiboperationen* identifiziert (siehe Abschnitt 4.1.3).

4.1.1 Umzusetzende Transaktionsart

Als umzusetzende Transaktionsart werden die *flachen Transaktionen mit dem Konzept der Sicherungspunkte* ausgewählt (siehe Abschnitt 3.2.1, Seite 28). Diese verfolgen ein relativ simples Konzept im Vergleich zu den komplexeren in Abschnitt 3.3 (Seite 33) eingeführten *verketteten* und *geschachtelten Transaktionen*. Zudem werden in jenen Transaktionsarten Verfahren eingeführt, die das in flachen Transaktionen eingesetzte Konzept der Sicherungspunkte zur Speicherung von Zwischenzuständen letztlich nur simulieren oder nachzuahmen versuchen. Neue Funktionalität wird dem Transaktionskonzept dadurch nicht geliefert.

4.1.2 Funktionalität und Eigenschaften von Transaktionen

Im Folgenden wird erläutert, welche Funktionalität und Eigenschaften Transaktionen im Graphenlabor besitzen. An geeigneten Stellen werden die Unterschiede zu Transaktionen in Datenbanksystemen aufgezeigt.

4.1.2.1 Herleitung aus der Definition

Eine (flache) Transaktion T_i im JGraLab muss gemäß Definition 3.1 (Seite 28) mindestens die folgenden Operationen für dessen Steuerung umsetzen:

- die *BOT*-Operation,
- die *COMMIT*-Operation und
- die *ABORT*-Operation.

Für das Konzept der Sicherungspunkte muss T_i die Operationen

- dsp(m) und
- rsp(m)

anbieten.

Führt T_i ihre *BOT*-Operation aus, muss T_i eine im System eindeutige *ID* zugewiesen werden.

Eine Transaktion T_i sollte bei ihrem Start explizit als *Read-Only-Transaktion* deklariert werden können. Dabei muss mit entsprechendem *Exception-Handling* sichergestellt werden, dass T_i während ihrer Ausführung keine Schreiboperationen durchführen kann.

4.1.2.2 Zustände einer Transaktion

Für eine Transaktion T_i werden im Kontext des Graphenlabors die folgenden Zustände definiert:

- Not Running,
- Running,
- Committed und
- Aborted.

Eine Transaktion T_i ist zum Zeitpunkt ihrer Instanziierung im Zustand *Not Running*. T_i wechselt erst dann in den Zustand *Running*, wenn ihre *BOT*-Operation ausgeführt wird. In Anlehnung an SQL-Datenbanksysteme kann ein *implizites BOT* bei der Instanziierung durchgeführt werden, so dass sich T_i schon zu Beginn im Zustand *Running* befindet. Beim erfolgreichen *COMMIT* gelangt T_i in den Zustand *Committed*. Die *ABORT*-Operation versetzt T_i in den Zustand *Aborted*.

4.1.2.3 Neustart einer Transaktion

Das in Kapitel 3 angesprochene *Neustarten* einer Transaktion (vollständiges Zurücksetzen und identisches Wiederausführen einer Transaktion, um beispielsweise eine Konfliktsituation zu beheben) ist im Kontext des Transaktionskonzepts für das Graphenlabor nicht sinnvoll umsetzbar. In einem SQL-Datenbanksystem sind dem Transaktionskonzept die auszuführenden Programme bekannt. Diese setzen sich aus *SQL-Statements* zusammen, welche im Kontext von Transaktionen ablaufen und in atomare *Lese-* und *Schreiboperationen* umgewandelt werden. So können die in einem Programm durchzuführenden Bearbeitungsschritte anhand der zugehörigen SQL-Statements beliebig oft identisch reproduziert werden. Ein erneutes Ausführen der Programme und somit ein Neustart der darin involvierten Transaktionen ist in einem solchen Szenario unproblematisch [GR94].

Diese Eigenschaft kann jedoch nicht auf das Transaktionskonzept innerhalb des Graphenlabors übertragen werden. Im Kontext des Graphenlabors stellt zum Beispiel das Ausführen einer Graphentransformation ein Programm dar, welches mit Hilfe einer oder

mehrerer Transaktionen umgesetzt werden kann. Dieses Programm greift für seine Umsetzung auf die Funktionalität des Graphenlabors und im Speziellen auf das darin enthaltene Transaktionskonzept zu. Das Transaktionskonzept selbst registriert dabei lediglich die durch das Programm gestarteten Transaktionen. Somit wird innerhalb des Transaktionskonzepts von dem Programm als Einheit abstrahiert, welches auf einer höheren Ebene durch den Anwender des Graphenlabors erstellt wird. Scheitert eine durch ein Programm gestartete Transaktion, wird dem Anwender damit signalisiert, dass ein Teil des Programms gescheitert ist. Der Neustart des Programms als Einheit muss dann (falls erwünscht) durch den Anwender des Graphenlabors ausgelöst werden. Die Erkenntnis, dass innerhalb des Graphenlabors kein Neustart von (einzelnen) Transaktionen möglich ist, beeinflusst die Wahl des Synchronisationsverfahrens (siehe Abschnitt 4.2.2).

4.1.3 Datenobjekte und Transaktionsoperationen

Um ein Transaktionskonzept für das Graphenlabor definieren zu können, müssen die relevanten *Datenobjekte* und die darauf ausführbaren *Lese-* und *Schreiboperationen* im JGraLab idenfiziert werden.

4.1.3.1 Analyse des Read-Write-Modells im Kontext des Graphenlabors

Das im Kontext des *Read-Write-Modells* in Definition 3.1 (Seite 28) definierte Universum D von *unteilbaren* und *disjunkten Datenobjekten* ist für das Graphenlabor nicht geeignet. Wird beispielsweise ein Knoten $v \in V$ eines Graphen G gelöscht, werden implizit auch alle zu v inzidenten Kanten $e \in E$ aus dem Graphen G entfernt. In diesem Fall ist der Knoten v mitsamt seiner Inzidenzen als ganzes Datenobjekt zu verstehen. Dieses kann jedoch nicht als *unteilbares* Datenobjekt betrachtet werden, da die zu v inzidenten Kanten wiederum selbst eigene Datenobjekte darstellen. Da zudem zwei voneinander verschiedene und miteinander verbundene Knoten v und w gleiche Kanten als Inzidenzen besitzen, können in diesem Kontext die betroffenen Knoten und Kanten *nicht* als disjunkte Datenobjekte definiert werden. An einem Datenobjekt durchgeführte Änderungen implizieren also im Graphenlabor in vielen Fällen auch Änderungen an mehreren anderen Datenobjekten.

Das *Read-Write-Modell* unterscheidet lediglich zwischen *Lese-* und *Schreiboperationen* auf Datenobjekten $x \in D$. Das bedeutet im Kontext des Graphenlabors, dass beispielsweise das Ermitteln des Alphaknotens einer Kante e und das Ermitteln eines Attributwerts von e verallgemeinert als Leseoperationen auf der Kante (dem Datenobjekt) e interpretiert werden. Analog werden das Setzen des Alphaknotens für e und das Setzen eines Attributwerts von e als Schreiboperationen zusammengefasst. Bei dieser Betrachtungsweise wird vor allem von der *Semantik* und *Wirkung* der auf dem Graphen und den Graphenele-

menten ausführbaren Operationen abstrahiert. Es erfolgt lediglich eine „oberflächliche“ Interpretation der Operationen, welche eine Gruppierung in Lese- und Schreiboperationen zur Folge hat.

Die Berücksichtigung der Semantik von Operationen im Graphenlabor ist vor allem im Kontext der Bestimmung von Konfliktsituationen (siehe Abschnitt 3.6.2.2, Seite 45) interessant. So müssen zwei parallel ausgeführte Schreiboperationen auf einem gleichen Datenobjekt *x* nicht notwendigerweise in Konflikt stehen. Beispielsweise verursachen das parallele Setzen des Alphaknotens für eine Kante *e* durch eine Transaktion T_1 und das Setzen eines Attributwerts von *e* durch eine Transaktion T_2 keine Konflikte, obwohl beide Transaktionen Schreiboperationen auf dem selben Datenobjekt *e* durchführen. Näheres dazu wird in Abschnitt 4.2.1 erläutert.

Eine simple Aufteilung der Operationen in Lese- und Schreiboperationen wie im Read-Write-Modell ist im Kontext des Graphenlabors also unzureichend. Darüberhinaus muss die Semantik der jeweiligen Operationen (auch im Zusammenhang mit anderen parallel auszuführenden Operationen) in Betracht gezogen werden.

4.1.3.2 Identifikation von Datenobjekten im Graphenlabor

Die Identifikation der Datenobjekte im Graphenlabor erfolgt mit Hilfe der Definition 1.1 für TGraphen (Seite 5), dem in Abbildung 1.3 (Seite 9) dargestellten Meta-Schema und der aktuellen Implementierung von JGraLab. Im Folgenden werden alle für das Graphenlabor relevanten Datenobjekte aufgelistet. Basierend auf den Erkenntnissen aus Kapitel 2 und speziell aus Abschnitt 2.1.4 (Seite 17) wird für jedes Datenobjekt die entsprechende Repräsentation in der Implementierung von JGraLab (zur besseren Übersicht wiederholend) aufgezeigt.

- Ein **TGraph G**.

  G wird im JGraLab durch die Klasse *GraphImpl* umgesetzt.

- Die **Knotenmenge V**.

  V wird im JGraLab durch das Array *vertex* in der Klasse *GraphImpl* repräsentiert, welches Referenzen auf die einzelnen Knoteninstanzen enthält. Das Feld *vCount* in der Klasse *GraphImpl* speichert die Anzahl der Knoten in *V*.

- Die **Kantenmenge E**.

  E wird im JGraLab durch die Arrays *edge* und *revEdge* in der Klasse *GraphImpl* repräsentiert, welches Referenzen auf die einzelnen Kanteninstanzen in normaler

bzw. in umgekehrter Ausrichtung enthält. Das Feld *eCount* der Klasse *GraphImpl* speichert die Anzahl der Kanten in *E*.

- Die **Anordnung der Knoten Vseq**.

  Die Klasse *GraphImpl* kennt durch die Referenzen *firstVertex* und *lastVertex* den ersten bzw. letzten Knoten in *Vseq*. Durch die Attribute *nextVertex* und *prevVertex* in der Klasse *VertexImpl* hält jeder Knoten Referenzen auf seinen Nachfolge- bzw. Vorgängerknoten in *Vseq*.

- Die **Anordnung der Kanten Eseq**.

  Die Klasse *GraphImpl* kennt durch die Referenzen *firstEdge* und *lastEdge* die erste bzw. letzte Kante in *Eseq*. Durch die Attribute *nextEdge* und *prevEdge* in der Klasse *EdgeImpl* hält jede Kante Referenzen auf seine Nachfolge- bzw. Vorgängerkante in *Eseq*.

- Ein **Knoten v** mitsamt seiner **Inzidenzliste Λseq(v)**.

  Ein Knoten *v* wird im JGraLab durch die Klasse *VertexImpl* umgesetzt. Die Klasse *VertexImpl* kennt durch die Referenzen *firstIncidence* und *lastIncidence* die erste bzw. letzte Inzidenz in Λ*seq*(v). Durch die Attribute *nextIncidence* und *prevIncidence* in der Klasse *IncidenceImpl* hält jede Inzidenz Referenzen auf seine Nachfolge- bzw. Vorgängerinzidenz in Λ*seq*(v).

- Eine **Kante e** mitsamt ihrem **Start**- und **Endknoten**.

  Eine Kante *e* wird im JGraLab durch die Klassen *EdgeImpl* und *ReversedEdgeImpl* in ihrer normalen bzw. ihrer umgekehrten Ausrichtung repräsentiert. Die jeweilige Oberklasse *IncidenceImpl* speichert im Attribut *incidentVertex* den Start- bzw. Endknoten von *e*.

- Ein **Attribut** eines Graphen, eines Knotens oder einer Kante.

  Im JGraLab können die aus einem Schema generierten Klassen Attribute *primitiver (int, long, double, boolean, Enum)* und *komplexer (String, Set, List, Map, Record)* Datentypen enthalten. Diese Attribute werden im Kontext des Transaktionskonzepts als eigene Datenobjekte betrachtet.

- Die **Freispeicherliste** für **Knoten** und **Kanten**.

  Die Klasse *GraphImpl* besitzt mit den Attributen *freeVertexList* und *freeEdgeList* sowohl eine Freispeicherliste für Knoten als auch eine Freispeicherliste für Kanten.

4.1.3.3 Lese- und Schreiboperationen im Graphenlabor

Im Kontext des Transaktionskonzepts werden im Folgenden die elementaren Lese- und Schreiboperationen im JGraLab ermittelt. Dabei werden die privaten (klasseninternen) Methoden bei der Betrachtung außen vor gelassen.

In Abbildung 4.1 werden die (elementaren) Leseoperationen in JGraLab dargestellt. Hierbei werden sowohl die entsprechenden Methoden der Interfaces *Graph*, *Vertex* und *Edge* als auch die *getter*-Methoden der aus dem Schema *MotorwayMapSchema* (siehe Abbildung 1.4, Seite 11) generierten Interfaces gezeigt.

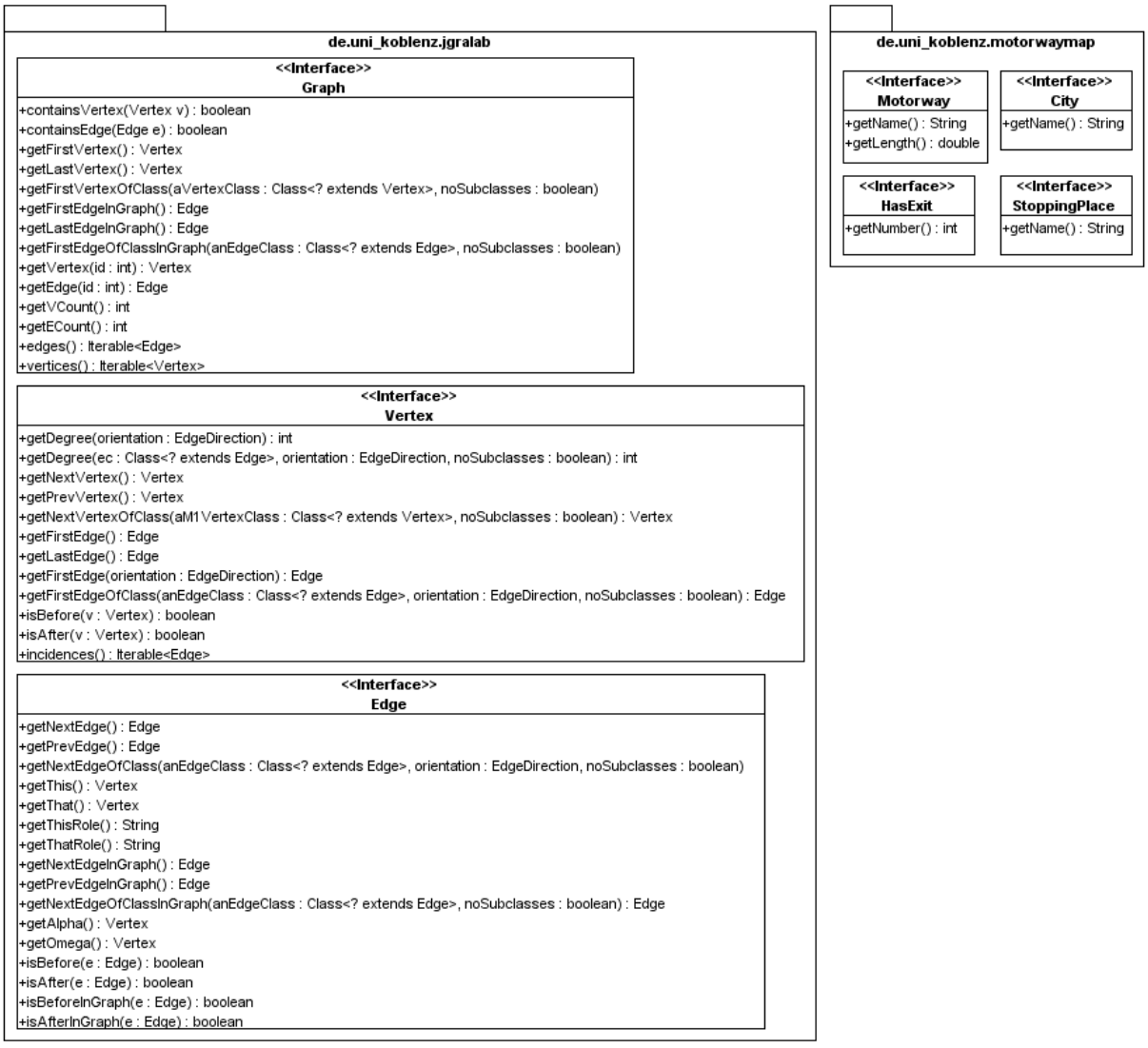


Abbildung 4.1: Relevante Leseoperationen im Graphenlabor

Analog dazu werden in Abbildung 4.2 die (elementaren) Schreiboperationen der Klassen *GraphImpl*, *VertexImpl* und *EdgeImpl* und die *setter*-Methoden der aus dem Schema *MotorwayMapSchema* generierten Interfaces dargestellt.

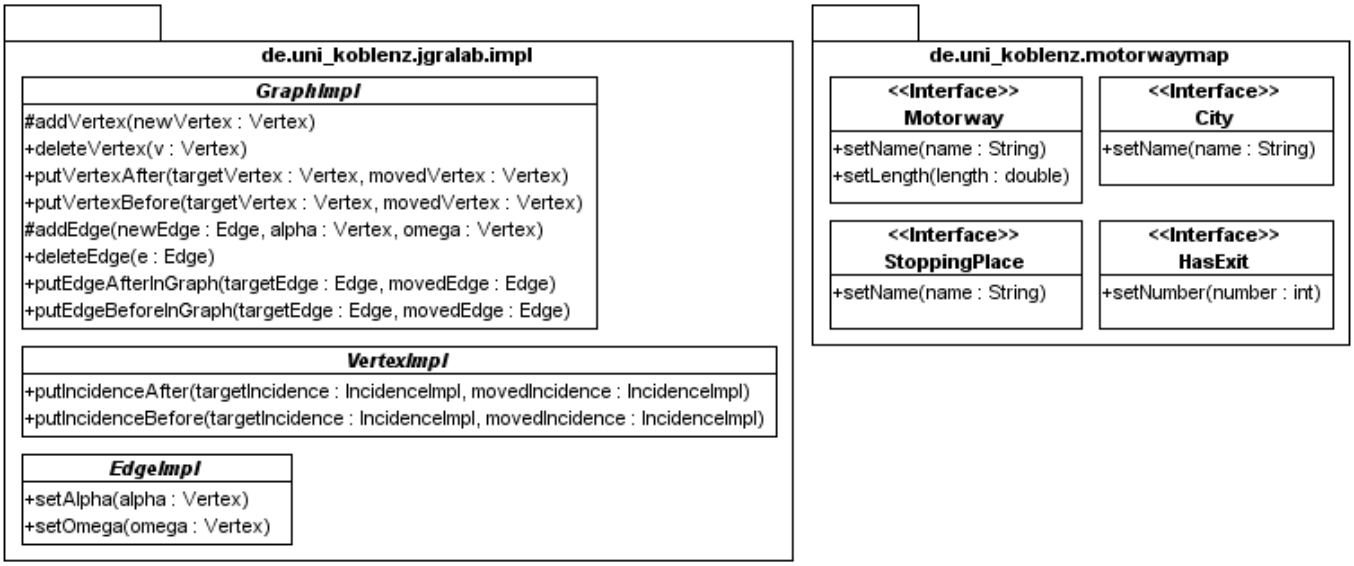


Abbildung 4.2: Relevante Schreiboperationen im Graphenlabor

Im Hinblick auf die Bestimmung von Konfliktsituationen (siehe Abschnitt 4.2.1) ist vor allem die Betrachtung der Schreiboperationen interessant, da letztlich nur Änderungen auf Datenobjekten zu Konflikten führen können. Aus diesem Grund werden nachfolgend die in Abbildung 4.2 gezeigten Schreiboperationen aus dem Paket *de.uni_koblenz.jgralab.impl* erläutert. Dabei werden auch die durch die Operationsausführung betroffenen Datenobjekte identifiziert.

- **addVertex(Vertex newVertex)**

  fügt den neuen Knoten *newVertex* in den Graphen *G* ein und verändert somit die Datenobjekte *V* und *Vseq*. Zudem wird die Freispeicherliste der Knoten verändert.

- **deleteVertex(Vertex v)**

  löscht den Knoten *v* aus *G*. Da auch alle zu *v* inzidenten Kanten mitgelöscht werden, sind von der Operation neben den Datenobjekten *V* und *Vseq* auch *E* und *Eseq* betroffen. Zudem werden die Inzidenzlisten aller Knoten v_i modifiziert, für die v_i ≠ v gilt und die mindestens eine zu *v* inzidente Kante als eigene Inzidenz besitzen. Die Freispeicherliste der Kanten wird ebenfalls verändert.

- **putVertexAfter(Vertex targetVertex, Vertex movedVertex)** und **putVertexBefore(Vertex targetVertex, Vertex movedVertex)**

  setzt den Knoten *movedVertex* als Nachfolge- bzw. Vorgängerknoten des Knotens *targetVertex* in *Vseq*. Die Knotenreihenfolge in *Vseq* wird verändert.

- **addEdge(Edge newEdge, Vertex alpha, Vertex omega)**

  fügt die neue Kante *newEdge* mit den bereits in *G* vorhandenen Startknoten *alpha* und dem Endknoten *omega* in *G* ein. Die von der Operationsausführung betroffenen Datenobjekte sind *E*, *Eseq*, Λ*seq*(alpha) und Λ*seq*(omega). Zudem wird die Freispeicherliste der Kanten verändert.

- **deleteEdge(Edge e)**

  löscht die Kante *e* aus *G*. Direkt betroffen sind die Datenobjekte *E*, *Eseq*, Λ*seq*(alpha) und Λ*seq*(omega). Ist *e* eine Kompositionskante, werden auch der Startknoten *alpha* und der Endknoten *omega* aus *G* entfernt, wodurch auch bei den Datenobjekten *V* und *Vseq* Änderungen auftreten. Zudem wird die Freispeicherliste der Kanten verändert.

- **putEdgeAfterInGraph(Edge targetEdge, Edge movedEdge)** und **putEdgeBeforeInGraph(Edge targetEdge, Edge movedEdge)**

  setzt die Kante *movedEdge* als Nachfolge- bzw. Vorgängerkante der Kante *targetEdge* in *Eseq*. Die Kantenreihenfolge in *Eseq* wird verändert.

- **putIncidenceAfter(IncidenceImpl targetIncidence, IncidenceImpl movedIncidence)** und **putIncidenceBefore(IncidenceImpl targetIncidence, IncidenceImpl movedIncidence)**

  setzt für den entsprechenden Knoten *v* die Inzidenz *movedIncidence* als Nachfolge- bzw. als Vorgängerinzidenz der Inzidenz *targetIncidence* in Λ*seq*(v). Die Inzidenzreihenfolge in Λ*seq*(v) wird verändert.

- **setAlpha(Vertex alpha)** und **setOmega(Vertex omega)**

  legt den Knoten *alpha* als Startknoten bzw. den Knoten *omega* als Endknoten der die Operation ausführenden Kanteninstanz *e* fest. Durch diese Operationen werden sowohl die Datenobjekte Λ*seq*(alpha) bzw. Λ*seq*(omega) als auch Λ*seq*(oldAlpha) bzw. Λ*seq*(oldOmega) verändert, wobei *oldAlpha* und *oldOmega* den alten Start- bzw. Endknoten vor der Operationsausführung für die Kante *e* darstellen.

4.2 Die Umsetzung der Synchronisation-Komponente im JGraLab

Die folgenden Abschnitte behandeln die Umsetzung der *Synchronisation*-Komponente im JGraLab. Dabei wird zunächst untersucht, welche Konfliktsituationen im Kontext des Graphenlabors ohne Synchronisation auftreten können (siehe Abschnitt 4.2.1). Anschließend werden die in Abschnitt 3.6 (Seite 42) vorgestellten Synchronisationsverfahren hinsichtlich ihrer Tauglichkeit und Umsetzbarkeit im JGraLab bewertet (siehe Abschnitt 4.2.2). Basierend auf den Ergebnissen dieser Bewertung wird ein Konzept für ein Synchronisationsverfahren im JGraLab ausgearbeitet (siehe Abschnitt 4.2.3).

4.2.1 Bestimmung von Konfliktsituationen

In diesem Abschnitt werden die im JGraLab auftretenden Konfliktsituationen dargestellt. Hierfür erfolgt für die in Abschnitt 4.1.3.2 identifizierten Datenobjekte eine Analyse der jeweils denkbaren Konfliktsituationen anhand beispielhafter Szenarien. Alle darin auftretenden Transaktionen greifen auf gemeinsame globale Datenobjekte zu. Dabei wird angenommen, dass keinerlei Synchronisation zwischen den beteiligten Transaktionen beim Zugriff auf die einzelnen Datenobjekte stattfindet. An geeigneten Stellen wird erläutert, inwiefern parallel ausgeführte Schreiboperationen auf dem gleichen Datenobjekt zusammengeführt werden können.

Im Folgenden wird der TGraph G als Datenobjekt außen vor gelassen, da G letztlich alle anderen Datenobjekte beinhaltet. Jedoch werden die Attribute von G (gemeinsam mit den Attributen der Knoten in V und der Kanten in E) berücksichtigt. In den Szenarien wird G mit dem Bezeichner g referenziert.

Um alle möglichen Konfliktsituationen erfassen zu können, werden folgende Systematiken angewendet:

- Handelt es sich bei dem Datenobjekt um eine *Menge*, wird das *Hinzufügen* und das *Löschen* von Elementen in bzw. aus dieser Menge betrachtet.
- Stellt das Datenobjekt eine *Liste* dar, wird neben dem Hinzufügen und dem Löschen auch das *Ändern der Reihenfolge* der Elemente innerhalb der Liste analysiert.
- In den restlichen Fällen wird das Überschreiben von einzelnen Datenwerten berücksichtigt.

4.2.1.1 Konflikte im Zusammenhang mit V

Um mögliche auftretende Konfliktsituationen im Zusammenhang mit der Knotenmenge *V* zu erläutern, wird zunächst das Szenario in Tabelle 4.1 verwendet.

Dabei wird angenommen, dass für die Knotenmenge V = {v1, v2, v3, v4, v5} vor dem *BOT* der Transaktionen T_1 und T_2 gilt. Nach dem *COMMIT* von T_2 ist V = {v1, v3, v4, v5, v6, v7}. Setzt man voraus, dass die Knoten *v6* und *v7* unterschiedliche *IDs* zugewiesen bekommen, ist für das gezeigte Szenario kein Konflikt aufgetreten. *V* enthält nach dem *COMMIT* von T_2 sowohl die von T_1 als auch die von T_2 durchgeführten Änderungen.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	g.addVertex(v6)	
4.		g.addVertex(v7)
5.	v2 = g.getVertex(2)	
6.	g.deleteVertex(v2)	
7.	COMMIT	
8.		COMMIT

Tabelle 4.1: Konfliktfreie parallele Schreiboperationen auf *V*

Ein Konflikt entsteht in Tabelle 4.1 also nur, falls den Knoten *v6* und *v7* identische *IDs* vom System zugewiesen werden.

Ein unlösbarer Konflikt tritt in Tabelle 4.2 nach dem COMMIT von T_2 auf, da T_1 im 6. Schritt den Knoten *v7* aus *V* löscht, der innerhalb von T_2 im 4. Schritt zu *V* hinzugefügt wurde.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	g.addVertex(v6)	
4.		g.addVertex(v7)
5.	v7 = g.getVertex(7)	
6.	g.deleteVertex(v7)	
7.	COMMIT	
8.		COMMIT

Tabelle 4.2: Unlösbarer Konflikt für *V*

4.2.1.2 Konflikte im Zusammenhang mit E

Die Überlegungen für *V* gelten analog auch für die Kantenmenge *E*. Für das Szenario in Tabelle 4.3 wird zu Beginn *E* = {e1, e2, e3, e4, e5} angenommen.

Nach dem *COMMIT* von T_2 gilt für die Kantenmenge *E* = {e1, e3, e4, e5, e6, e7}. Falls die Kanten *e6* und *e7* unterschiedliche *IDs* zugewiesen bekommen, entsteht hier keine Konfliktsituation. Erhalten die Kanten *e6* und *e7* jedoch identische *IDs*, tritt ein Konflikt auf.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	v1 = g.getVertex(1)	
4.	v2 = g.getVertex(2)	
5.	g.addEdge(e6, v1, v2)	
6.		v1 = g.getVertex(1)
7.		v3 = g.getVertex(3)
8.		g.addEdge(e7, v1, v3)
9.	e2 = g.getEdge(2)	
10.	g.deleteEdge(e2)	
11.	COMMIT	
12.		COMMIT

Tabelle 4.3: Konfliktfreie parallele Schreiboperationen auf *E*

Ein unlösbarer Konflikt tritt in Tabelle 4.4 nach dem *COMMIT* von T_2 auf, da T_1 im 10. Schritt die Kante *e7* aus *E* löscht, die von T_2 im 8. Schritt zu *E* hinzugefügt wurde.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	v1 = g.getVertex(1)	
4.	v2 = g.getVertex(2)	
5.	g.addEdge(e6, v1, v2)	
6.		v1 = g.getVertex(1)
7.		v3 = g.getVertex(3)
8.		g.addEdge(e7, v1, v3)
9.	e7 = g.getEdge(7)	
10.	g.deleteEdge(e7)	
11.	COMMIT	
12.		COMMIT

Tabelle 4.4: Unlösbarer Konflikt für *E* durch das Löschen einer Kante

Ein unlösbarer Konflikt (*Lost Update*) wird in Tabelle 4.5 nach dem *COMMIT* von T_2 festgestellt. Dieser tritt auf, da T_1 im 5. Schritt die Kante *e6* mit dem Starknoten *v1* hinzufügt, den T_2 im 7. Schritt löscht.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	v1 = g.getVertex(1)	
4.	v2 = g.getVertex(2)	
5.	g.addEdge(e6, v1, v2)	
6.		v1 = g.getVertex(1)
7.		g.deleteVertex(v1)
8.	COMMIT	
9.		COMMIT

Tabelle 4.5: Unlösbarer Konflikt für *E* durch das Löschen eines Knotens

4.2.1.3 Konflikte im Zusammenhang mit Vseq

Um mögliche Konfliktsituationen im Zusammenhang mit *Vseq* zu erläutern, wird zunächst das Szenario in Tabelle 4.1 verwendet. Vor dem *BOT* der Transaktionen T_1 und T_2 ist *Vseq* = <v1, v2, v3, v4, v5>. Hier tritt keine Konfliktsituation für *Vseq* auf. Nach dem *COMMIT* von T_2 gilt *Vseq* = <v1, v3, v4, v5, v6, v7>.

Tabelle 4.6 zeigt hingegen eine unlösbare Konfliktsituation für *Vseq*. Vor dem *BOT* von T_1 gilt auch hier *Vseq* = <v1, v2, v3, v4, v5>. T_1 setzt den Vorgängerknoten von *v4* im 6. Schritt auf *v2*, so dass *Vseq* = <v1, v2, v4, v3, v5> nach dem COMMIT von T_1 gelten müsste. Da T_2 jedoch den Knoten *v1* im 8. Schritt als Vorgängerknoten von *v4* setzt (*Vseq* = <v1, v4, v2, v3, v5>), kommt es nach dem *COMMIT* von T_2 zu einem *Lost Update* der in T_1 durchgeführten Änderung. Die Änderungen von T_1 und T_2 können nicht zusammengeführt werden.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	v2 = g.getVertex(2)	
4.		v1 = g.getVertex(1)
5.	v4 = g.getVertex(4)	
6.	g.putVertexAfter(v2, v4)	
7.		v4 = g.getVertex(4)
8.		g.putVertexAfter(v1, v4)
9.	COMMIT	
10.		COMMIT

Tabelle 4.6: Unlösbare Konfliktsituation für *Vseq* - Beispiel 1

Das in Tabelle 4.7 dargestellte Szenario führt ebenfalls zu einem unlösbaren Konflikt. T_2 ändert im 8. Schritt die Position von *v2* in *Vseq*, wodurch auch hier die von T_1 im 6. Schritt durchgeführte Änderung zum *Lost Update* wird.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	v2 = g.getVertex(2)	
4.		v2 = g.getVertex(2)
5.	v4 = g.getVertex(4)	
6.	g.putVertexAfter(v2, v4)	
7.		v5 = g.getVertex(5)
8.		g.putVertexAfter(v5, v2)
9.	COMMIT	
10.		COMMIT

Tabelle 4.7: Unlösbare Konfliktsituation für *Vseq* - Beispiel 2

Die Konflikte in Tabelle 4.6 und Tabelle 4.7 treten auf, da beide Transaktionen *explizit* den Vorgängerknoten von *v4* in *Vseq* durch die Methode *putVertexAfter* ändern.

In Tabelle 4.8 gilt zu Beginn *Vseq* = <v1, v2, v3, v4, v5>. Auch in diesem Szenario wird zuerst in T_2 und anschließend in T_1 der Vorgängerknoten von *v4* geändert. Allerdings erfolgt es in T_1 *explizit* durch den Aufruf der Methode *putVertexAfter* (7. Schritt), während es in T_2 *implizit* dadurch ausgelöst wird, dass der Knoten *v3* aus *G* gelöscht wird (5. Schritt). Nach dem *COMMIT* von T_2 gilt *Vseq* = <v1, v4, v2, v5>. Dabei entsteht kein Konflikt, da T_2 den Vorgängerknoten von *v4* nicht explizit geändert hat.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	v1 = g.getVertex(1)	
4.		v3 = g.getVertex(3)
5.		g.deleteVertex(v3)
6.	v4 = g.getVertex(4)	
7.	g.putVertexAfter(v1, v4)	
8.	COMMIT	
9.		COMMIT

Tabelle 4.8: Implizite Änderung der Knotenreihenfolge in *Vseq* durch T_2

Zusammenführen lassen sich die in Tabelle 4.9 gezeigten Änderungen. Nach dem *COMMIT* von T_2 gilt *Vseq* = <v1, v3, v2, v4, v5>. Da *Vseq* sowohl die Änderungen von T_1 als auch die von T_2 enthält, tritt kein Konflikt auf.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	v2 = g.getVertex(2)	
4.		v1 = g.getVertex(1)
5.	v4 = g.getVertex(4)	
6.	g.putVertexAfter(v2, v4)	
7.		v3 = g.getVertex(3)
8.		g.putVertexAfter(v1, v3)
9.	COMMIT	
10.		COMMIT

Tabelle 4.9: Zusammenführbare Änderungen in *Vseq*

Ein unlösbarer Konflikt tritt in Tabelle 4.10 auf, da T_2 im 7. Schritt den Knoten *v4* löscht. Die von T_1 durchgeführte Änderung im 6.Schritt wird zum *Lost Update*. Dieses tritt auf, da die zuerst durch das *COMMIT* von T_1 *festgeschriebene* Änderung für den Knoten *v4* durch das Löschen von *v4* in T_2 und dessen *COMMIT* verloren geht.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	v2 = g.getVertex(2)	
4.		v4 = g.getVertex(4)
5.	v4 = g.getVertex(4)	
6.	g.putVertexAfter(v2, v4)	
7.		g.deleteVertex(v4)
8.	COMMIT	
9.		COMMIT

Tabelle 4.10: *Lost Update* für *Vseq* durch das Löschen eines Knotens

Tabelle 4.11 zeigt eine weitere unlösbare Konfliktsituation. Nach dem *COMMIT* von T_1 tritt ein Konflikt auf, da im 8. Schritt die Methode *putVertexAfter* mit dem Knoten *v4* als zweiten Parameter aufgerufen wird. *v4* existiert jedoch nach dem *COMMIT* von T_2 nicht mehr. Der Knoten *v4* stellt ein Phantom für T_1 dar.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	v2 = g.getVertex(2)	
4.		v4 = g.getVertex(4)
5.	v4 = g.getVertex(4)	
6.		g.deleteVertex(v4)
7.		COMMIT
8.	g.putVertexAfter(v2, v4)	
9.	COMMIT	

Tabelle 4.11: *Phantomproblem* für *Vseq* durch das Löschen eines Knotens

4.2.1.4 Konflikte im Zusammenhang mit Eseq

Die Überlegungen für *Vseq* gelten analog für *Eseq*. Geht man von dem Anfangszustand *Eseq* = <e1, e2, e3, e4, e5> aus, gilt nach dem Ausführen von T_1 und T_2 in Tabelle 4.3 *Eseq* = <e1, e3, e4, e5, e6, e7>.

Analog zu Tabelle 4.6 und Tabelle 4.7 entstehen in Tabelle 4.12 bzw. Tabelle 4.13 unlösbare Konfliktsituationen durch das explizite Ändern der Vorgängerkante von *e4* in *Eseq* mittels der Methode *putEdgeAfterInGraph*.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e2 = g.getEdge(2)	
4.	e4 = g.getEdge(4)	
5.	g.putEdgeAfterInGraph(e2, e4)	
6.		e1 = g.getEdge(1)
7.		e4 = g.getEdge(4)
8.		g.putEdgeAfterInGraph(e1, e4)
9.	COMMIT	
10.		COMMIT

Tabelle 4.12: Unlösbare Konfliktsituation für *Eseq* - Beispiel 1

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e2 = g.getEdge(2)	
4.	e4 = g.getEdge(4)	
5.	g.putEdgeAfterInGraph(e2, e4)	
6.		e5 = g.getEdge(5)
7.		e2 = g.getEdge(2)
8.		g.putEdgeAfterInGraph(e5, e2)
9.	COMMIT	
10.		COMMIT

Tabelle 4.13: Unlösbare Konfliktsituation für *Eseq* - Beispiel 2

Analog zu Tabelle 4.8 tritt in Tabelle 4.14 kein Konflikt auf, da T_2 die Vorgängerkante von *e4* nur *implizit* durch das Löschen der Kante *e3* ändert.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e1 = g.getEdge(1)	
4.		e3 = g.getEdge(3)
5.		g.deleteEdge(e3)
6.	e4 = g.getEdge(4)	
7.	g.putEdgeAfterInGraph(e1, e4)	
8.	COMMIT	
9.		COMMIT

Tabelle 4.14: Implizite Änderung der Kantenreihenfolge in *Eseq* durch T_2

Die in Tabelle 4.15 durchgeführten Änderungen der Transaktionen T_1 und T_2 lassen sich zusammenführen, so dass nach dem *COMMIT* von T_2 hier *Eseq* = <e1, e3, e2, e4, e5> gilt.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e2 = g.getEdge(2)	
4.	e4 = g.getEdge(4)	
5.	g.putEdgeAfterInGraph(e2, e4)	
6.		e1 = g.getEdge(1)
7.		e3 = g.getEdge(3)
8.		g.putEdgeAfterInGraph(e1, e3)
9.	COMMIT	
10.		COMMIT

Tabelle 4.15: Zusammenführbare Änderungen in *Eseq*

In Tabelle 4.16 entsteht nach dem *COMMIT* von T_2 ein unlösbarer Konflikt durch das *Lost Update* der von T_1 im 5. Schritt durchgeführten Änderung.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e2 = g.getEdge(2)	
4.	e4 = g.getEdge(4)	
5.	g.putEdgeAfterInGraph(e2, e4)	
6.		e4 = g.getEdge(4)
7.		g.deleteEdge(e4)
8.	COMMIT	
9.		COMMIT

Tabelle 4.16: *Lost Update* für *Eseq* durch das Löschen einer Kante

In Tabelle 4.17 stellt die Kante *e4* ein Phantom für T_1 dar.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e2 = g.getEdge(2)	
4.		e4 = g.getEdge(4)
5.	e4 = g.getEdge(4)	
6.		g.deleteEdge(e4)
7.		COMMIT
8.	g.putEdgeAfterInGraph(e2, e4)	
9.	COMMIT	

Tabelle 4.17: *Phantomproblem* für *Eseq* durch das Löschen einer Kante

4.2.1.5 Konflikte im Zusammenhang mit einem Knoten *v* und Λseq(v)

Für die folgenden Szenarien wird für die Inzidenzliste eines Knotens *v1* der Anfangszustand Λ*seq*(v1) = < (e1,out) , (e2,out), (e3,out), (e4,out), (e5, out)> angenommen.

Die Änderungen der in Tabelle 4.3 (Seite 68) dargestellten Transaktionen führen auch für Λ*seq*(v1) nicht zum Konflikt. Nach dem *COMMIT* von T_2 gilt Λ*seq*(v1) = <(e1,out), (e3,out), (e4,out), (e5,out), (e6,out), (e7,out)>.

Eine ähnliche Situation wie in Tabelle 4.3 wird in Tabelle 4.18 gezeigt. Hier wird zusätzlich angenommen, dass die Kanten *e6* und *e7* bereits vor dem *BOT* von T_1 existieren und deren Startknoten *v2* ist. Das Setzen des Startknotens für *e6* und *e7* im 5. bzw. 8. Schritt mittels *setAlpha* haben den gleichen Effekt für den Knoten *v1* wie die Aufrufe von *addEdge* in Tabelle 4.3. Das Ändern des Startknotens für *e2* im 11. Schritt, entspricht dem Löschen einer Inzidenz aus Λ*seq*(v1). Auch für dieses Szenario treten keine Konflikte auf.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e6 = g.getEdge(6)	
4.	v1 = g.getVertex(1)	
5.	e6.setAlpha(v1)	
6.		e7 = g.getEdge(7)
7.		v1 = g.getVertex(1)
8.		e7.setAlpha(v1)
9.	e2 = g.getEdge(2)	
10.	v2 = g.getVertex(2)	
11.	e2.setAlpha(v2)	
12.	COMMIT	
13.		COMMIT

Tabelle 4.18: Hinzufügen und Löschen von Inzidenzen in Λ*seq*(v1) durch *setAlpha*

Analog zu *Vseq* und *Eseq* treten in Tabelle 4.19 und Tabelle 4.20 unlösbare Konflikte aufgrund expliziter Änderungen der Vorgängerinzidenz von *(e4,out)* auf.

Schritt	T_1	T_2
1.	BOT	
2.	v1 = g.getVertex(1)	
3.		BOT
4.		v1 = g.getVertex(1)
5.	e2 = g.getEdge(2)	
6.	e4 = g.getEdge(4)	
7.	v1.putIncidenceAfter(e2, e4)	
8.		e1 = g.getEdge(1)
9.		e4 = g.getEdge(4)
10.		v1.putIncidenceAfter(e1, e4)
11.	COMMIT	
12.		COMMIT

Tabelle 4.19: Unlösbare Konfliktsituation für Λ*seq*(v1) - Beispiel 1

Schritt	T_1	T_2
1.	BOT	
2.	v1 = g.getVertex(1)	
3.		BOT
4.		v1 = g.getVertex(1)
5.	e2 = g.getEdge(2)	
6.	e4 = g.getEdge(4)	
7.	v1.putIncidenceAfter(e2, e4)	
8.		e5 = g.getEdge(5)
9.		e2 = g.getEdge(2)
10.		v1.putIncidenceAfter(e5, e2)
11.	COMMIT	
12.		COMMIT

Tabelle 4.20: Unlösbare Konfliktsituation für Λ*seq*(v1) - Beispiel 2

In Tabelle 4.21 tritt kein Konflikt auf, da T_2 die Vorgängerinzidenz von *(e4, out)* in Λ*seq*(v1) nur implizit löscht.

Schritt	T_1	T_2
1.	BOT	
2.	v1 = g.getVertex(1)	
3.		BOT
4.		e3 = g.getEdge(3)
5.		g.deleteEdge(3)
6.	e1 = g.getEdge(1)	
7.	e4 = g.getEdge(4)	
8.	v1.putIncidenceAfter(e1, e4)	
9.	COMMIT	
10.		COMMIT

Tabelle 4.21: Implizite Änderung der Inzidenzreihenfolge in Λ*seq*(v1) durch T_2

Die in Tabelle 4.22 durchgeführten Änderungen der Transaktionen T_1 und T_2 lassen sich zusammenführen, so dass nach dem *COMMIT* von T_2 für Λseq(v1) = <(e1,out) (e3,out) (e2,out), (e4,out), (e5,out)> gilt.

Schritt	T_1	T_2
1.	BOT	
2.	v1 = g.getVertex(1)	
3.		BOT
4.		v1 = g.getVertex(1)
5.	e2 = g.getEdge(2)	
6.	e4 = g.getEdge(4)	
7.	v1.putIncidenceAfter(e2, e4)	
8.		e1 = g.getEdge(1)
9.		e3 = g.getEdge(3)
10.		v1.putIncidenceAfter(e1, e3)
11.	COMMIT	
12.		COMMIT

Tabelle 4.22: Zusammenführbare Änderungen in Λseq(v1)

Für Λseq(v1) kann das Löschen von Graphenelementen analog zu *Vseq* und *Eseq* zu *Lost Updates* oder *Phantomproblemen* führen. Im Folgenden wird nur der *Lost Update* als Konfliktsituation betrachtet.

In Tabelle 4.23 geht die von T_1 im 7. Schritt durchgeführte Änderung an Λseq(v1) verloren, da T_2 den Knoten *v1* im 8. Schritt aus *G* entfernt.

Schritt	T_1	T_2
1.	BOT	
2.	v1 = g.getVertex(1)	
3.		BOT
4.		v1 = g.getVertex(1)
5.	e2 = g.getEdge(2)	
6.	e4 = g.getEdge(4)	
7.	v1.putIncidenceAfter(e2, e4)	
8.		g.deleteVertex(v1)
9.	COMMIT	
10.		COMMIT

Tabelle 4.23: *Lost Update* für Λseq(v1) durch das Löschen von *v1*

In Tabelle 4.24 geht die von T_1 im 7. Schritt durchgeführte Änderung an Λseq(v1) verloren, da T_2 die Kante *e2* im 8. Schritt aus *G* entfernt.

Schritt	T_1	T_2
1.	BOT	
2.	v1 = g.getVertex(1)	
3.		BOT
4.		e2 = g.getEdge(2)
5.	e2 = g.getEdge(2)	
6.	e4 = g.getEdge(4)	
7.	v1.putIncidenceAfter(e2, e4)	
8.		g.deleteEdge(e2)
9.	COMMIT	
10.		COMMIT

Tabelle 4.24: *Lost Update* fuer Λseq(v1) durch das Löschen einer Kante

4.2.1.6 Konflikte im Zusammenhang mit einer Kante e

Für eine einzelne Kante können ebenfalls Konflikte auftreten. In Tabelle 4.25 wird der Startknoten der Kante *e1* durch die Transaktion T_1 zunächst auf den Knoten *v1* gesetzt. Nach dem *COMMIT* von T_2 wird der Startknoten von *e1* auf *v2* gesetzt, wodurch die Änderung von T_1 überschrieben wird. Es kommt zu einem *Lost Update*.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e1 = g.getEdge(1)	
4.		e1 = g.getEdge(1)
5.	v1 = g.getVertex(1)	
6.	e1.setAlpha(v1)	
7.		v2 = g.getVertex(2)
8.		e1.setAlpha(v2)
9.	COMMIT	
10.		COMMIT

Tabelle 4.25: Unlösbare Konfliktsituation für eine Kante *e1*

In Tabelle 4.26 setzt T_1 den Knoten *v1* als Startknoten für die Kante *e1*, während T_2 den Endknoten von *e1* auf *v2* abändert. Obwohl beide Transaktionen parallele Schreiboperationen auf dem gleichen Datenobjekt *e1* durchführen, kommt es in diesem Fall nicht zu einem Konflikt.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e1 = g.getEdge(1)	
4.		e1 = g.getEdge(1)
5.	v1 = g.getVertex(1)	
6.	e1.setAlpha(v1)	
7.		v2 = g.getVertex(2)
8.		e1.setOmega(v2)
9.	COMMIT	
10.		COMMIT

Tabelle 4.26: Zusammenführbare Änderungen für eine Kante *e1*

Auch für eine Kante kann das Löschen von Graphenelementen zu *Lost Updates* oder *Phantomproblemen* führen. Im Folgenden wird nur der *Lost Update* als Konfliktsituation betrachtet. In Tabelle 4.27 entsteht nach dem *COMMIT* von T_2 ein *Lost Update* der von T_1 im 6. Schritt durchgeführten Änderung, da *e1* in T_2 im 7. Schritt aus *G* entfernt wird.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e1 = g.getEdge(1)	
4.		e1 = g.getEdge(1)
5.	v1 = g.getVertex(1)	
6.	e1.setAlpha(v1)	
7.		g.deleteEdge(e1)
8.	COMMIT	
9.		COMMIT

Tabelle 4.27: *Lost Update* für eine Kante *e1* durch das Löschen von *e1*

In Tabelle 4.28 entsteht der *Lost Update* dadurch, dass T_2 den Knoten *v1* im 7. Schritt entfernt, welcher in T_1 im 6. Schritt als Startknoten für die Kante *e1* gesetzt wurde.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	e1 = g.getEdge(1)	
4.		v1 = g.getVertex(1)
5.	v1 = g.getVertex(1)	
6.	e1.setAlpha(v1)	
7.		g.deleteVertex(v1)
8.	COMMIT	
9.		COMMIT

Tabelle 4.28: *Lost Update* für eine Kante *e1* durch das Löschen des Startknotens

4.2.1.7 Konflikte im Zusammenhang mit Attributen

Im Folgenden werden Konfliktsituationen im Zusammenhang mit Attributen von Graphen, Knoten und Kanten anhand des String-Attributs *name* eines Knotens *v1* analysiert. In Tabelle 4.29 wird in T_1 das Attribut *name* von *v1* auf den Wert "v1" gesetzt. Diese Änderung geht nach dem *COMMIT* von T_2 verloren und wird zum *Lost Update*.

Bei Attibuten eines primitiven Datentyps oder vom Typ *String* entstehen in solchen Situationen unlösbare Konflikte, da die Attributwerte unwiderruflich überschrieben werden. Bei nicht-primitiven Datentypen wie *List*, *Set*, *Map* oder *Record* können unter Umständen Änderungen zusammengeführt werden, so dass keine Konfliktsituationen entstehen. Näheres dazu findet sich in Abschnitt 4.2.3.9.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.		v1 = g.getVertex(1)
4.	v1= g.getVertex(1)	
5.	v1.setName(“v1“)	
6.		v1.setName(“v2“)
7.	COMMIT	
8.		COMMIT

Tabelle 4.29: Konfliktsituation durch das Setzen von Attributwerten

In Tabelle 4.30 tritt ein *Lost Update* der in T_1 im 5. Schritt durchgeführten Änderung an dem Attribut *name* auf, da T_2 die zugehörige Instanz *v1* im 6. Schritt aus *G* entfernt.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.		v1 = g.getVertex(1)
4.	v1= g.getVertex(1)	
5.	v1.setName(“v1“)	
6.		g.deleteVertex(v1)
7.	COMMIT	
8.		COMMIT

Tabelle 4.30: *Lost Update* für ein Attribut durch das Löschen der zugehörigen Instanz

In Tabelle 4.31 stellt der Knoten *v1* ein Phantom für T_1 dar, so dass die im 7. Schritt durchgeführte Änderung am Attribut *name* ungültig ist.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.		v1 = g.getVertex(1)
4.	v1= g.getVertex(1)	
5.		g.deleteVertex(v1)
6.		COMMIT
7.	v1.setName(“v1“)	
8.	COMMIT	

Tabelle 4.31: *Phantomproblem* für ein Attribut durch das Löschen der zugehörigen Instanz

4.2.1.8 Dirty Reads durch das ABORT von Transaktionen

Durch das *ABORT* einer Transaktion T_i kann es in einer parallel laufenden Transaktion T_j zu *Dirty Reads* kommen, falls T_j auf Graphenelemente zugreift, welche in T_i hinzugefügt wurden.

Tabelle 4.32 zeigt eine solche Situation. T_2 fügt im 5. Schritt dem Graphen *G* die Kante *e5* hinzu. Im 6. Schritt greift T_1 auf diese zu und ändert dessen Endknoten im 8. Schritt auf den Knoten *v5* um. Da jedoch die Transaktion T_2 im 10. Schritt ein *ABORT* durchführt und somit Schritt 5 zurückgenommen werden muss, hat T_1 Operationen auf einer Kante *e5* durchgeführt, die nicht persistent gespeichert wurde.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.		v1 = g.getVertex(1)
4.		v2 = g.addVertex(2)
5.		g.addEdge(e5, v1, v2)
6.	e5 = g.getEdge(5)	
7.	v5 = g.getVertex(5)	
8.	e5.setOmega(v5)	
9.	COMMIT	
10.		ABORT

Tabelle 4.32: *Dirty Read* durch das *ABORT* einer Transaktion

4.2.1.9 Konflikte im Zusammenhang mit der Freispeicherliste

Ein unsynchronisierter Zugriff auf die Freispeicherlisten der Knoten und Kanten kann zu Konflikten führen. Tabelle 4.33 stellt eine solche Konfliktsituation für die Freispeicherliste der Knoten dar.

Schritt	T_1	T_2
1.	BOT	
2.		BOT
3.	v5 = g.getVertex(5)	
4.	g.deleteVertex(v5)	
5.		g.addVertex(v6)
6.	ABORT	
7.		COMMIT

Tabelle 4.33: Konfliktsituation für die Freispeicherliste der Knoten

Im 4. Schritt löscht T_1 den Knoten *v5* mit der ID *5*. Es wird angenommen, dass die *5* als nächste zu vergebene ID in der Freispeicherliste der Knoten vermerkt wird. Daraufhin erzeugt T_2 einen neuen Knoten *v6*, dem die eben freigewordene *5* als ID zugewiesen wird. Da jedoch T_1 ein *ABORT* durchführt und der Knoten *v5* letztlich nicht gelöscht wird, enthält *V* nach dem *COMMIT* von T_2 zwei Knoten *v5* und *v6* mit identischer *ID*.

4.2.2 Beurteilung der Synchronisationsverfahren

Um die in Abschnitt 4.2.1 (Seite 66) gezeigten Konfliktsituationen im Graphenlabor behandeln zu können, ist eine Synchronisation der parallelen Zugriffe auf die identifizierten Datenobjekte notwendig. Im Folgenden werden die einzelnen in Abschnitt 3.6 (Seite 42) erläuterten Synchronisationsverfahren hinsichtlich ihrer Tauglichkeit und Umsetzbarkeit im JGraLab beurteilt.

4.2.2.1 Sperrbasierte Synchronisation

Im Kontext des Graphenlabors ist die Verwendung der *sperrbasierten Synchronisation* (siehe Abschnitt 3.6.4, Seite 49) ungeeignet. Wie in Abschnitt 4.1.3.1 (Seite 60) erläutert, lösen Schreiboperationen im Graphenlabor in vielen Fällen Änderungen an mehreren Datenobjekten aus. Beim Einsatz von Sperren müssten jeweils alle betroffenen Datenobjekte exklusiv gesperrt werden. Da beispielsweise ein Knoten *v* nur mitsamt seiner Inzidenzen als ganzes Datenobjekt betrachtet werden kann, müssen vor der Ausführung einer Schreiboperation auf *v* nicht nur *v* selbst, sondern auch alle zu *v* inzidenten Kanten mit einer exklusiven Sperre versehen werden. So sind Szenarien wahrscheinlich, in denen durch die Ausführung weniger Schreiboperationen ein Großteil der Datenobjekte in einem TGraphen exklusiv gesperrt werden.

Setzt man in diesem Zusammenhang das (strenge) *2PL*-Protokoll voraus, werden die von einer Transaktion T_i gehaltenen (exklusiven) Sperren meist erst bei dessen *COMMIT* oder *ABORT* freigegeben. Parallel laufende Transaktionen T_j, die auf gesperrte Datenobjekte zugreifen möchten, müssen unter Umständen sehr lange auf die Freigabe der entsprechenden Sperren warten. Die betroffenen Transaktionen sind während dieser Wartezeit in ihrer Ausführung blockiert. Um aber eine performante interaktive Nutzung des JGraLabs (zum Beispiel im Kontext eines Editors für TGraphen) zu ermöglichen, müssen Transaktionen möglichst schnell und effizient ohne lange Wartezeiten durchführbar sein.

Ein weiteres Argument gegen die Umsetzung der sperrbasierten Synchronisation ist das mögliche Auftreten von *Deadlocks*. Entscheidet man sich für den Einsatz von Sperren im Graphenlabor, müssen entsprechende Mechanismen bereitgestellt werden, die Deadlocks (im Voraus) vermeiden. Zudem entsteht bei der sperrbasierten Synchronisation ein zusätzlicher Verwaltungsaufwand durch die Vergabe, das Setzen und die Freigabe von Sperren.

Insgesamt kann die sperrbasierte Synchronisation im Kontext des Graphenlabors als zu *restriktiv* und *ineffizient* eingestuft werden.

4.2.2.2 Zeitstempel-basierende Synchronisation

Die *Zeitstempel-basierende Synchronisation* (siehe Abschnitt 3.6.5, Seite 52) als zweites pessimistisches Synchronisationsverfahren verzichtet auf Sperren und setzt dafür Zeitstempel ein. Der Verwaltungsaufwand für Sperren entfällt also, stattdessen müssen Lese- und Schreibzeitstempel für Datenobjekte und Zeitstempel für den Ausführungsbeginn von Transaktionen mitgeführt werden.

Die für die Zeitstempel-basierende Synchronisation definierten Regeln führen in vielen Fällen zum Abbruch und Neustart einer Transaktion. Wie bereits in Abschnitt 4.1.2 (Seite 58) erwähnt, kann eine Transaktion im Graphenlabor jedoch nicht neugestartet werden.

Die zeitstempel-basierende Synchronisation ist zwar insgesamt weniger restriktiv als die sperrbasierte Synchronisation, ist jedoch im Graphenlabor in der vorgestellten Form aufgrund des notwendigen und im Graphenlabor nicht durchführbaren Neustarts einer Transaktion nicht praktisch umsetzbar.

4.2.2.3 Optimistische Synchronisation

Die *optimistische Synchronisation* (siehe Abschnitt 3.6.6, Seite 52) ist für die Umsetzung eines Synchronisationsverfahrens im Graphenlabor besser geeignet. Hierbei kann eine Transaktion T_i zunächst ihre Operationen unabhängig und isoliert von allen anderen (parallel laufenden) Transaktionen T_j (lokal) ausführen. Erst nachdem T_i ihr *COMMIT* ausführt, wird überprüft, ob Konsistenzverletzungen oder Konfliktsituationen durch die in T_i durchgeführten Änderungen aufgetreten sind.

Diese Vorgehensweise ermöglicht in vielen Fällen ein schnelles und effizientes Ausführen von Transaktionen. Das Zurücksetzen einer Transaktion T_i (beispielsweise nach dem Ausführen der *ABORT*-Operation) ist dabei relativ einfach durchzuführen, da sich die durch T_i durchgeführten Änderungen während ihrer Lesephase (zunächst) nur auf den lokalen Puffer auswirken. Erst in der (sicheren) Schreibphase werden die jeweiligen Änderungen persistent gespeichert.

Allerdings muss auch hier wie bei der Zeitstempel-basierenden Synchronisation angemerkt werden, dass das Zurücksetzen und anschließende Neustarten einer Transaktion im Graphenlabor nicht unterstützt wird.

4.2.2.4 Mehrversionen-Synchronisation

Die Anwendung der *Mehrversionen-Synchronisation* (siehe Abschnitt 3.6.7, Seite 55) ist im Kontext des Graphenlabors im Zusammenspiel mit der optimistischen Synchronisation denkbar. Die Verwaltung von verschiedenen Versionen einzelner Datenobjekte in einem zentralen *Versionen-Pool* durch die *Mehrversionen-Synchronisation* kann dabei zur Umsetzung von lokalen Puffern für die einzelnen Transaktionen im Rahmen der optimistischen Synchronisation verwendet werden.

4.2.3 Konzept für ein Synchronisationsverfahren im JGraLab

Im Folgenden wird ein Synchronisationsverfahren für JGraLab erläutert, welches Mechanismen der *optimistischen Synchronisation* und der *Mehrversionen-Synchronisation* kombiniert.

4.2.3.1 Temporäre und persistente Versionen von Datenobjekten

In Anlehnung an die *Mehrversionen-Synchronisation* werden für alle in Abschnitt 4.1.3.2 (Seite 61) für das Graphenlabor identifizierten Datenobjekte (außer den Freispeicherlisten) verschiedene Versionen in einem *Versionen-Pool* verwaltet. So erzeugen Änderungen an einem Datenobjekt *x* neue Versionen im Versionen-Pool. Dabei werden *persistente* Versionen von *temporären* Versionen unterschieden. Eine (neue) persistente Version x_p eines Datenobjekts *x* entsteht beim erfolgreichen *COMMIT* einer Transaktion T_i, die während ihrer Lesephase mindestens eine Schreiboperation auf *x* durchgeführt hat. Eine temporäre Version x_t von *x* wird für eine Transaktion T_i erzeugt, sobald T_i eine Schreiboperation auf *x* durchführt. x_t ist nur innerhalb der sie erzeugenden Transaktion T_i gültig und sichtbar. Durch das Mitführen der (nur innerhalb der jeweiligen Transaktion gültigen) temporären Versionen wird im Versionen-Pool der im Kontext der optimistischen Synchronisation notwendige *lokale Puffer* für eine Transaktion T_i umgesetzt.

Für das Anlegen von temporären Versionen sind zwei Varianten denkbar:

- In der ersten Variante erzeugt nur die erste Änderung an einem Datenobjekt *x* innerhalb der Lesephase einer Transaktion T_i eine temporäre Version x_t für T_i. Alle danach folgenden Änderungen an dem Datenobjekt *x* durch T_i werden auf der zuerst angelegten temporären Version x_t durchgeführt. Im weiteren Verlauf der Lesephase von T_i werden in der Regel keine weiteren temporären Versionen vom Datenobjekt *x* für T_i erstellt.

- In der zweiten Variante führt jede innerhalb der Lesephase einer Transaktion T_i durchgeführte Änderung an einem Datenobjekt *x* zur Erzeugung einer neuen temporären Version x_t für T_i im Versionen-Pool.

Da die erste Variante einen geringeren Speicherplatzbedarf zur Folge hat, wird im Folgenden von dessen Verwendung ausgegangen.

Für eine im Versionen-Pool angelegte Version eines Datenobjekts *x* müssen im Allgemeinen die folgenden Informationen gespeichert werden:

- Eine eindeutige *Versionsnummer*.
- Der *Wert* des Datenobjekts, welcher zu der jeweiligen Version gehört.
- Ein *Flag*, der anzeigt, ob es sich um eine *temporäre* oder *persistente* Version von *x* handelt.
- Eine Referenz auf die Transaktion, welche die neue Version des Datenobjekts erzeugt hat. Diese Information ist lediglich für temporäre Versionen relevant.

4.2.3.2 Versionenzähler

Für die eindeutige Vergabe von Versionsnummern für Datenobjekte einer Grapheninstanz werden Versionenzähler benötigt. Im Folgenden werden zwei Varianten erläutert:

- In der ersten Variante wird ein (globaler) Versionenzähler *versionCounter* eingeführt. Beim Systemstart besitzt dieser den Wert 0. Bei jeder Erzeugung einer neuen temporären oder persistenten Version x_i eines Datenobjekts *x* wird auf *versionCounter* zugegriffen. Dabei wird *versionCounter* um den Wert 1 erhöht. Der neue Wert von *versionCounter* wird x_i als Versionsnummer zugeordnet. Zu einem Zeitpunkt darf höchstens eine Transaktion auf *versionCounter* zugreifen.
- In der zweiten Variante existiert neben einem (globalen) Versionenzähler *persistentVersionCounter* ein (lokaler) Versionenzähler *temporaryVersionCounter*. *persistentVersionCounter* besitzt beim Systemstart den Wert 0 und ist lediglich beim Anlegen von neuen persistenten Versionen relevant. So wird bei jeder Erzeugung einer neuen persistenten Version x_p eines Datenobjekts *x* der Wert von *persistentVersionCounter* um 1 erhöht und x_p als Versionsnummer zugeordnet. Der Zugriff muss (analog zu *versionCounter*) synchronisiert ablaufen.

Jede Transaktion T_i besitzt zusätzlich einen (lokalen) Versionenzähler *temporaryVersionCounter*, welcher lediglich beim Anlegen von neuen temporären Versionen innerhalb von T_i relevant ist. T_i.*temporaryVersionCounter* besitzt beim *BOT* von T_i den Wert 0. Bei jeder Erzeugung einer neuen temporären Version x_t eines Datenobjekts *x* durch T_i wird der Wert des zugehörigen Versionenzählers T_i.*temporaryVersionCounter* um 1 erhöht. Der neue Wert von T_i.*temporaryVersionCounter* wird x_t als Versionsnummer zugeordnet. Da nur T_i auf ihren zugehörigen Versionenzähler T_i.*temporaryVersionCounter* zugreift, sind keine Synchronisationsmaßnahmen beim Zugriff auf diesen notwendig.

Im Folgenden wird von der Verwendung der zweiten Variante ausgegangen. Durch die Einführung eines eigenen Versionenzählers für das Erzeugen persistenter Versionen ist es (einfacher) festzustellen, ob nach dem *COMMIT* einer Transaktion T_i eine Konfliktüberprüfung notwendig ist. So kann auf diese verzichtet werden, falls die Werte von *persistentVersionCounter* zum *BOT*- und *COMMIT*-Zeitpunkt von T_i identisch sind. Bei der Verwendung eines einzigen Versionenzählers *versionCounter* für temporäre und persistente Änderungen ist diese Überprüfung nicht (so einfach) möglich. Die Durchführung der Konfliktüberprüfung wird in Abschnitt 4.2.3.9 detaillierter erläutert.

Für jedes Datenobjekt *x* muss zu jedem Zeitpunkt die (höchste) gültige persistente Versionsnummer (im Folgenden als *persistentVersion* bezeichnet) ermittelbar sein. Dies ist vor allem im Kontext der Konfliktüberprüfung hilfreich.

4.2.3.3 Umsetzung der Versionierung

In diesem Abschnitt folgt eine Erläuterung der Versionierungsumsetzung für die einzelnen in Abschnitt 4.1.3.2 (Seite 61) identifizierten Datenobjekte.

- Für die Knotenmenge *V* muss sichergestellt werden, dass für jede neu angelegte Version eine Kopie des Arrays *vertex* der Klasse *GraphImpl* erzeugt wird. Zudem muss beachtet werden, dass bei jeder neu angelegten Version von *vertex* auch eine entsprechende Version für das Feld *vCount* erstellt wird.

- Die Überlegungen für *V* gelten analog für die Kantenmenge *E* und somit für die Arrays *edge* und *revEdge* und das Feld *eCount* der Klasse *GraphImpl*.

- Für *Vseq* werden zum einen die Referenzen auf die erste und die letzte Knoteninstanz *firstVertex* bzw. *lastVertex* in der Klasse *GraphImpl* versioniert. Davon unabhängig werden zum anderen für einen Knoten *v* die Referenzen *nextVertex* und *prevVertex* auf den Nachfolge- bzw. Vorgängerknoten von *v* in *Vseq* in der Klasse *VertexImpl* versioniert.

Werden durch den *COMMIT* einer Transaktion T_i die Referenzen *nextVertex* oder *prevVertex* für *v* geändert, wird das jeweilige Feld *v.nextVertex.persistentVersion* bzw. *v.prevVertex.persistentVersion* nur für den Fall aktualisiert, dass der Nachfolge- bzw. Vorgängerknoten von *v* in *Vseq explizit* durch T_i geändert wurde. *Explizit* bedeutet, dass innerhalb von T_i die Methoden *putVertexAfter* oder *putVertexBefore* entsprechend mit dem Knoten *v* als Parameter ausgeführt werden. Beispielsweise wird durch *putVertexAfter(v1, v)* der Vorgängerknoten von *v* in *Vseq* explizit auf den Knoten *v1* gesetzt.

Um (persistente) Änderungen an *Vseq* als Ganzes (besser) nachvollziehbar zu machen, wird ein versioniertes *long*-Feld *vertexListVersion* für jeden Graphen eingeführt. Die zu einem Zeitpunkt aktuelle persistente Version von *vertexListVersion* repräsentiert die Versionsnummer der aktuellen persistenten Version von *Vseq*. Die temporären Versionen von *vertexListVersion* sind nur innerhalb der zugehörigen Transaktionen relevant und repräsentieren daher Versionsnummern von *Vseq*, welche nur für die jeweiligen Transaktionen gelten.

- Die Überlegungen für *Vseq* gelten analog für *Eseq* und somit für die Felder *firstEdge* und *lastEdge* der Klasse *GraphImpl* und die Felder *nextEdge* und *prevEdge* der Klasse *EdgeImpl*.

  Auch hier wird für die Referenzen *nextEdge* und *prevEdge* einer Kante *e* das Feld *e.nextEdge.persistentVersion* bzw. *e.prevEdge.persistentVersion* jeweils nur bei expliziten Änderungen durch entsprechende Aufrufe der Methoden *putEdgeAfterInGraph* und *putEdgeBeforeInGraph* aktualisiert.

  Zudem wird analog zu *vertexListVersion* ein versioniertes *long*-Feld *edgeListVersion* für jeden Graphen eingeführt.

- Die Überlegungen für *Vseq* und *Eseq* gelten analog für Λseq(v) eines Knotens *v* und somit für die Felder *firstIncidence* und *lastIncidence* der Klasse *VertexImpl* und die Felder *nextIncidence* und *prevIncidence* der Klasse *IncidenceImpl*.

  Für die Referenzen *nextIncidence* und *prevIncidence* einer Inzidenz *i* wird das Feld *i.nextIncidence.persistentVersion* bzw. *i.prevIncidence.persistentVersion* jeweils nur bei expliziten Änderungen durch entsprechende Aufrufe der Methoden *putIncidenceAfter* und *putIncidenceBefore* aktualisiert.

  Zudem wird ein versioniertes *long*-Feld *incidenceListVersion* für jeden Knoten eingeführt.

- Da die Informationen zu dem Start- und dem Endknoten einer Kante *e* in zwei verschiedenen Instanzen (vom Typ *EdgeImpl* bzw. *ReversedEdgeImpl*) gespeichert sind, werden diese jeweils unabhängig voneinander versioniert. Dabei werden jeweils Versionen von *incidentVertex* der Klasse *IncidenceImpl* angelegt.

- Für jedes Attribut eines Graphen, eines Knotens oder einer Kante werden jeweils eigene (temporäre und persistente) Versionen angelegt. Um in diesem Kontext Attribute eines primitiven Datentyps (*int*, *long*, *double*, *boolean*) referenzierbar zu machen, müssen diese als Objekte behandelt werden können.

4.2.3.4 Zugriff auf die Freispeicherliste

Im Kontext des Transaktionskonzepts werden *global* gültige Freispeicherlisten eingeführt. Dies bedeutet, dass alle Transaktionen auf die gleichen Freispeicherlisten zugreifen. Es existieren also lediglich persistente Versionen der Freispeicherlisten. Um Konfliktsituationen für Freispeicherlisten (siehe Abschnitt 4.2.1.9, Seite 82) zu vermeiden, werden die folgenden Regeln definiert:

- Möchte eine Transaktion T_i vor ihrem *COMMIT* ein neues Graphenelement erzeugen, greift diese auf die jeweilige globale Freispeicherliste zu, um eine ID für das Graphenelement zu erhalten. Die Freispeicherliste muss entsprechend aktualisiert werden, so dass die gerade vergebene ID im weiteren Verlauf (zunächst) nicht erneut vergeben werden kann.

- Löscht eine Transaktion T_i vor ihrem *COMMIT* ein Graphenelement, wird die jeweilige globale Freispeicherliste *nicht* aktualisiert. Mit dieser Maßnahme wird die in Tabelle 4.33 (Seite 82) dargestellte Konfliktsituation durch den *ABORT* einer Transaktion vermieden.

- Erst beim *COMMIT* oder *ABORT* einer Transaktion T_i erfolgt eine *Aktualisierung* der Freispeicherlisten. So werden beim *COMMIT* die IDs der durch T_i gelöschten Graphenelemente und beim *ABORT* die durch T_i angeforderten IDs wieder freigegeben.

  Beim *COMMIT* von T_i darf eine ID *n* einer Kante oder eines Knotens nur freigegeben werden, falls in keiner anderen parallel laufenden (Read-Write-)Transaktion T_j eine Kante bzw. ein Knoten mit der ID *n* existiert. So wird verhindert, dass die ID *n* in einer Transaktion T_j vergeben werden kann, in der je nach Kontext bereits eine Kante oder ein Knoten mit der ID *n* existiert. Ansonsten würde das bereits vorhandene Graphenelement in T_j „überschrieben“ werden.

  Alle IDs, die beim *COMMIT* von T_i nicht freigegeben werden können, werden den im Graphen verwalteten Listen *edgeIndexesToBeFreed* bzw. *vertexIndexesToBeFreed* hinzugefügt. Beim nächsten erfolgreichen *COMMIT*-Versuch einer Transaktion T_j wird erneut versucht, die in *edgeIndexesToBeFreed* und *vertexIndexesToBeFreed* gespeicherten IDs (nach den oben genannten Regeln) freizugeben.

- Der Zugriff auf eine Freispeicherliste muss atomar und synchronisiert durchgeführt werden. Folglich kann zu einem Zeitpunkt höchstens eine Transaktion auf die jeweilige Freispeicherliste zugreifen.

4.2.3.5 Isolierte Sicht von Transaktionen

Jede Transaktion T_i besitzt während ihrer Ausführung eine gleichbleibende und unabhängig von den restlichen Transaktionen isolierte Sicht auf die Datenobjekte. Dafür merkt sich T_i bei ihrem *BOT* den aktuellen Wert des (globalen) Versionenzählers *persistentVersionCounter* im Feld *persistentVersionAtBot*. Eine Transaktion T_i hat während ihrer gesamten Ausführung für jedes Datenobjekt *x* lediglich Zugriff auf die zugehörige persistente Version x_p mit der höchsten Versionsnummer *vn*, für die *vn* ≤ *persistentVersionAtBot* gilt. Zudem kann T_i für jedes Datenobjekt *x* auf alle durch T_i selbst erzeugten temporären Versionen x_t zugreifen. Für T_i sind für jedes Datenobjekt *x* die durch andere parallel ablaufende Transaktionen T_j erzeugten temporären oder persistenten Versionen x_i nicht sichtbar.

Fügt eine Transaktion T_i einem Graphen *G* ein neues Graphenelement *e* hinzu, ist dieses also (zunächst) nur innerhalb von T_i gültig. Für alle anderen parallel laufenden Transaktionen T_j existiert *e* nicht. So wird verhindert, dass eine Transaktion ein Graphenelement löschen kann, welches von einer anderen parallel ablaufenden Transaktion hinzugefügt wurde. Durch die isolierte Sicht von Transaktionen können zudem keine *Dirty Reads* (siehe Abschnitt 3.6.1.2, Seite 43) und *Inconsistent Reads* (siehe Abschnitt 3.6.1.3, Seite 44) auftreten.

4.2.3.6 Die Lesephase

Die Lesephase einer Transaktion T_i beginnt mit dem Ausführen ihrer *BOT*-Operation. In Anlehnung an die *optimistische Synchronisation* führt jede Transaktion T_i zunächst ihre Lese- und Schreiboperationen ohne weitere Synchronisation auf einem nur für T_i gültigen *lokalen Puffer* aus, welcher im Versionen-Pool verwaltet wird. Dabei muss sichergestellt werden, dass die *BOT*-Operation atomar und synchronisiert ausgeführt wird. Dies bedeutet, dass keine andere Transaktion T_j existieren darf, die parallel ihr *COMMIT* durchführt. Hiermit wird vor allem die parallele Erzeugung neuer persistenter Versionen von Datenobjekten während der Ausführungsdauer der *BOT*-Operation von T_i verhindert.

Erst wenn T_i ihre *COMMIT*-Operation durchführt, erfolgt - angelehnt an die *optimistische Synchronisation* - ein Übergang in die *Validierungsphase*. Handelt es sich bei T_i um eine *Read-Only-Transaktion*, wird T_i nach dem *COMMIT* beendet ohne ihre Validierungsphase ausführen zu müssen.

4.2.3.7 Die Validierungsphase

Beim COMMIT einer (*Read-Write-*)Transaktion T_i muss zunächst überprüft werden, ob durch die in T_i durchgeführten Änderungen Konflikte aufgetreten sind. Eine Konfliktüberprüfung ist dabei nur notwendig, falls zum *COMMIT*-Zeitpunkt T_i.*persistent VersionAtBot* < *persistentVersionCounter* gilt, also mindestens eine andere (*Read-Write-*) Transaktion T_j ein erfolgreiches *COMMIT* vor dem *COMMIT* von T_i durchgeführt hat. Mögliche Konflikte sind dabei entweder *Lost Updates* oder *Phantomprobleme*.

Um *Lost Updates* feststellen zu können, muss T_i für jedes in T_i geänderte Datenobjekt *x* die Werte *x.persistentVersion* und T_i.*persistentVersionAtBot* vergleichen. Gilt für ein Datenobjekt *x*, dass *x.persistentVersion* > T_i.*persistentVersionAtBot*, ist (mindestens) eine neue persistente Version von *x* seit dem *BOT* von T_i durch (mindestens) eine andere Transaktion T_j erzeugt worden. Dies ist zudem ein Hinweis darauf, dass ein *Lost Update* für *x* aufgetreten sein kann. Unterscheiden sich auch die Werte der letzten (zum *COMMIT*-Zeitpunkt von T_i gültigen) persistenten Version x_p und der zuletzt in T_i angelegten temporären Version x_t, wird ein *Lost Update* für *x* festgestellt.

Ein *Phantomproblem* tritt auf, falls T_i Operationen auf Knoten oder Kanten ausführt, die zum *COMMIT*-Zeitpunkt von T_i nicht mehr in dem Graphen *G* existieren.

Wird (mindestens) ein Konflikt festgestellt, kann die Durchführung der Validierungsphase für T_i beendet werden. Eine entsprechende Exception wird geworfen, um die Feststellung einer Konfliktsituation zu signalisieren. Wird die Validierung im Rahmen des *COMMITs* von T_i durchgeführt, wird der Übergang von T_i in die Schreibphase verweigert. Eine detaillierte Erläuterung zur Durchführung der Konfliktüberprüfung erfolgt in Abschnitt 4.2.3.8 und Abschnitt 4.2.3.9.

Das Ausführen der *COMMIT*-Operation einer Transaktion T_i muss atomar und synchronisiert ablaufen. Es muss sichergestellt werden, dass sich während der Validierungsphase einer Transaktion T_i keine weiteren Transaktionen T_j in der Schreibphase befinden. Ansonsten könnten während der Validierungsphase von T_i neue persistente Versionen von Datenobjekten entstehen, welche zur Ungültigkeit der Validierungsphase von T_i führen würden.

4.2.3.8 Mengen zur Protokollierung durchgeführter Änderungen

Um Konflikte für eine Transaktion T_i feststellen zu können, müssen die von T_i durchgeführten Änderungen protokolliert werden. Zur Protokollierung werden die im Folgenden erläuterten Mengen eingeführt:

- In der Liste *addedVertices*$_{T_i}$: iseq V werden Referenzen auf die in T_i hinzugefügten Knoteninstanzen mitgeführt.

- Die Liste $addedEdges_{T_i}$: iseq E enthält Referenzen auf die in T_i hinzugefügten Kanteninstanzen.

- In der Liste $deletedVertices_{T_i}$: iseq V wird protokolliert, welche Knoteninstanzen in T_i gelöscht worden sind.

  Wird eine Knoteninstanz *v* ∈ ran $addedVertices_{T_i}$ gelöscht, wird *v* aus der Liste $addedVertices_{T_i}$ enfernt, jedoch *nicht* der Liste $deletedVertices_{T_i}$ hinzugefügt. Alle Referenzen auf *v* müssen in den übrigen Mengen entfernt werden.

- Die Liste $deletedEdges_{T_i}$: iseq E enthält Referenzen auf die in T_i gelöschten Kanteninstanzen.

  Wird eine Kanteninstanz *e* ∈ ran $addedEdges_{T_i}$ gelöscht, wird *e* aus $addedEdges_{T_i}$ entfernt, jedoch *nicht* der Liste $deletedEdges_{T_i}$ hinzugefügt. Alle Referenzen auf *e* müssen in den übrigen Mengen entfernt werden.

- In der Menge $changedVseqVertices_{T_i}$: $\mathbb{P}$ {(v, p, m) | v ∈ *V* ∧ p ∈ {next, prev} ∧ m ∈ {true, false}} werden Referenzen auf Knoteninstanzen *v* gespeichert, deren Nachfolge- (*next*) bzw. Vorgängerknoten (*prev*) in *Vseq explizit* geändert wurden. Zudem wird vermerkt, ob *v* dabei seine Position in *Vseq* verändert (*true*) oder behalten hat (*false*). Letztere Information wird auch für die Durchführung der Schreibphase benötigt (siehe Abschnitt 4.2.3.12).

  Explizit bedeutet dabei, dass in T_i die Methoden *putVertexAfter* oder *putVertexBefore* der Klasse *GraphImpl* ausgeführt wurden. Beim Aufruf von *putVertexAfter(v1, v3)* werden beispielsweise die Tupel *(v1, next, false)* und *(v3, prev, true)* in die Menge $changedVseqVertices_{T_i}$ aufgenommen. Bei *putVertexBefore(v1, v3)* werden die Tupel *(v1, prev, false)* und *(v3, next, true)* hinzugefügt.

  $changedVseqVertices_{T_i}$ enthält dabei lediglich Tupel *(v, p, m)*, für die *v* ∉ ran $deletedVertices_{T_i}$ gilt.

- Analog zu $changedVseqVertices_{T_i}$ enthält die Menge $changedEseqEdges_{T_i}$: $\mathbb{P}$ {(e, p, m) | e ∈ *E* ∧ p ∈ {next, prev} ∧ m ∈ {true, false}} Referenzen auf Kanteninstanzen *e*, deren Nachfolge- bzw. Vorgängerkante in *Eseq explizit* geändert wurden.

  Dabei sind die Methoden *putEdgeAfterInGraph* und *putEdgeBeforeInGraph* der Klasse *GraphImpl* relevant.

  $changedEseqEdges_{T_i}$ enthält lediglich Tupel *(e, p, m)*, für die *e* ∉ ran $deletedEdges_{T_i}$ gilt.

- In der Menge $changedIncidences_{T_i} : V \nrightarrow \mathbb{P}$ {((e, d), p, m) | e ∈ *E* ∧ d ∈ {in, out} ∧ p ∈ {next, prev} ∧ m ∈ {true, false}} wird einem Knoten *v* eine Menge von Inzidenzen zugeordnet, deren Nachfolge- bzw. Vorgängerinzidenz in Λ*seq*(v) explizit geändert wurden.

  Dabei sind die Methoden *putIncidenceAfter* und *putIncidenceBefore* der Klasse *VertexImpl* relevant.

  Für alle *v* ∈ dom $changedIncidences_{T_i}$ gilt, dass *v* ∉ ran $deleted\,Vertices_{T_i}$. Für alle Tupel *((e, d), p, m)* ∈ ran $changedIncidences_{T_i}$ muss *e* ∉ ran $deletedEdges_{T_i}$ gelten.

- Die Menge $changedEdges_{T_i}$: ℙ {(e, i) | e ∈ *E* ∧ i ∈ {alpha, omega}} speichert Referenzen auf Kanteninstanzen, deren Start- (*alpha*) oder Endknoten (*omega*) geändert wurde.

  $changedEdges_{T_i}$ enthält nur Tupel *(e, i)*, für die *e* ∉ ran $deletedEdges_{T_i}$ gilt.

- In der Menge $changedAttributes_{T_i}$: *V* ∪ *E* ∪ {G} ↛ ℙ *A* werden Knoten *v* ∈ *V*, Kanten *e* ∈ *E* und dem Graphen *G* eine Menge von Attributen zugeordnet, deren Werte durch T_i geändert wurden. *A* repräsentiert dabei die Menge aller Attribute.

  Für jede Knoten- und Kanteninstanz *i* ∈ dom $changedAttributes_{T_i}$ muss gelten, dass *i* ∉ ran $deletedVertices_{T_i}$ für *i* ∈ *V* bzw. *i* ∉ ran $deletedEdges_{T_i}$ für *i* ∈ *E*.

  Um die Protokollierung auch für Attribute eines primitiven Datentyps zu ermöglichen, müssen diese (wie bereits erwähnt) als Objekte behandelt werden können.

4.2.3.9 Konfliktüberprüfung und -erkennung

Im Folgenden wird mit Hilfe von Pseudocode erläutert, wie die Konflikterkennung schrittweise für eine Transaktion T_i (nach dessen *COMMIT*) durchgeführt wird. Sobald (mindestens) ein Konflikt festgestellt wird, kann die Konfliktüberprüfung beendet werden.

Bei den Darstellungen im Pseudocode werden die folgenden Notationen für ein Datenobjekt *x* im Kontext der Transaktion T_i verwendet:

- $x_{p,COMMIT}$ liefert den Wert der persistenten Version von *x* zum *COMMIT*-Zeitpunkt von T_i.

- $x_{t,COMMIT}$ liefert den Wert der temporären Version von *x* in T_i zum *COMMIT*-Zeitpunkt von T_i.

- *x.pv()* liefert die zum COMMIT-Zeitpunkt von T_i für *x* gültige persistente Versionsnummer (*x.persistentVersion*).

pvAtBot() liefert den für T_i zum *BOT*-Zeitpunkt gespeicherten Wert von *persistentVersionCounter* (T_i.*persistentVersionAtBot*).

Für eine Knoten-, Kanten- oder Grapheninstanz *i* liefert der Aufruf *i.attr()* eine Menge zurück, welche Referenzen auf alle versionierten Attribute von *i* enthält.

Konflikte durch das Hinzufügen von Kanten

In Listing 4.1 wird für jede in T_i hinzugefügte Kante *e* ran $\in addedEdges_{T_i}$ überprüft, ob die zugehörigen Start- (*alpha*) und Endknoten (*omega*) beim *COMMIT* von T_i noch existieren (Zeile 5 bzw. 6). Ist dies für *alpha* und/ oder *omega* nicht der Fall, ist ein Konflikt für *e* aufgetreten. Bei der Überprüfung sind nur diejenigen Knoten relevant, die nicht innerhalb von T_i hinzugefügt worden sind.

```
1  forall(e in ran addedEdges_{T_i}) {
2    assert(e not in ran deletedEdges_{T_i});
3    Vertex alpha = e_{t,COMMIT}.getAlpha();
4    Vertex omega = e_{t,COMMIT}.getOmega();
5    if((alpha not in vertex_{p,COMMIT} && alpha not in ran addedVertices_{T_i}) ||
6       (omega not in vertex_{p,COMMIT} && omega not in ran addedVertices_{T_i}))
7      break; // (Phantom) alpha- or omega- vertex of e no longer valid
8  }
```

Listing 4.1: Konfliktüberprüfung für $addedEdges_{T_i}$

Konflikte durch das Löschen von Knoten

```
1  forall(v in ran deletedVertices_{T_i}) {
2    assert(v not in ran addedVertices_{T_i});
3    if(v in vertex_{p,COMMIT}) {
4      if(v.incidenceListVersion.pv() > pvAtBot())
5        break; // (Lost Update) - Λseq(v) modified since BOT
6      if((v.prevVertex.pv() > pvAtBot()) || (v.nextVertex.pv() > pvAtBot()))
7        break; // (Lost Update) - prev-/ nextVertex of v modified since BOT
8      forall(a in v.attr()) {
9        if(a.pv() > pvAtBot())
10         break; // (Lost Update) - attribute a modified since BOT
11     }
12   }
13 }
```

Listing 4.2: Konfliktüberprüfung für $deletedVertices_{T_i}$

Treten in Listing 4.1 keine Konflikte auf, wird in einem zweiten Schritt geprüft, ob durch das Löschen von Knoten in T_i *Lost Updates* auftreten (siehe Listing 4.2). Für jeden noch existierenden (Zeile 3) Knoten *v* ∈ ran $deletedVertices_{T_i}$ wird überprüft, ob sich Λ*seq(v)* (Zeile 4), der Vorgänger- oder Nachfolgeknoten (Zeile 6) von *v* in *Vseq* oder (mindestens) eines der Attribute von *v* (Zeile 9) seit dem *BOT* von T_i geändert haben. Wird (mindestens) eine solche Änderung für einen Knoten *v* erkannt, ist ein *Lost Update* für *v* (bzw. dem jeweiligen Attribut *a*) aufgetreten.

Konflikte durch das Löschen von Kanten

Konnten in Listing 4.2 keine Konflikte festgestellt werden, wird anschließend in Listing 4.3 überprüft, ob durch das Löschen von Kanten in T_i *Lost Updates* aufgetreten sind.

```
1  forall(e in ran deletedEdges_{T_i}) {
2    assert(e not in ran addedEdges_{T_i});
3    if(e in edge_{p,COMMIT}) {
4      if((e.getAlpha().pv() > pvAtBot()) || (e.getOmega().pv() > pvAtBot()))
5        break; // (Lost Update) - alpha- or omega of e changed since BOT
6      if((e.prevIncidence.pv() > pvAtBot()) ||
7         (e.nextIncidence.pv() > pvAtBot()))
8        break; // (Lost Update) - prev- or nextEdge of e changed since BOT
9      Edge re = e.getReversedEdge();
10     if((re.prevIncidence.pv() > pvAtBot()) ||
11        (re.nextIncidence.pv() > pvAtBot()))
12       break; // (Lost Update) - prev- or nextEdge of re changed since BOT
13     if((e.prevEdge.pv() > pvAtBot()) ||
14        (e.nextEdge.pv() > pvAtBot()))
15       break; // (Lost Update) - prev-/ nextEdge of e modified since BOT
16     forall(a in e.attr()) {
17       if(a.pv() > pvAtBot())
18         break; // (Lost Update) - attribute a modified since BOT
19     }
20   }
21 }
```

Listing 4.3: Konfliktüberprüfung für $deletedEdges_{T_i}$

Für jede noch existierende (Zeile 3) Kante *e* ∈ ran $deletedEdges_{T_i}$ wird zunächst überprüft, ob sich der Start- oder Endknoten (Zeile 4) von *e* seit dem *BOT* von T_i geändert hat. Anschließend erfolgt für die Kante *e* in normaler und umgekehrter Richtung die Überprüfung, ob sich die jeweilige Vorgänger- (Zeile 6 bzw. 10) oder Nachfolgeinzidenz (Zeile 7 bzw. 11) (explizit) geändert hat. Zudem werden mögliche Änderungen an der Vorgänger- (Zeile 13) bzw. Nachfolgekante (Zeile 14) von *e* in *Eseq* abgefragt. Zuletzt wird für *e* überprüft, ob sich (mindestens) eines ihrer Attribute seit dem *BOT* von T_i geändert hat (Zeile 17). Alle in Listing 4.3 feststellbaren Konflikte stellen *Lost Updates* dar.

Konflikte durch Änderungen der Knotenreihenfolge in *Vseq*

Wurden auch in Listing 4.3 keine Konflikte festgestellt, erfolgt in Listing 4.4 die Überprüfung auf Konflikte durch das (explizite) Ändern der Knotenreihenfolge in *Vseq*.

```
1  if(g.vertexListVersion.pv() > pvAtBot()) {
2    forall((v,p,m) in changedVseqVertices_{T_i}) {
3      assert(v not in ran deletedVertices_{T_i});
4      boolean conflictDetected = false;
5      if(v not in ran addedVertices_{T_i} && v not in vertex_{p,COMMIT})
6        break; // (Phantom) - v no longer valid
7      else {
8        switch(p) {
9          case "prev":
10           if((v.prevVertex_{t,COMMIT} not in vertex_{p,COMMIT}) &&
11              (v.prevVertex_{t,COMMIT} not in ran addedVertices_{T_i}))
12             conflictDetected = true; // (Phantom) - prevVertex
13           else if((v.prevVertex.pv() > pvAtBot()) &&
14                   (v.prevVertex_{t,COMMIT} != v.prevVertex_{p,COMMIT}))
15             // (Lost Update) - prevVertex changed since BOT
16             conflictDetected = true;
17           if(m == true) {
18             if(v.nextVertex_{p,COMMIT} != null &&
19                v.nextVertex_{p,COMMIT}.prevVertex_{p,COMMIT}.pv() > pvAtBot() &&
20                !((v.nextVertex_{p,COMMIT}, prev, {true|false})
21                   in changedVseqVertices_{T_i} &&
22                   v.nextVertex_{p,COMMIT} == v.nextVertex_{t,COMMIT}))
23               conflictDetected = true;
24           }
25           break;
26         case "next":
27           if((v.nextVertex_{t,COMMIT} not in vertex_{p,COMMIT}) &&
28              (v.nextVertex_{t,COMMIT} not in ran addedVertices_{T_i}))
29             conflictDetected = true; // (Phantom) - nextVertex
30           else if((v.nextVertex.pv() > pvAtBot()) &&
31                   (v.nextVertex_{t,COMMIT} != v.nextVertex_{p,COMMIT}))
32             // (Lost Update) - nextVertex changed since BOT
33             conflictDetected = true;
34           if(m == true) {
35             if(v.prevVertex_{p,COMMIT} != null &&
36                v.prevVertex_{p,COMMIT}.nextVertex_{p,COMMIT}.pv() > pvAtBot() &&
37                !((v.prevVertex_{p,COMMIT}, next, {true|false})
38                   in changedVseqVertices_{T_i} &&
39                   v.prevVertex_{p,COMMIT} == v.prevVertex_{t,COMMIT}))
40               conflictDetected = true;
41           }
42           break;
43       }
44     }
```

```
45      if(conflictDetected)
46        break;
47    }
48  }
```

Listing 4.4: Konfliktüberprüfung für $changedVseqVertices_{T_i}$

Die Überprüfung ist nur notwendig, falls sich *vertexListVersion* seit dem *BOT* von T_i geändert hat (Zeile 1). Ist dies der Fall, muss zunächst für jeden Knoten *v* festgestellt werden, ob *v* überhaupt noch existiert (Zeile 5). Existiert *v* nicht mehr, hat T_i Operationen auf einem Knoten durchgeführt, der beim *COMMIT* von T_i nicht mehr gültig ist. Der Knoten *v* stellt ein *Phantom* dar, so dass ein Konflikt festgestellt und die Konfliktüberprüfung beendet wird.

Ist ein Knoten *v* auch beim *COMMIT* von T_i noch gültig, muss je nach Kontext überprüft werden, ob der (temporäre) Vorgänger- (Zeilen 10-11) bzw. Nachfolgeknoten (Zeilen 27-28) von *v* noch eine gültige Instanz darstellt. Ist dies nicht der Fall, ist für *v* ein Konflikt aufgetreten, da es sich beim Vorgänger- bzw. Nachfolgeknoten um einen *Phantomknoten* handelt.

Ist der (temporäre) Vorgänger- bzw. Nachfolgeknoten gültig, muss anschließend je nach Kontext überprüft werden, ob Änderungen an der persistenten Version des Vorgänger- (Zeile 13) bzw. Nachfolgeknotens (Zeile 30) seit dem *BOT* von T_i aufgetreten sind und sich die jeweiligen Werte der temporären und der persistenten Version zum *COMMIT*-Zeitpunkt von T_i unterscheiden (Zeile 14 bzw. 31). Ist dies der Fall wird ein *Lost Update* für *v* festgestellt.

Kann kein *Lost Update* festgestellt werden, wird daraufhin zunächst überprüft, ob *v* seine Position in *Vseq* innerhalb von T_i geändert hat (Zeile 17 bzw. Zeile 34). Ist *v* zudem seit dem *BOT* von T_i explizit und persistent als Vorgänger- (Zeile 19) bzw. Nachfolgeknoten (Zeile 36) eines Knotens *w* gesetzt worden und unterscheiden sich *w* in der temporären und der persistenten Version von *Vseq* (Zeilen 20-22 bzw. 37-39), ist hier ein *Lost Update* aufgetreten.

Konflikte durch Änderungen der Kantenreihenfolge in *Eseq*

Wurden auch in Listing 4.4 keine Konflikte ermittelt, erfolgt in Listing 4.5 analog zu *Vseq* eine Überprüfung auf *Phantomprobleme* und *Lost Updates* im Bezug auf *Eseq*.

```
1  if(g.edgeListVersion.pv() > pvAtBot()) {
2    forall((e,p,m) in changedEseqEdges_{T_i}) {
3      assert(e not in ran deletedEdges_{T_i});
4      boolean conflictDetected = false;
5      if(e not in ran addedEdges_{T_i} && e not in edge_{p,COMMIT})
6        break; // (Phantom) - e no longer valid
```

```
7     else {
8       switch(p) {
9         case "prev":
10          if((e.prevEdge_{t,COMMIT} not in edge_{p,COMMIT}) &&
11             (e.prevEdge_{t,COMMIT} not in ran addedEdges_{T_i}))
12            conflictDetected = true; // (Phantom) - prevEdge invalid
13          else if((e.prevEdge.pv() > pvAtBot()) &&
14                  (e.prevEdge_{t,COMMIT} != e.prevEdge_{p,COMMIT}))
15              conflictDetected = true; // (Lost Update) - prevEdge
16          if(m == true) {
17            if(e.nextEdge_{p,COMMIT} != null &&
18               e.nextEdge_{p,COMMIT}.prevEdge_{p,COMMIT}.pv() > pvAtBot() &&
19               !((e.nextEdge_{p,COMMIT}, prev, {true|false})
20                  in changedEseqEdges_{T_i} &&
21                  e.nextEdge_{p,COMMIT} == e.nextEdge_{t,COMMIT}))
22              conflictDetected = true;
23          }
24          break;
25        case "next":
26          if((e.nextEdge_{t,COMMIT} not in edge_{p,COMMIT}) &&
27             (e.nextEdge_{t,COMMIT} not in ran addedEdges_{T_i}))
28            conflictDetected = true; // (Phantom) - nextEdge invalid
29          else if((e.nextEdge.pv() > pvAtBot()) &&
30                  (e.nextEdge_{t,COMMIT} != e.nextEdge_{p,COMMIT}))
31              conflictDetected = true; // (Lost Update) - nextEdge
32          if(m == true) {
33            if(e.prevEdge_{p,COMMIT} != null &&
34               e.prevEdge_{p,COMMIT}.nextEdge_{p,COMMIT}.pv() > pvAtBot() &&
35               !((e.prevEdge_{p,COMMIT}, next, {true|false})
36                  in changedEseqEdges_{T_i} &&
37                  e.prevEdge_{p,COMMIT} == e.prevEdge_{t,COMMIT}))
38              conflictDetected = true;
39          }
40          break;
41      }
42      if(conflictDetected) {
43        break;
44      }
45    }
46  }
47 }
```

Listing 4.5: Konfliktüberprüfung für $changedEseqEdges_{T_i}$

Konflikte durch Änderungen an Kanten

Konnte auch in Listing 4.5 kein Konflikt ermittelt werden, muss die Konfliktüberprüfung für jede in T_i geänderte Kante *e* durchgeführt werden (siehe Listing 4.6).

```
1  forall((e,i) in changedEdges_{T_i}) {
2    assert(e not in ran deletedEdges_{T_i});
3    boolean conflictDetected = false;
4    if((e not in ran addedEdges_{T_i}) && e not in edge_{p,COMMIT})
5      break; // (Phantom) - e no longer valid
6    else {
7      switch(i) {
8        case "alpha":
9          if((e.getAlpha()_{t,COMMIT} not in vertex_{p,COMMIT}) &&
10            (e.getAlpha()_{t,COMMIT} not in ran addedVertices_{T_i}))
11           conflictDetected = true; // (Phantom) - alpha
12         else if((e.getAlpha().pv() > pvAtBot()) &&
13                 (e.getAlpha()_{t,COMMIT} != e.getAlpha()_{p,COMMIT}))
14             conflictDetected = true; // (Lost Update) - alpha
15         break;
16       case "omega":
17         if((e.getOmega()_{t,COMMIT} not in vertex_{p,COMMIT}) &&
18            (e.getOmega()_{t,COMMIT} not in ran addedVertices_{T_i}))
19           conflictDetected = true; // (Phantom) - omega
20         else if((e.getOmega().pv() > pvAtBot()) &&
21                 (e.getOmega()_{t,COMMIT} != e.getOmega()_{p,COMMIT}))
22             conflictDetected = true;// (Lost Update) - omega
23         break;
24      }
25   }
26   if(conflictDetected) {
27     break;
28   }
29 }
```

Listing 4.6: Konfliktüberprüfung für $changedEdges_{T_i}$

Ein Konflikt wird für eine Kante *e* festgestellt, falls *e* beim *COMMIT* von T_i nicht mehr existiert (Zeile 4). Existiert *e* noch, muss je nach Kontext überprüft werden, ob der (temporäre) Start- bzw. Endknoten von *e* noch gültig ist (Zeilen 9-10 bzw. Zeilen 17-18).

Existiert der Start- bzw. Endknoten, tritt ein *Lost Update* auf, falls sich je nach Kontext die persistente Version des Start- (Zeile 12) bzw. Endknotens (Zeile 20) seit dem *BOT* von T_i geändert hat und sich jeweils die Werte der temporären und der persistenten Version zum *COMMIT*-Zeitpunkt von T_i unterscheiden (Zeile 13 bzw. 21).

Konflikte durch Änderungen der Inzidenzreihenfolge in Λ*seq(v)*

Sind in Listing 4.6 keine Konflikte feststellbar, wird die Konfliktüberprüfung für jede in T_i geänderte Inzidenzliste Λ*seq(v)* eines Knotens *v* fortgesetzt (siehe Listing 4.7).

```
forall(v in dom changedIncidences_{T_i}) {
  assert(v not in ran deletedVertices_{T_i});
  boolean conflictDetected = false;
  if(v not in ran addedVertices_{T_i} && v not in vertex_{p,COMMIT})
    break; // (Phantom) - v no longer valid
  else {
    if(v not in ran addedVertices_{T_i} && v.incidenceListVersion.pv() ≤ pvAtBot())
      continue; // incidence list of v hasn't changed since BOT
    forall(((e,d),p,m) in changedIncidences_{T_i}(v)) {
      assert(e.getNormalEdge() not in ran deletedEdges_{T_i});
      if(e.getNormalEdge() not in ran addedEdges_{T_i} &&
         e.getNormalEdge() not in edge_{p,COMMIT})
        break; // (Phantom) - e no longer valid
      else {
        switch(p) {
          case "prev":
            if(((e,d).prevIncidence_{t,COMMIT}.getNormalEdge()
                not in edge_{p,COMMIT}) && ((e,d).prevIncidence_{t,COMMIT}.
                getNormalEdge() not in ran addedEdges_{T_i}))
              conflictDetected = true; // (Phantom) - prevIncidence
            else if(((e,d).prevIncidence.pv() != pvAtBot() &&
              ((e,d).prevIncidence_{t,COMMIT} != (e,d).prevIncidence_{p,COMMIT})))
                conflictDetected = true; // (Lost Update) - prevIncidence
            if(m == true) {
              if((e,d).nextIncidence_{p,COMMIT} != null &&
                 (e,d).nextIncidence_{p,COMMIT}.prevIncidence_{p,COMMIT}.pv()
                 > pvAtBot() &&
                !(((e,d).nextIncidence_{p,COMMIT}, prev, {true|false})
                   in changedIncidences_{T_i}(v) &&
                   (e,d).nextIncidence_{p,COMMIT} == (e,d).nextIncidence_{t,COMMIT}))
              conflictDetected = true;
            }
            break;
          case "next":
            if(((e,d).nextIncidence_{t,COMMIT}.getNormalEdge()
               not in edge_{p,COMMIT}) && ((e,d).nextIncidence_{t,COMMIT}.
               gerNormalEdge() not in ran addedEdges_{T_i}))
              conflictDetected = true; // (Phantom) - nextIncidence
            else if(((e,d).nextIncidence.pv() != pvAtBot() &&
             ((e,d).nextIncidence_{t,COMMIT} != (e,d).nextIncidence_{p,COMMIT})))
                conflictDetected = true; // (Lost Update) - nextIncidence
            if(m == true) {
              if((e,d).prevIncidence_{p,COMMIT} != null &&
                 (e,d).prevIncidence_{p,COMMIT}.nextIncidence_{p,COMMIT}.pv()
                 > pvAtBot() &&
                !(((e,d).prevIncidence_{p,COMMIT}, next, {true|false})
                   in changedIncidences_{T_i}(v) &&
                   (e,d).prevIncidence_{p,COMMIT} == (e,d).prevIncidence_{t,COMMIT}))
              conflictDetected = true;
            }
            break;
```

```
52          }
53        }
54      }
55    }
56    if(conflictDetected) {
57      break;
58    }
59  }
```

Listing 4.7: Konfliktüberprüfung für $changedIncidences_{T_i}$

Zunächst muss ermittelt werden, ob der aktuelle Knoten *v* ein *Phantom* darstellt (Zeile 4). Ist *v* noch gültig, ist die Konfliktüberprüfung für Λ*seq(v)* nur erforderlich, falls sich *v.incidenceListVersion* seit dem *BOT* von T_i geändert hat (Zeilen 7-8).

Hat sich *incidenceListVersion* geändert, muss für jede Inzidenz in Λ*seq(v)* zuerst überprüft werden, ob die zugehörige Kante *e* beim *COMMIT* von T_i noch existiert (Zeilen 11-12). Ist *e* noch gültig, erfolgt die weitere Konfliktüberprüfung analog zu *Vseq* und *Eseq*.

Konflikte durch Änderungen an Attributen

Den letzten Schritt der Konfliktüberprüfung zeigt Listing 4.8. Hier werden auftretende Konflikte für die in T_i geänderten Attribute ermittelt.

```
1  forall(i in dom changedAttributes_{T_i}) {
2    if(i instanceof Vertex) {
3      assert(i not in ran deletedVertices_{T_i});
4      if(i in ran addedVertices_{T_i}) {
5        continue;
6      }
7      if(i not in vertex_{p,COMMIT}) {
8        // (Phantom) - i no longer valid
9        break;
10     }
11   }
12   if(i instanceof Edge) {
13     assert(i not in ran deletedEdges_{T_i});
14     if(i in ran addedEdges_{T_i}) {
15       continue;
16     }
17     if(i.getNormalEdge() not in edge_{p,COMMIT}) {
18       // (Phantom) - i no longer valid
19       break;
20     }
21   }
```

```
22    boolean conflictDetected = false;
23    forall(a in changedAttributes_{T_i}(i)) {
24      if((a.pv() > pvAtBot()) && !(a_{t,COMMIT}.equals(a_{p,COMMIT}))) {
25        conflictDetected = true; // (Lost Update) - a changed since BOT
26        break;
27      }
28    }
29    if(conflictDetected) { break;}
30  }
```

Listing 4.8: Konfliktüberprüfung für $changedAttributes_{T_i}$

Zunächst muss geprüft werden, ob es sich bei der aktuellen Instanz *i* um einen Knoten (Zeile 2) oder eine Kante (Zeile 12) handelt. Dabei muss jeweils überprüft werden, ob *i* beim *COMMIT* von T_i noch gültig ist (Zeile 7 bzw. Zeile 17). Da ein Graph nicht gelöscht werden kann, wird der Fall, dass *i* einer Grapheninstanz entspricht, nicht berücksichtigt.

Ist das jeweilige *i* noch gültig, wird für jedes geänderte Attribut *a* von *i* überprüft, ob sich die persistente Version von *a* seit dem *BOT* von T_i geändert hat und sich die Werte der temporären Version von *a* für T_i und der persistenten Version von *a* zum *COMMIT*-Zeitpunkt von T_i unterscheiden (Zeile 23). Ist dies der Fall, wird ein *Lost Update* für *a* festgestellt.

Sowohl bei den primitiven (*int*, *long*, *double*, *boolean* und *Enum*) als auch bei den komplexen (*String*, *List*, *Set*, *Map* und *Record*) Datentypen werden (aus Einfachheitsgründen) die gleichen Mechanismen zur Erkennung von Konflikten angewandt. Um die Gleichheit der Attributwerte zu überprüfen, wird die Java-Methode *equals* verwendet, da bei den Vergleichsoperatoren == und *!*= aufgrund des *Aliasings* lediglich eine Überprüfung auf die Gleichheit bzw. Ungleichheit der Referenzen stattfindet [Krü07].

4.2.3.10 Vorwegnahme der Validierungsphase

Da ein Neustart von Transaktionen zur Konfliktbehebung im Graphenlabor nicht umsetzbar ist (siehe Abschnitt 4.1.2.3, Seite 59), kann es unter Umständen sinnvoll sein, das Auftreten von Konfliktsituationen schon vor dem *COMMIT* einer Transaktion T_i zu überprüfen. So kann die Konfliktüberprüfung für T_i bereits in dessen Lesephase durchgeführt werden. Wird dabei ein Konflikt festgestellt, können schon vor dem *COMMIT* von T_i geeignete Maßnahmen getroffen werden (siehe Abschnitt 4.2.3.11).

Dieses Vorgehen kann jedoch nicht mehr als optimistische Synchronisation verstanden werden, da die Konflikterkennung schon während der Lesephase stattfindet. Auch wiederspricht sie der in Abschnitt 4.2.3.5 (Seite 90) getätigten Annahme, dass Transaktionen während ihrer Lesephase keinerlei Änderungen von parallel ablaufenden Transaktionen sehen können.

4.2.3.11 Vorgehen nach dem Auftreten eines Konflikts

Werden nach dem *COMMIT* einer Transaktion T_i keine Konflikte festgestellt, kann T_i in die *Schreibphase* übergehen. Tritt mindestens ein Konflikt auf, sollte T_i jedoch *nicht* gemäß der *Atomaritäts*-Eigenschaft des *ACID*-Paradigmas (siehe Abschnitt 3.1, Seite 27) automatisch ein *ABORT* und somit ein lokales Undo durchführen. Stattdessen sollte dem Anwender von JGraLab die Entscheidung überlassen werden, ob T_i ein *ABORT* durchführen soll, die Transaktion auf einen definierten Sicherungspunkt zurückgesetzt wird (siehe Abschnitt 4.3.3) oder weitere (konfliktauflösende) Operationen im Kontext von T_i durchgeführt werden.

4.2.3.12 Die Schreibphase

Gelangt eine Transaktion T_i schließlich in die Schreibphase, werden neue persistente Versionen der durch T_i geänderten Datenobjekte im Versionen-Pool erzeugt. Die Schreibphase muss atomar und synchronisiert ausgeführt werden. So darf sich zu einem Zeitpunkt höchstens eine Transaktion in der Schreibphase befinden. Zudem muss sichergestellt werden, dass sich während der Schreibphase von T_i keine weiteren Transaktionen T_j gleichzeitig in der Validierungsphase befinden. Auch darf keine weitere Transaktion T_j existieren, die parallel ihre *BOT*-Operation durchführt.

Für die Umsetzung der Schreibphase werden im Folgenden zwei alternative Varianten erläutert.

Erneutes Ausführen aller Schreiboperationen

In der ersten Variante werden alle von T_i durchgeführten Schreiboperationen in identischer Reihenfolge auf den beim *COMMIT* von T_i gültigen persistenten Versionen der durch T_i geänderten Datenobjekte erneut durchgeführt. Um dies umsetzen zu können, muss eine Protokollierung der in T_i ausgeführten Schreiboperationen erfolgen.

Um die Protokollierung von Schreiboperationen zu ermöglichen, müssen diese im JGraLab als Objekte behandelt werden können. Dies kann mit Hilfe des *Command-Patterns* realisiert werden [FFS04]. Abbildung 4.3 zeigt die Anwendung des Command-Patterns im JGraLab. Für jede in Abbildung 4.2 (Seite 64) gezeigte Schreiboperation wird eine eigene Klasse erstellt. Alle Klassen implementieren das Interface *Command*, welches die Methoden *execute()* und *undo()* definiert, mit denen eine Operation (erneut) ausgeführt bzw. (bei Bedarf) wieder rückgängig gemacht werden kann.

Durch die Protokollierung der Schreiboperationen fällt die Umsetzung der Schreibphase relativ simpel aus, da die dabei durchzuführenden Operationen bereits vorhanden sind und nicht neu implementiert werden müssen. Andererseits kann sich diese Variante vor allem bei einer großen Anzahl von Schreiboperationen als recht ineffizient darstellen. Wenn eine Transaktion beispielsweise 10 mal den Wert des gleichen *String*-Attributs ändert, wäre die Ausführung aller zugehörigen Schreiboperationen unnötig, da für diesen Fall nur die letzte Schreiboperation relevant ist. In vielen Fällen würden in der Schreibphase also mehr Schreiboperationen ausgeführt, als im Idealfall notwendig wäre.

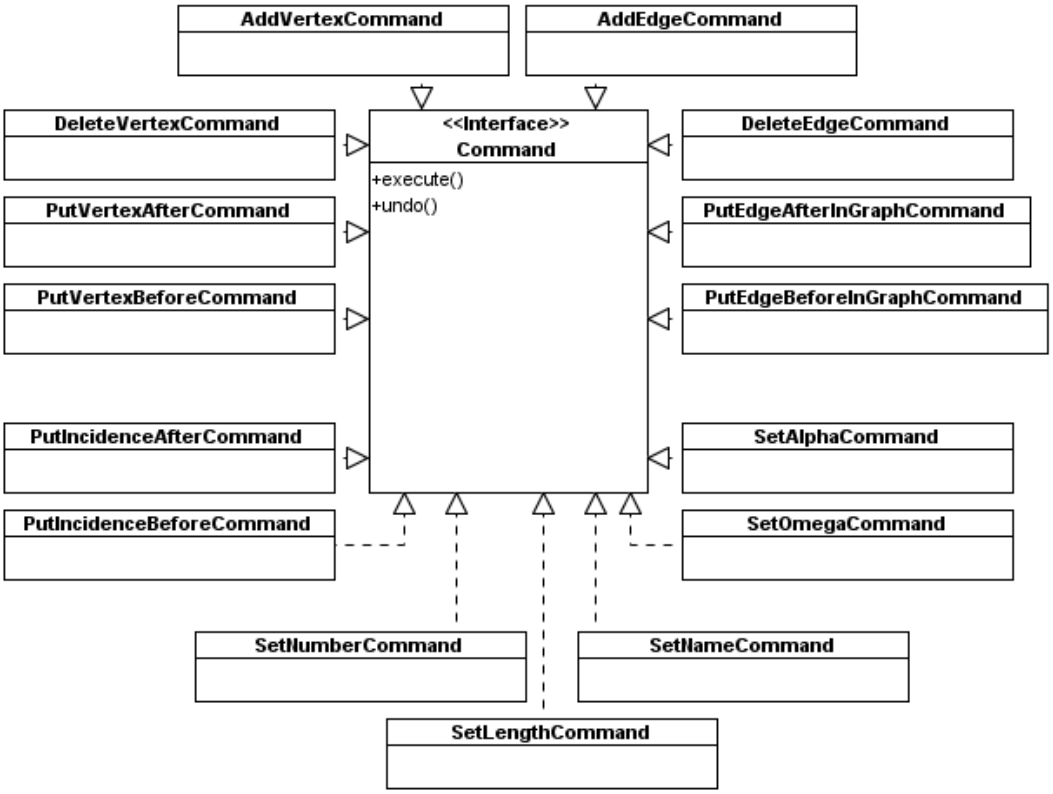


Abbildung 4.3: Die Anwendung des Command-Patterns im JGraLab

Erneutes Ausführen lediglich notwendiger Schreiboperationen

Die zweite Variante setzt die Schreibphase effizienter um. Hierbei wird komplett auf die Protokollierung von Schreiboperationen verzichtet. Stattdessen werden die in Abschnitt 4.2.3.8 (Seite 91) zur Konfliktüberprüfung und -erkennung mitgeführten Mengen berücksichtigt. Im Folgenden werden die einzelnen Schritte für eine Transaktion T_i erläutert. Die dabei auszuführenden Operationen werden jeweils auf den letzten persistenten Versionen der betroffenen Datenobjekte ausgeführt. Die Grapheninstanz wird mit dem Bezeichner *g* referenziert.

1. $\forall$ *v* $\in$ ran *deletedVertices*$_{T_i}$: *v* $\in$ $V_{p,COMMIT}$ wird die Methode *g.deleteVertex(v)* durchgeführt.

   Alle betroffenen Knoten *v* werden aus $V_{p,COMMIT}$ und $Vseq_{p,COMMIT}$ entfernt. Zudem werden alle entsprechenden Inzidenzlisten $\Lambda seq(v)_{p,COMMIT}$ gelöscht. Hierbei muss berücksichtigt werden, dass auch implizit jeweils alle zu *v* inzidenten Kanten aus $E_{p,COMMIT}$, $Eseq_{p,COMMIT}$ und allen Inzidenzlisten $\Lambda seq(v_i)_{p,COMMIT}$ entfernt werden, wobei v_i mindestens eine dieser Kanten als Inzidenz besitzt.

   Zusätzlich wird die (globale) Freispeicherliste der Knoten aktualisiert.

2. $\forall$ *e* $\in$ ran *deletedEdges*$_{T_i}$: *e* $\in$ $E_{p,COMMIT}$ wird die Methode *g.deleteEdge(e)* durchgeführt.

   Dadurch wird jede betroffene Kante *e* aus $E_{p,COMMIT}$, $Eseq_{p,COMMIT}$, $\Lambda seq(alpha)_{p,COMMIT}$ und $\Lambda seq(omega)_{p,COMMIT}$ gelöscht, wobei *alpha* = $e_{p,COMMIT}$.getAlpha() und *omega* = $e_{p,COMMIT}$.getOmega() gilt.

   Zusätzlich wird die (globale) Freispeicherliste der Kanten aktualisiert.

3. $\forall$ *v* $\in$ ran *addedVertices*$_{T_i}$ wird die Methode *g.addVertex(v)* durchgeführt.

   Jeder betroffene Knoten *v* wird in $V_{p,COMMIT}$ und $Vseq_{p,COMMIT}$ eingefügt.

   Da den zugehörigen Knoteninstanzen bereits (eindeutige) IDs zugewiesen wurden, wird die Freispeicherliste der Knoten nicht verändert.

4. $\forall$ *e* $\in$ ran *addedEdges*$_{T_i}$ wird die Methode *g.addEdge(e, alpha, omega)* durchgeführt.

   Die zu übergebenen Knoten *alpha* und *omega* werden mittels $e_{t,COMMIT}$.getAlpha() bzw. $e_{t,COMMIT}$.getOmega() ermittelt. Jede Kante *e* wird in $E_{p,COMMIT}$, $Eseq_{p,COMMIT}$, $\Lambda seq(alpha)_{p,COMMIT}$ und $\Lambda seq(omega)_{p,COMMIT}$ eingefügt.

   Die Freispeicherliste der Kanten wird ebenfalls nicht verändert.

5. $\forall$ *(v, p, m)* $\in$ *changedVseqVertices*$_{T_i}$ wird

   - für *p* = *prev*
     - g.putVertexAfter(v.$prevVertex_{t,COMMIT}$, v) für *m* = *true* oder
     - g.putVertexBefore(v, v.$prevVertex_{t,COMMIT}$) für *m* = *false* und

- für *p = next*
  - g.putVertexBefore(v.$nextVertex_{t,COMMIT}$, v) für *m = true* oder
  - g.putVertexAfter(v, v.$nextVertex_{t,COMMIT}$) für *m = false*

ausgeführt, falls *v.*$prevVertex_{t,COMMIT}$!= *v.*$prevVertex_{p,COMMIT}$ bzw. *v.*$nextVertex_{t,COMMIT}$!= *v.*$nextVertex_{p,COMMIT}$ gilt.

6. ∀ *(e, p, m)* ∈ *changedEseqEdges*$_{T_i}$ wird
   - für *p = prev*
     - g.putEdgeAfterInGraph(e.$prevEdge_{t,COMMIT}$, e) für *m = true* oder
     - g.putEdgeBeforeInGraph(e, e.$prevEdge_{t,COMMIT}$) für *m = false* und
   - für *p = next*
     - g.putEdgeBeforeInGraph(e.$nextEdge_{t,COMMIT}$, e) für *m = true* oder
     - g.putEdgeAfterInGraph(e, e.$nextEdge_{t,COMMIT}$) für *m = false*

   ausgeführt, falls *e.*$prevEdge_{t,COMMIT}$!= *e.*$prevEdge_{p,COMMIT}$ bzw. *e.*$nextEdge_{t,COMMIT}$!= *e.*$nextEdge_{p,COMMIT}$ gilt.

7. ∀ *(e, i)* ∈ *changedEdges*$_{T_i}$ wird
   - e.setAlpha(e.$getAlpha()_{t,COMMIT}$) für *i = alpha* oder
   - e.setOmega(e.$getOmega()_{t,COMMIT}$) für *i = omega*

   ausgeführt, falls *e.*$getAlpha()_{t,COMMIT}$!= *e.*$getAlpha()_{p,COMMIT}$ bzw. *e.*$getOmega()_{t,COMMIT}$!= *e.*$getOmega()_{p,COMMIT}$ gilt.

8. ∀ *v* ∈ dom *changedIncidences*$_{T_i}$ wird ∀ ((e, d), p, m) ∈ *changedIncidences*$_{T_i}$(v)
   - für *p = prev*
     - v.putIncidenceAfter((e, d).$prevIncidence_{t,COMMIT}$, (e, d)) für *m = true* oder

  - v.putIncidenceBefore((e, d), (e, d).$prevIncidence_{t,COMMIT}$) für *m = false* und

- für *p = next*

  - v.putIncidenceBefore((e, d).$nextIncidence_{t,COMMIT}$, (e, d)) für *m = true* oder

  - v.putIncidenceAfter((e, d), (e, d).$nextIncidence_{t,COMMIT}$) für *m = false*

  ausgeführt, falls *(e, d).*$prevIncidence_{t,COMMIT}$!= *(e, d).*$prevIncidence_{p,COMMIT}$ bzw. $(e, d).nextIncidence_{t,COMMIT}$!= *(e, d).*$nextIncidence_{p,COMMIT}$ gilt.

9. $\forall$ *i* $\in$ dom *changedAttributes*$_{T_i}$ wird $\forall$ *a* $\in$ *changedAttributes*$_{T_i}$*(i)* der Wert $a_{p,COMMIT}$ durch $a_{t,COMMIT}$ ersetzt, falls $a_{t,COMMIT}$!= $a_{p,COMMIT}$ gilt.

Aufgrund der höheren Effizienz bei der Ausführung wird im Folgenden die Anwendung der zweiten Variante für die Schreibphase angenommen.

4.2.3.13 Garbage Collection nach der Schreibphase

Nachdem die Schreibphase einer Transaktion T_i erfolgreich durchgeführt wurde, können für alle durch T_i geänderten Datenobjekte *x* die zugehörigen temporären Versionen x_t aus dem Versionen-Pool entfernt werden. Für jedes *x* können zudem alle älteren persistenten Versionen x_p aus dem Versionen-Pool gelöscht werden, sofern diese nicht beim *BOT* mindestens einer noch aktiven Transaktion T_j gültig waren.

4.3 Die Umsetzung der Recovery-Komponente im JGraLab

Dieser Abschnitt behandelt die Umsetzung der *Recovery*-Komponente im JGraLab. Zunächst wird analysiert, welche der in Abschnitt 3.5.1 (Seite 37) erwähnten Fehlerklassen im Kontext des Graphenlabors relevant sind (siehe Abschnitt 4.3.1). Anschließend wird auf die Durchführung des lokalen Undos einer Transaktion eingegangen (siehe Abschnitt 4.3.2). Abschließend wird erläutert, wie das Konzept der Sicherungspunkte im JGraLab umgesetzt wird (siehe Abschnitt 4.3.3).

4.3.1 Relevante Fehlerklassen

Im Kontext des Graphenlabors sind nur die *lokalen Fehler* relevant. *Fehler mit Hauptspeicherverlust* werden nicht berücksichtigt, da im Graphenlabor keine Einlagerung und Auslagerung von Datenobjekten zwischen Hauptspeicher und Hintergrundspeicher stattfindet. Alle persistenten Änderungen an Graphen(elementen) durch den *COMMIT* einer Transaktion werden nur im Hauptspeicher gespeichert. Die davon getrennte Speicherung des *gesamten* Graphen in einer *.TG*-Datei muss im Graphenlabor explizit vom Anwender angestossen werden. Bei einem Fehler mit Hauptspeicherverlust ist die Wiederherstellung des Hauptspeicherzustands vor dem Fehlerauftritt nicht vorgesehen. *Fehler mit Hintergrundspeicherverlust* sind im Kontext des Graphenlabors ebenfalls irrelevant.

4.3.2 Durchführung des lokalen Undos im JGraLab

Aufgrund der Verwaltung von persistenten und temporären Versionen von Datenobjekten in einem Versionen-Pool fällt die Durchführung des lokalen Undos für eine Transaktion T_i relativ simpel aus. Führt T_i die *ABORT*-Operation aus, muss zunächst dafür gesorgt werden, dass im Versionen-Pool alle durch T_i erzeugten temporären Versionen x_t von geänderten Datenobjekten *x* entfernt werden. Da T_i aufgrund des in Abschnitt 4.2 (Seite 66) erläuterten Synchronisationsverfahrens während ihrer Lesephase keine persistenten Versionen von Datenobjekten erzeugen kann, ist keinerlei Undo von Änderungen an Datenobjekten erforderlich.

Zusätzlich muss sichergestellt werden, dass alle durch T_i von den Freispeicherlisten angeforderten Knoten- und Kanten-IDs zur erneuten Vergabe freigegeben werden. Hierzu müssen die Mengen $addedVertices_{T_i}$ und $addedEdges_{T_i}$ betrachtet werden. Der Zugriff auf die jeweilige Freispeicherliste muss in diesem Kontext atomar und synchronisert erfolgen. Nachdem die Aktualisierung der Freispeicherlisten abgeschlossen ist, können anschließend alle durch T_i zur Konflikterkennung angelegten Mengen (siehe Abschnitt 4.2.3.8, Seite 91) gelöscht werden. T_i ist nach ihrem *ABORT* nicht mehr gültig und wird daher (unwiderruflich) aus dem System entfernt.

4.3.3 Das Konzept der Sicherungspunkte im JGraLab

Im Folgenden wird das Anlegen von Sicherungspunkten (siehe Abschnitt 4.3.3.1) innerhalb einer Transaktion T_i und das Zurücksetzen von T_i auf einen definierten Sicherungspunkt (siehe Abschnitt 4.3.3.2) erläutert.

4.3.3.1 Das Anlegen von Sicherungspunkten

Das Anlegen eines Sicherungspunkts erfolgt für jede Transaktion T_i mit der Operation *dsp(m)* (siehe Abschnitt 4.1.2.1, Seite 58), wobei *m* eine eindeutige und gültige *ID* für den anzulegenden Sicherungspunkt darstellt.

Ein in T_i angelegter Sicherungspunkt *s* muss sich den aktuellen Wert des (temporären) Versionenzählers T_i.*temporaryVersionCounter* in einem Feld *versionAtSavepoint* merken. So wird vermerkt, dass im Kontext vom Sicherungspunkt *s* für alle Datenobjekte *x* die zugehörige temporäre Version x_t mit der höchsten Versionsnummer *n* gültig ist, für die *n* ≤ *s.versionAtSavepoint* gilt. Existiert für ein Datenobjekt *x* zum Anlagezeitpunkt des Sicherungspunkts *s* (noch) keine zugehörige temporäre Version x_t, ist im Kontext von *s* die persistente Version x_p mit der höchsten Versionsnummer *n* gültig, für die *n* ≤ T_i.*persistentVersionAtBot* gilt.

Zusätzlich muss der Sicherungspunkt *s* Kopien der von T_i mitgeführten Mengen zur Konfliktüberprüfung (siehe Abschnitt 4.2.3.8, Seite 91) speichern.

Jede Transaktion T_i hält in dem Feld *latestDefinedSavepoint* eine Referenz auf den zuletzt angelegten Sicherungspunkt. Zu Beginn von T_i trägt *latestDefinedSavepoint* den Wert *null*. Legt T_i einen Sicherungspunkt *s* an, wird für *latestDefinedSavepoint* eine Referenz auf *s* gesetzt. Gilt *latestDefinedSavepoint* != *null*, muss bei jeder Änderung eines Datenobjekts *x* überprüft werden, ob das Erzeugen einer neuen temporären Version x_t erforderlich ist, um die Gültigkeit des in *latestDefinedSavepoint* referenzierten Sicherungspunkts zu garantieren. Für jedes Datenobjekt *x* muss für den Fall *latestDefinedSavepoint* != *null* demnach überprüft werden, ob für die Versionsnummer *n* der zur Zeit für T_i gültigen temporären Version x_t die Bedingung *n* ≤ *latestDefinedSavepoint.versionAtSavepoint* gilt. Ist dies der Fall oder existiert keine gültige temporäre Version x_t für T_i, muss für *x* eine neue temporäre Version erzeugt werden.

Die beschriebene Vorgehensweise beim Anlegen von Sicherungspunkten wird mit Hilfe des Beispielszenarios in Tabelle 4.34 für eine Transaktion T_i verdeutlicht. Dabei wird *temporaryVersionCounter* mit *tvc* und *latestDefinedSavepoint* mit *lds* abgekürzt. Die Bedeutung der Spalten *lrs* und $savepointList_{T_i}$ und die Angaben in Klammern im 15. bis 17. Schritt werden im nachfolgenden Abschnitt 4.3.3.2 erläutert.

Für das Szenario wird vereinfachend angenommen, dass lediglich drei Datenobjekte *x*, *y* und *z* existieren. Schreiboperationen auf diesen Datenobjekten werden mit *w(x)*, *w(y)* bzw. *w(z)*, Leseoperationen mit *r(x)*, *r(y)* bzw. *r(z)* notiert.

Zum *BOT*-Zeitpunkt von T_i werden tvc mit *0* und *lds* mit *null* initialisiert. Im 5. Schritt legt T_i einen Sicherungspunkt mit der ID *sp1* an. *sp1* speichert für *versionAtSavepoint* den aktuellen Wert von *tvc* (*2*). *lds* erhält eine Referenz auf den Sicherungspunkt *sp1*.

Schritt	T_i	tvc	lds	lrs	$savepointList_{T_i}$
1.	BOT	0	null	null	<>
2.	w(x)	1	null	null	<>
3.	w(y)	2	null	null	<>
4.	w(y)	2	null	null	<>
5.	dsp(sp1)	2	sp1	null	<sp1>
6.	w(z)	3	sp1	null	<sp1>
7.	w(y)	4	sp1	null	<sp1>
8.	w(x)	5	sp1	null	<sp1>
9.	dsp(sp2)	5	sp2	null	<sp1, sp2>
10.	w(z)	6	sp2	null	<sp1, sp2>
11.	w(z)	6	sp2	null	<sp1, sp2>
12.	w(x)	7	sp2	null	<sp1, sp2>
13.	dsp(sp3)	7	sp3	null	<sp1, sp2, sp3>
14.	w(x)	8	sp3	null	<sp1, sp2, sp3>
15.	rsp(sp2)	5 (8)	sp2	sp2	<sp1, sp2, sp3>
16.	r(x)	5 (8)	sp2	sp2	<sp1, sp2, sp3>
17.	w(x)	5 (8)	sp2	null (sp2)	<sp1, sp2> (<sp1, sp2, sp3>)
...	...	...	...	...	

Tabelle 4.34: Szenario zur Umsetzung von Sicherungspunkten

Nach dem Anlegen von *sp1* werden im 6., 7. und 8. Schritt neue temporäre Versionen der Datenobjekte *z*, *y* bzw. *x* angelegt. Für *z* ist dies notwendig, da noch keine temporäre Version z_t für T_i existiert. Für *y* ist es erforderlich, da für die Versionsnummer *n* der zur Zeit gültigen temporären Version y_t gilt, dass n (2) ≤ *lds.versionAtSavepoint* (2). Analoges gilt für *x*, da die Versionsnummer *n* von x_t (1) ≤ *lds.versionAtSavepoint* (2) ist.

Im 9. und 13. Schritt werden weitere Sicherungspunkte *sp2* und *sp3* angelegt, wobei jeweils der Wert von *lds* entsprechend aktualisiert wird.

4.3.3.2 Das Zurücksetzen einer Transaktion auf einen Sicherungspunkt

Das Zurücksetzen einer Transaktion T_i auf einen definierten Sicherungspunkt erfolgt mit der Operation *rsp(m)* (siehe Abschnitt 4.1.2.1, Seite 58), wobei *m* eine gültige *ID* eines in T_i angelegten Sicherungspunkts darstellt.

Im Folgenden werden zwei Vorgehensweisen beim Zurücksetzen einer Transaktion T_i auf einen Sicherungspunkt erläutert. Dabei wird angenommen, dass T_i alle angelegten Sicherungspunkte in einer Liste $savepointList_{T_i}$: iseq SP speichert, wobei *SP* hier das Universum aller Sicherungspunkte darstellt.

Entfernen von nachfolgenden Sicherungspunkten

In der ersten Variante werden beim Zurücksetzen von T_i auf einen Sicherungspunkt *s* alle Sicherungspunkte *t* für T_i aus $savepointList_{T_i}$ entfernt, die nach der Initialisierung von *s* definiert worden sind, sobald eine weitere Änderung an einem beliebigen Datenobjekt durch T_i durchgeführt wird.

T_i*.temporaryVersionCounter* wird dabei auf den Wert von *s.versionAtSavepoint* zurückgesetzt. Für alle in T_i geänderten Datenobjekte *x* können alle temporären Versionen x_t entfernt werden, deren Versionsnummer *n* > *s.versionAtSavepoint* ist, sobald eine weitere Änderung an einem beliebigen Datenobjekt durch T_i durchgeführt wird. Zudem erhält *lds* eine Referenz auf den Sicherungspunkt *s*.

Um nach dem Zurücksetzen von T_i auf einen Sicherungspunkt *s* (ohne unmittelbar folgender Änderung an einem beliebigen Datenobjekt) den korrekten lesenden Zugriff auf die Datenobjekte zu garantieren, wird ein weiteres Feld *latestRestoredSavepoint* (*lrs*) in T_i benötigt. *lrs* erhält zu Beginn von T_i den Wert *null*. Nach Ausführung von *rsp(s)* wird in *lrs* eine Referenz auf den Sicherungspunkt *s* gesetzt. So wird vermerkt, dass zur Zeit für alle Datenobjekte *x* die zugehörige temporäre Version x_t mit der höchsten Versionsnummer *n* gültig ist, für die *n* ≤ *lrs.versionAtSavepoint* gilt. Sobald nach dem Ausführen von *rsp(s)* mindestens eine weitere Änderung an einem beliebigen Datenobjekt durchgeführt wurde und anschließend alle ungültigen temporären Versionen für T_i aus dem Versionen-Pool entfernt worden sind, ist die im Feld *lrs* gehaltene Referenz auf *s* nicht mehr notwendig. *lrs* wird auf den Wert *null* zurückgesetzt.

In Tabelle 4.34 wird T_i im 15. Schritt auf den Sicherungspunkt *sp2* zurückgesetzt. Dabei werden *tvc* auf den Wert *sp2.versionAtSavepoint* (5), *lds* und *lrs* auf den Wert *sp2* gesetzt. Im 16. Schritt wird eine Leseoperation auf dem Datenobjekt *x* unter Berücksichtigung der in *lrs* gespeicherten Referenz durchgeführt. Durch die Schreiboperation auf dem Datenobjekt *x* im 17. Schritt wird der Sicherungspunkt *sp3* aus dem System entfernt ($savepointList_{T_i}$ = <sp1, sp2>) und *lrs* auf den Wert *null* zurückgesetzt.

Hin- und Herspringen zwischen Sicherungspunkten

In der zweiten Variante werden beim Zurücksetzen von T_i auf einen Sicherungspunkt *s keine* Sicherungspunkte aus $savepointList_{T_i}$ entfernt. Der aktuelle Wert von *temporaryVersionCounter* bleibt unverändert. Dafür muss sich die Transaktion T_i in dem Feld *lrs* (dauerhaft) merken, auf welchen Sicherungspunkt T_i zuletzt zurückgesetzt wurde.

Anschließend muss sichergestellt werden, dass T_i für alle Datenobjekte *x* lediglich auf die temporären Versionen x_t mit der höchsten Versionsnummer *n* zugreifen kann, für die *n* ≤ *lrs.versionAtSavepoint* gilt.

Während die erste Variante nur das *Undo* von Änderungen durch das Zurücksetzen einer Transaktion T_i auf einen Sicherungspunkt unterstützt, wird in der zweiten Variante auch das *Redo* von Änderungen innerhalb von T_i ermöglicht.

In Tabelle 4.34 bleiben (in Klammern) im 17. Schritt alle Sicherungspunkte erhalten, so dass $savepointList_{T_i}$ = <sp1, sp2, sp3> gilt. *tvc* bleibt unverändert (*8*), *lds* und *lrs* werden auf *sp2* gesetzt.

Da die erste Variante durch das Entfernen von (nachfolgenden) Sicherungspunkten und nicht mehr gültigen temporären Versionen von Datenobjekten weniger Speicherplatz beansprucht und eine intuitivere Umsetzung des Konzepts der Sicherungspunkte darstellt, wird im Folgenden von der Verwendung der ersten Variante ausgegangen.

Für beide Varianten gilt gleichermaßen, dass beim Zurücksetzen einer Transaktion T_i auf einen Sicherungspunkt *s* die Mengen zur Konfliktüberprüfung von T_i durch die in *s* gespeicherten aktualisiert werden müssen.

4.4 Überprüfung von Konsistenzbedingungen

Im Kontext des Graphenlabors ist die Überprüfung von Konsistenzbedingungen beim *COMMIT* einer Transaktion im Bezug auf das in der Ebene *M2* definierte Schema für TGraphen denkbar. Dabei wird die Einhaltung der Beziehungen zwischen den im Schema definierten Elementen bei der Erzeugung und dem Entfernen von Graphenelementen bereits durch die API von JGraLab gewährleistet. Allerdings werden die im Schema definierten Kardinalitäten (noch) nicht mitberücksichtigt. Um deren Erfüllung zu überprüfen, kann GReQL 2 (Graph Repository Query Language) verwendet werden [Mar06]. GReQL 2 ist eine an SQL angelehnte Sprache, mit der Anfragen an TGraphen formuliert werden können.

Die Definition weiterer Konsistenzbedingungen hängt von der Anwendungsumgebung ab, in der JGraLab verwendet wird. Da das Transaktionskonzept im Graphenlabor *anwendungsunabhängig* genutzt wird, sind solche Konsistenzbedingungen nicht prüfbar und somit in diesem Kontext irrelevant.

Im Rahmen dieser Diplomarbeit wird die Überprüfung von Konsistenzbedingungen nicht umgesetzt. Allerdings sollte in der Implementierung des Transaktionskonzepts eine *Hook-Schnittstelle* vorgesehen werden, die eine spätere Integration einer *Consistency-Check*-Komponente in das System erlaubt.

5 Objektorientierter Feinentwurf der Integration des Transaktionskonzepts im JGraLab

In Kapitel 4 wurden die formalen Grundlagen des Transaktionskonzepts erläutert. In diesem Kapitel erfolgt der objektorientierte Feinentwurf der Integration des Transaktionskonzepts in die aktuelle Version von JGraLab. Hierfür werden zunächst diejenigen Bereiche der aktuellen Implementierung identifiziert, welche durch die Einführung des Transaktionskonzepts gegebenenfalls verändert und/ oder erweitert werden müssen (siehe Abschnitt 5.1). Anschließend werden die Implementationsdetails zur Integration des Transaktionskonzepts im JGraLab erläutert (siehe Abschnitt 5.2).

5.1 Notwendige Anpassungen der aktuellen JGraLab-Implementation

In den nachfolgenden Abschnitten wird erläutert, inwiefern und an welchen Stellen die aktuelle Implementation von JGraLab für die Integration des Transaktionskonzepts angepasst und/ oder erweitert werden muss. Dabei werden allgemeine Aspekte (siehe Abschnitt 5.1.1), die Versionierung (siehe Abschnitt 5.1.2), die Konflikterkennung (siehe Abschnitt 5.1.3) und neu zu entwickelnde Aspekte (siehe Abschnitt 5.1.4) angesprochen. Die Erläuterungen zu der Umsetzung dieser Anpassungen und Erweiterungen erfolgt in Abschnitt 5.2.

5.1.1 Erweiterungen für die Nutzung des Transaktionskonzepts

Im Folgenden werden allgemeine Aspekte der Integration und der Nutzung des Transaktionskonzepts im JGraLab angesprochen.

5.1.1.1 Einführung neuer Java-Packages

Für die Umsetzung des Transaktionskonzepts ist die Einführung neuer Java-Packages (auch im generierten Code) hilfreich. Hiermit wird eine Trennung zwischen den Interfaces und den Implementationsklassen mit und ohne Transaktionskonzept vorgenommen.

5.1.1.2 Eigene Implementationsklassen für das Transaktionskonzept

Für das Transaktionskonzept ist die Umsetzung eigener Implementationsklassen für Graphen-, Knoten- und Kanteninstanzen erforderlich. Ein wichtiges Ziel ist dabei, möglichst viel von der aktuellen Implementierung im Paket *de.uni_koblenz.jgralab.impl* wiederzuverwenden.

5.1.1.3 Eigene GraphFactory-Implementierung

Wie bereits in Abschnitt 2.3 (Seite 21) erläutert, ist im JGraLab die *GraphFactory* für die Instanziierung von Graphen-, Knoten- und Kanteninstanzen zur Laufzeit verantwortlich. Die Zuordnung zwischen den Graphen-, Knoten- und Kanteninterfaces und den jeweiligen Implementationsklassen erfolgt in der konkreten *GraphFactory*-Implementationsklasse, welche im generierten Code erzeugt wird. Für das Beispielschema *MotorwayMapSchema* (siehe Abbildung 1.4, Seite 11) ist dies die Klasse *MotorwayMapSchemaFactory* aus dem Paket *de.uni_koblenz.motorwaymap.impl* (siehe Listing 2.1, Seite 22).

Für das Transaktionskonzept ist die Generierung einer eigenen oder die Anpassung der bestehenden *GraphFactory*-Implementationsklasse notwendig, um die Zuordnung zwischen den Graphen-, Knoten- und Kanteninterfaces und den für das Transaktionskonzept erforderlichen Implementationsklassen vornehmen zu können.

5.1.1.4 Laden und Erstellen von Graphen mit Transaktionsunterstützung

Der Anwender von JGraLab sollte entscheiden können, ob der lesende und schreibende Zugriff auf eine Grapheninstanz mit oder ohne Transaktionsunterstützung erfolgen soll. Diese Entscheidung muss beim Laden existierender Graphen und beim Erzeugen neuer Graphen erfolgen. Für das Beispielschema *MotorwayMapSchema* ist die Klasse *MotorwayMapSchema* (siehe Abbildung 2.4, Seite 20) für das Laden und Erzeugen von Grapheninstanzen *ohne* Transaktionsunterstützung verantwortlich. *MotorwayMapSchema* bietet dafür die Methoden *loadMotorwayMap* bzw. *createMotorwayMap* an. Im Hin-

blick auf das Transaktionskonzept muss die Klasse *MotorwayMapSchema* auch das Laden und Erzeugen von Grapheninstanzen *mit* Transaktionsunterstützung ermöglichen.

Dabei muss sichergestellt werden, dass die geladenen und neu erzeugten Grapheninstanzen mit Transaktionsunterstützung jeweils Referenzen auf Instanzen der neuen oder angepassten (für das Transaktionskonzept notwendigen) *GraphFactory*-Implementierung erhalten.

5.1.2 Anpassungen für die Umsetzung der Versionierung

JGraLab muss im Rahmen des Transaktionskonzepts *temporäre* und *persistente* Versionen der Datenobjekte verwalten können. Werden im JGraLab die in Abbildung 4.1 (Seite 63) und Abbildung 4.2 (Seite 64) abgebildeten Lese- bzw. Schreiboperationen durch eine Transaktion T_i während ihrer Lesephase ausgeführt, muss sichergestellt werden, dass innerhalb dieser Operationen nur Zugriffe auf die jeweils für T_i gültigen Versionen der Datenobjekte erfolgen. Im Allgemeinen müssen also *lesende* und *schreibende* Zugriffe auf Datenobjekte innerhalb der angesprochenen Methoden im Rahmen des Transaktionskonzepts angepasst werden. Dieser Aspekt wird im Folgenden mit Hilfe ausgewählter Beispiele für Methoden aus der Klasse *GraphImpl* und aus der generierten Klasse *MotorwayImpl* (siehe Abbildung 2.3, Seite 19) erläutert.

5.1.2.1 Methoden aus der Klasse GraphImpl

Listing 5.1 zeigt als Beispiel für eine Leseoperation die Implementierung der Methode *getVertex* der Klasse *GraphImpl*, welche die ID eines Knotens als Parameter erhält und die zugehörige Knoteninstanz aus dem Array *vertex* zurückliefert.

```
1 public Vertex getVertex(int vId) {
2   assert (vId >= 0 && vId <= vMax);
3   return vertex[vId];
4 }
```

Listing 5.1: Implementierung der Methode *getVertex*

In der Methode *getVertex* findet (unter anderem) ein lesender Zugriff auf das Array *vertex* statt, welches im JGraLab die Knotenmenge *V* repräsentiert. Für eine Transaktion T_i existiere bereits eine temporäre Version $vertex_t$ mit der Versionsnummer *n*. So muss innerhalb von *getVertex* bei der Ausführung durch T_i sichergestellt werden, dass bei dem lesenden Zugriff auf *vertex* der für T_i gültige Wert aus der temporären Version $vertex_t$ mit der Versionsnummer *n* geladen wird.

In Listing 5.2 wird als Beispiel für eine Schreiboperation ein Auszug aus der Implementierung der Methode *addVertex* der Klasse *GraphImpl* dargestellt, welche die als Parameter übergebene Knoteninstanz *newVertex* in den Graphen einfügt. Dabei ruft *addVertex* die private Methode *appendVertexToVSeq* auf (Zeile 16). Die ... stehen stellvertretend für Codeausschnitte, welche im aktuellen Kontext irrelevant sind und daher aus Übersichtsgründen im Folgenden weggelassen werden.

```
1  protected void addVertex(Vertex newVertex) {
2    VertexImpl v = (VertexImpl) newVertex;
3    int vId = v.getId();
4    ...
5    if (vId != 0) {
6      throw new GraphException("can not add a vertex with id != 0");
7    } else {
8      vId = freeVertexList.allocateIndex();
9      if (vId == 0) {
10       expandVertexArray(getExpandedVertexCount());
11       vId = freeVertexList.allocateIndex();
12     }
13     v.setId(vId);
14   }
15   ...
16   appendVertexToVSeq(v);
17 }
18
19 private final void appendVertexToVSeq(VertexImpl v) {
20   if (firstVertex == null) {
21     firstVertex = v;
22   }
23   if (lastVertex != null) {
24     lastVertex.setNextVertex(v);
25     v.setPrevVertex(lastVertex);
26   }
27   lastVertex = v;
28   vertex[v.getId()] = v;
29   ++vCount;
30 }
```

Listing 5.2: Ausschnitt aus der Implementierung der Methode *addVertex*

In den Zeilen 28 und 29 erfolgen beispielsweise (zunächst) lesende Zugriffe auf das Array *vertex* und das Feld *vCount*. Dabei muss sichergestellt werden, dass die für T_i aktuell gültigen (persistenten oder temporären) Versionen von *vertex* und *vCount* geladen werden. Zudem werden in den angesprochenen Zeilen schreibende Zugriffe auf *vertex* und *vCount* durchgeführt. Falls noch keine temporären Versionen $vertex_t$ und $vCount_t$ für T_i existieren, müssen (in der Lesephase) aufgrund der erfolgten Änderungen neue temporäre Versionen von *vertex* und *vCount* für T_i im Versionen-Pool erstellt werden. Die gleichen

Überlegungen gelten analog für die Zugriffe auf die Felder *firstVertex* und *lastVertex* in den Zeilen 20 - 27.

5.1.2.2 Zugriff auf die Freispeicherlisten

In Zeile 8 von Listing 5.2 erfolgt der Zugriff auf die Freispeicherliste der Knoten (*freeVertexList*). Im Rahmen des Transaktionskonzepts muss der Zugriff von parallel laufenden Transaktionen auf diese synchronisiert werden. Dabei muss beachtet werden, dass auch in der Methode *deleteVertex* der Klasse *GraphImpl* auf die Freispeicherliste der Knoten zugegriffen wird. Für die Freispeicherliste der Kanten gelten analog die gleichen Überlegungen für die Methoden *addEdge* und *deleteEdge* der Klasse *GraphImpl*.

5.1.2.3 Vergrößerung der Graphenkapazitäten zur Aufnahme von Graphenelementen

In Zeile 10 von Listing 5.2 wird für den Fall, dass die von der Freispeicherliste der Knoten zurückgelieferte ID den Wert 0 besitzt (Zeile 9), eine Vergrößerung der Graphenkapazitäten zur Aufnahme von Knoten vorgenommen (*expandVertexArray*). Im Kontext des Transaktionskonzepts muss dieser Vorgang für alle zu einem Zeitpunkt aktiven Transaktionen durchgeführt werden. So müssen beispielsweise für alle aktiven Transaktionen T_i die jeweils gültigen (temporären oder persistenten) Versionen des Arrays *vertex* gleichermaßen erweitert werden. Diese Überlegungen gelten analog für die Vergrößerung der Graphenkapazitäten zur Aufnahme von Kanten.

5.1.2.4 Methoden aus der generierten Klasse MotorwayImpl

In Listing 5.3 ist ein Ausschnitt aus der Implementierung der generierten Klasse *MotorwayImpl* abgebildet.

```
1  public class MotorwayImpl extends VertexImpl
2  implements test.Motorway, de.uni_koblenz.jgralab.Vertex {
3    protected double length;
4    protected java.lang.String name;
5    ...
6    public Object getAttribute(String attributeName)
7    throws NoSuchFieldException {
8      if (attributeName.equals("length")) return length;
9      if (attributeName.equals("name")) return name;
10     throw new NoSuchFieldException("Motorway doesn't contain an attribute"
11     + attributeName);
12   }
13
```

```
14
15    public void setAttribute(String attributeName, Object data)
16    throws NoSuchFieldException {
17      if (attributeName.equals("length")) {
18        setLength((Double) data); return;
19      }
20      if (attributeName.equals("name")) {
21        setName((java.lang.String) data); return;
22      }
23      throw new NoSuchFieldException("Motorway doesn't contain an attribute"
24      + attributeName);
25    }
26
27    public double getLength() {
28      return length;
29    }
30
31    public java.lang.String getName() {
32      return name;
33    }
34
35    public void setLength(double length) {
36      this.length = length;
37      ...
38    }
39
40    public void setName(java.lang.String name) {
41      this.name = name;
42      ...
43    }
44    ...
45    public void readAttributeValues(GraphIO io)
46    throws GraphIOException {
47      length = io.matchDouble();
48      setLength(length);
49      name = io.matchUtfString();
50      setName(name);
51    }
52
53    public void writeAttributeValues(GraphIO io)
54    throws GraphIOException, IOException {
55      io.space();
56      io.writeDouble(length);
57      io.writeUtfString(name);
58    }
59  }
```

Listing 5.3: Ausschnitt aus der Implementierung der Klasse *MotorwayImpl*

Im Kontext der Versionierung sind die Methoden zum Auslesen und Setzen von Attributwerten relevant. Dabei existieren in der Klasse *MotorwayImpl* neben den herkömmlichen getter- und setter-Methoden *getLength* und *getName* (Zeile 27 bzw. 31) bzw. *setLength* und *setName* (Zeile 35 bzw. 40) auch die generischen Methoden *getAttribute* und *setAttribute* (Zeile 6 bzw. 15). Letztere können universell für alle in *MotorwayImpl* auftretenden Attribute verwendet werden. Für alle genannten Methoden und die darin auftretenden lesenden und schreibenden Zugriffe auf die Attribute muss der Versionierungsaspekt berücksichtigt werden.

Die Methoden *readAttributeValues* (Zeile 45) und *writeAttributeValues* (Zeile 53) werden beim Laden eines Graphen aus einer .TG-Datei bzw. bei der persistenten Speicherung eines Graphen in einer *.TG*-Datei benötigt. Diese Aspekte werden in Abschnitt 5.1.2.7 und Abschnitt 5.1.2.8 erläutert.

5.1.2.5 Besonderheit beim Löschen von Knoten

Eine Besonderheit muss beim Löschen von Knoten im JGraLab beachtet werden. Diese wird mit Hilfe von Listing 5.4 erläutert, welche einen entsprechenden Ausschnitt aus der Implementierung der Klasse *GraphImpl* zeigt.

```
1  public abstract class GraphImpl extends AttributedElementImpl
2  implements Graph {
3    private List<VertexImpl> deleteVertexList;
4    ...
5    public void deleteVertex(Vertex v) {
6      deleteVertexList.add((VertexImpl) v);
7      internalDeleteVertex();
8      vertexListModified();
9    }
10
11   private void internalDeleteVertex() {
12     while (!deleteVertexList.isEmpty()) {
13       VertexImpl v = deleteVertexList.remove(0);
14       ... // make all necessary operations to delete the current vertex
15     }
16   }
17 }
```

Listing 5.4: Ausschnitt aus der Implementierung der Klasse *GraphImpl*

Innerhalb der Methode *deleteVertex* wird in Zeile 6 der zu löschende Knoten in die innerhalb der Klasse *GraphImpl* global gültigen Liste *deleteVertexList* (Zeile 3) eingefügt. In Zeile 7 wird die klasseninterne Methode *internalDeleteVertex* (Zeile 11) aufgerufen. In dieser werden alle in *deleteVertexList* befindlichen Knoteninstanzen durchlaufen. Für jede Knoteninstanz werden die entsprechenden Löschoperationen durchgeführt.

Im Kontext des Transaktionskonzepts muss für jede Transaktion T_i eine eigene Version von *deleteVertexList* verwaltet werden, um das Löschen von Knoten innerhalb einer Transaktion T_i korrekt durchführen zu können.

5.1.2.6 Anpassung der vom Transaktionskonzept unabhängigen Versionierung

In der aktuellen Implementierung vom JGraLab werden bereits (vom Transaktionskonzept unabhängige) Versionenzähler eingesetzt, um Änderungen an bestimmten Datenobjekten nachvollziehbar zu machen. So besitzt die Klasse *GraphImpl* die *long*-Felder:

- **graphVersion**, welche die Versionsnummer des Graphen *G* speichert,
- **vertexListVersion**, welche die Versionsnummer von *Vseq* repräsentiert und
- **edgeListVersion**, welche die Versionsnummer von *Eseq* speichert.

Der Wert von *graphVersion* wird bei jeder Änderung, die am Graphen oder den darin enthaltenen Graphenelementen durchgeführt wird, um 1 erhöht. *vertexListVersion* und *edgeListVersion* werden jeweils bei jeder an *Vseq* bzw. *Eseq* durchgeführten Änderung um den Wert 1 inkrementiert.

Die Klasse *VertexImpl* besitzt das Feld **incidenceListVersion** vom Typ *long*, welches die Versionsnummer der Inzidenzliste der jeweiligen Knoteninstanz speichert und bei jeder Änderung an der entsprechenden Inzidenzliste um den Wert 1 erhöht wird.

Im Rahmen des Transaktionskonzepts sollte *graphVersion* nur bei persistenten Änderungen in der Schreibphase einer Transaktion T_i erhöht werden und bei temporären Änderungen unverändert bleiben.

vertexListVersion, *edgeListVersion* und *incidenceListVersion* werden im Kontext der (generischen) Klassen *VertexIterable*, *EdgeIterable* bzw. *IncidenceIterable* benötigt, die das Java-Interface *Iterable* implementieren und mit denen *Vseq*, *Eseq* bzw. Λ*seq(v)* einer Knoteninstanz *v* durchlaufen werden können.

Listing 5.5 zeigt beispielhaft die Methode *vertices* der Klasse *GraphImpl*, die eine Instanz der Klasse *VertexIterable* zurückliefert.

```
1 public Iterable<Vertex> vertices() {
2   return new VertexIterable<Vertex>(this);
3 }
```

Listing 5.5: Die Implementierung der Methode *vertices*

In Listing 5.6 wird ein Auszug aus der Implementierung der Klasse *VertexIterable* dargestellt.

```
1  public class VertexIterable<V extends Vertex>
2  implements Iterable<V> {
3    ...
4    public VertexIterable(Graph g) {
5      iter = new VertexIterator(g);
6    }
7
8    public Iterator<V> iterator() {
9      return iter;
10   }
11
12   class VertexIterator implements Iterator<V> {
13     ...
14     protected Graph graph = null;
15     protected long vertexListVersion;
16
17     VertexIterator(Graph g) {
18       graph = g;
19       vertexListVersion = g.getVertexListVersion();
20     }
21
22     public V next() {
23       if (graph.isVertexListModified(vertexListVersion))
24         throw new ConcurrentModificationException("The vertex list
25         of the graph has been modified - the iterator is no longer valid");
26       ...
27       return current;
28     }
29   }
30 }
```

Listing 5.6: Die Klasse *VertexIterable*

In Zeile 5 wird eine Instanz der internen Klasse *VertexIterator* erzeugt. Diese speichert in den Zeilen 18 und 19 in ihrem Konstruktor eine Referenz auf den Graphen *g* bzw. in dem Feld *vertexListVersion* die zum Zeitpunkt der Initialisierung aktuelle Versionsnummer von *Vseq* in *g*. Wird die Methode *next* der Klasse *VertexIterator* aufgerufen, um den jeweils nächsten Knoten in *Vseq* zu ermitteln, wird in Zeile 23 zunächst überprüft, ob die zu Beginn ermittelte Versionsnummer für Vseq in *g* immer noch aktuell ist. Ist dies nicht der Fall, wurde zwischenzeitlich eine Änderung an *Vseq* vorgenommen. Dadurch wird der Iterator als ungültig eingestuft und kann im weiteren Verlauf nicht weiterverwendet werden. Eine entsprechende Exception wird geworfen (Zeile 24).

Um diesen Kontrollmechanismus bei der Verwendung der Iterator-Klassen auch im Kontext einzelner Transaktionen umsetzen zu können, muss jede Transaktion T_i eigene temporäre Versionen der Felder *vertexListVersion*, *edgeListVersion* und *incidenceListVersion* verwalten. Die Versionierung der angesprochenen Felder wurde bereits in Abschnitt 4.2.3.3 (Seite 87) erläutert.

5.1.2.7 Laden von TGraphen aus einer .TG-Datei

Beim Laden eines TGraphen muss bei der Verwendung des Transaktionskonzepts sichergestellt werden, dass bereits bei der Initialisierung des Graphen und dessen Graphenelementen die Versionierung entsprechend umgesetzt wird. Dies bedeutet unter anderem, dass für alle relevanten Datenobjekte *x* zu Beginn persistente Versionen x_p im Versionen-Pool angelegt und die Freispeicherlisten korrekt initialisiert werden müssen.

5.1.2.8 Persistente Speicherung von TGraphen in einer .TG-Datei

Bei der persistenten Speicherung einer Grapheninstanz in einer .TG-Datei im Kontext einer Transaktion T_i werden (lediglich) die zum jeweiligen Zeitpunkt für T_i gültigen (temporären) Versionen der Datenobjekte berücksichtigt. So kann jederzeit der in T_i gültige Zustand der Grapheninstanz gesichert werden.

Möchte man die zu einem Zeitpunkt gültigen *persistenten* Versionen der Datenobjekte einer Grapheninstanz in einer .TG-Datei abspeichern, kann der Speichervorgang beispielsweise im Rahmen einer *Read-Only*-Transaktion T_j gestartet werden. Hier werden nur die zum *BOT*-Zeitpunkt von T_j gültigen (persistenten) Versionen der Datenobjekte berücksichtigt.

5.1.2.9 Verwendung der Anfragesprache GReQL

Werden innerhalb einer Transaktion T_i die Funktionalitäten der Anfragesprache GReQL aus dem Paket *de.uni_koblenz.jgralab.greql2* (siehe Abbildung 2.1, Seite 14) verwendet, muss sichergestellt sein, dass nur die für T_i aktuell gültigen (temporären) Versionen der Datenobjekte berücksichtigt werden.

5.1.3 Anpassungen für die Konfliktüberprüfung und -erkennung

Für das Transaktionskonzept müssen zur Konfliktüberprüfung die in Abschnitt 4.2.3 (Seite 85) eingeführten Mengen in jeder Transaktion T_i mitgeführt werden. Um diesen Aspekt

umzusetzen, müssen die im Folgenden erläuterten Anpassungen im JGraLab vorgenommen werden.

5.1.3.1 Verwaltung der Mengen zur Konfliktüberprüfung und -erkennung

Führt eine Transaktion T_i eine Schreiboperation aus, müssen zum Zwecke der Konfliktüberprüfung und -erkennung den entsprechenden Mengen Referenzen der betroffenen Datenobjekte hinzugefügt werden. Dies sollte jedoch nur dann erfolgen, falls keine Exception durch die jeweilige Schreiboperation ausgelöst wurde.

5.1.4 Neu zu entwickelnde Aspekte des Transaktionskonzepts

Das Anlegen von Sicherungspunkten und das Zurücksetzen einer Transaktion T_i auf einen Sicherungspunkt (siehe Abschnitt 4.3.3, Seite 108), die Umsetzung der Konfliktüberprüfung während der Validierungsphase (siehe Abschnitt 4.2.3.9, Seite 93) und das Speichern von persistenten Änderungen während der Schreibphase (siehe Abschnitt 4.2.3.12, Seite 103) stellen Aspekte dar, die vollständig neu implementiert werden müssen.

Um die Atomarität bei der Ausführung der *BOT*-, *ABORT* und *COMMIT*-Operation einer Transaktion T_i zu garantieren, müssen geeignete Synchronisationsmechanismen im JGraLab umgesetzt werden.

5.2 Objektorientierter Feinentwurf des Transaktionskonzepts

Im Folgenden wird auf den objektorientierten Feinentwurf des Transaktionskonzepts und auf dessen Integration in die aktuelle Version von JGraLab eingegangen. Dabei wird die Umsetzung der in Abschnitt 5.1 angesprochenen Aspekte erläutert.

5.2.1 Paketstruktur

Für die Umsetzung des Transaktionskonzepts im JGraLab wird die Paketstruktur wiefolgt angepasst:

- Für die Definition von Interfaces für das Transaktionskonzept wird ein neues Paket *de.uni_koblenz.jgralab.trans* anlegt.

- Die Standardimplementierung für Graphen-, Knoten- und Kanteninstanzen wird in dem Paket *de.uni_koblenz.jgralab.impl.std* umgesetzt. Die entsprechenden Implementationsklassen mit Transaktionsunterstützung werden im Paket *de.uni_koblenz.jralab.impl.trans* erstellt.

  Im schon vorhandenen Paket *de.uni_koblenz.jgralab.impl* setzen die Implementationsklassen für Graphen-, Knoten- und Kanteninstanzen nur diejenige Funktionalität um, welche von der Standardimplementierung und der Implementierung mit Transaktionsunterstützung gleichermaßen verwendet werden können.

Für den aus einem Schema generierten Code muss die Paketstruktur ebenfalls angepasst werden. So wird für das Beispiel aus Abbildung 2.3 (Seite 19) die Standardimplementierung (aus Analogiegründen) vom Paket *de.uni_koblenz.motorwaymap.impl* in das Paket *de.uni_koblenz.motorwaymap.impl.std* verschoben. Für die Implementationsklassen mit Transaktionsunterstützung wird das Paket *de.uni_koblenz.motorwaymap.impl.trans* erzeugt.

Da die Definition neuer Interfaces für das Transaktionskonzept im generierten Code nicht notwendig ist, entfällt die Erzeugung des Pakets *de.uni_koblenz.motorwaymap.trans*.

5.2.2 Wiederverwendung bereits vorhandener Funktionalität

Ein wichtiges Ziel bei der Umsetzung des Transaktionskonzepts ist die Wiederverwendung von bereits vorhandener Funktionalität. Dies bedeutet, dass möglichst viel von der im Paket *de.uni_koblenz.jgralab.impl* vorhandenen Implementation für das Transaktionskonzept übernommen werden sollte. Abbildung 5.1 zeigt die Realisierung dieser Zielsetzung.

Zur besseren Unterscheidung werden im Folgenden die Klassen aus den Paketen *de.uni_koblenz.jgralab.impl.trans* und *de.uni_koblenz.motorwaymap.impl.trans* mit dem Präfix *trans_*, die Klassen aus den Paketen *de.uni_koblenz.jgralab.impl.std* und *de.uni_koblenz.motorwaymap.impl.std* mit dem Präfix *std_* referenziert. Die Klassen im Paket *de.uni_koblenz.jgralab.impl* erhalten kein Präfix.

Die Implementationsklassen im Paket *de.uni_koblenz.jgralab.impl* enthalten nur noch diejenigen Attribute, die *nicht* von der Versionierung im Transaktionskonzept (siehe Abschnitt 4.2.3, Seite 85) betroffen sind. Für den Zugriff auf die zu versionierenden Attribute werden abstrakte *getter*- und *setter*-Methoden definiert, die in den (direkten) Unterklassen entsprechend implementiert werden müssen. Das Auslesen und Setzen der entsprechenden Attributwerte erfolgt dabei ausschließlich und einheitlich über die zugehörigen getter- und setter-Methoden.

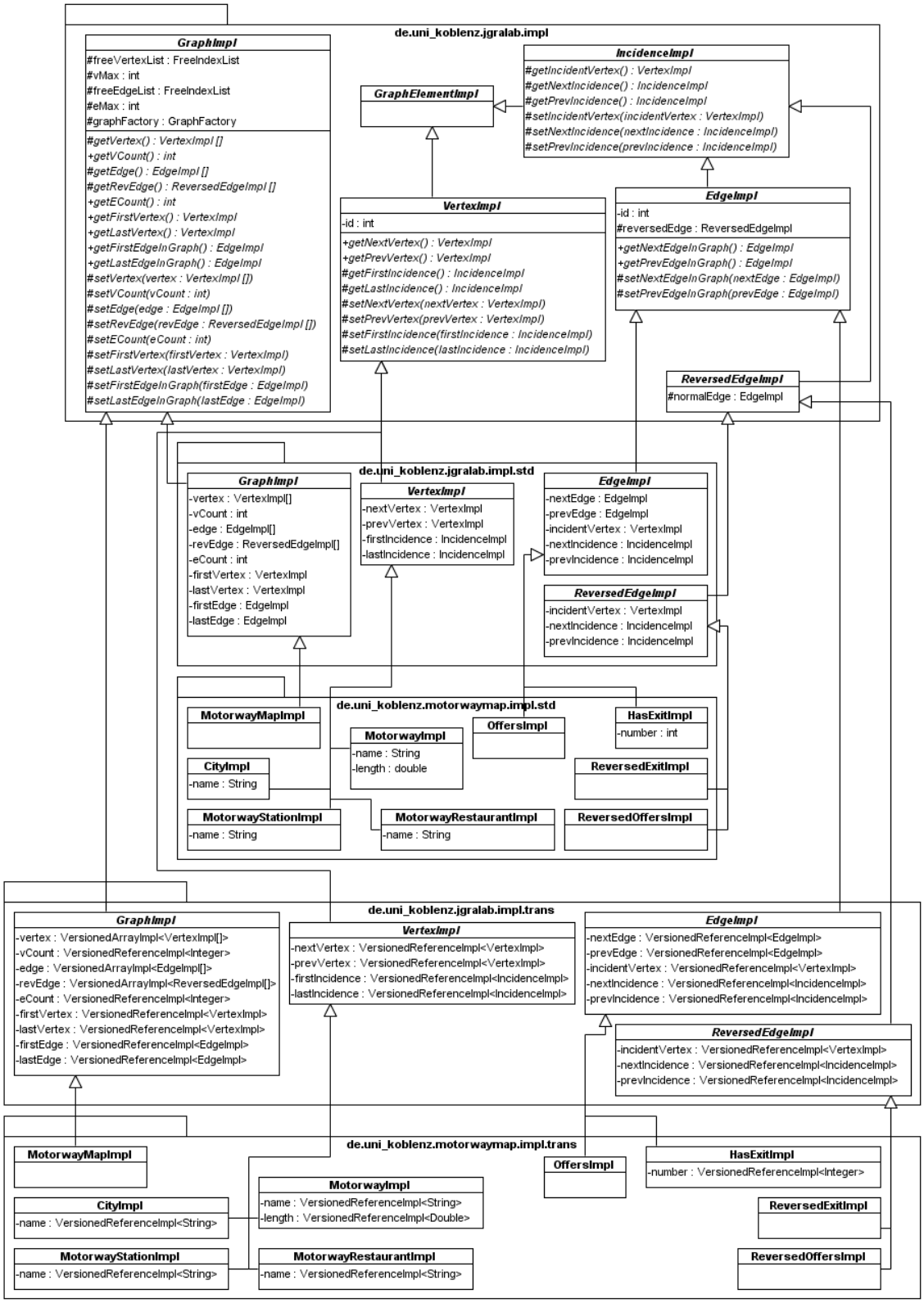


Abbildung 5.1: Vererbungsstruktur für die Umsetzung des Transaktionskonzepts

Die Klassen *std_GraphImpl*, *std_VertexImpl*, *std_EdgeImpl* und *std_ReversedEdgeImpl* und die Klassen *trans_GraphImpl*, *trans_VertexImpl*, *trans_EdgeImpl* und *trans_ReversedEdgeImpl* enthalten die in *GraphImpl*, *VertexImpl*, *EdgeImpl* bzw. *ReversedEdgeImpl* fehlenden Attribute in ursprünglicher bzw. in versionierter Form. Die genaue Umsetzung der Versionierung wird in Abschnitt 5.2.12 erläutert.

Um Codeduplizierung zu vermeiden und die in Abbildung 5.1 gezeigte Vererbungsstruktur umsetzen zu können, werden die im Rahmen des Transaktionskonzepts zu versionierenden Attribute aus *IncidenceImpl* in die Unterklassen *std_EdgeImpl* und *std_ReversedEdgeImpl* bzw. *trans_EdgeImpl* und *trans_ReversedEdgeImpl* verschoben.

Für den generierten Code kann die Codeduplizierung nicht vermieden werden, da die Einführung gemeinsamer Basisklassen für die gleichnamigen Klassen in den Paketen *de.uni_koblenz.motorwaymap.impl.std* und *de.uni_koblenz.motorwaymap.impl.trans* nicht sinnvoll umsetzbar ist.

Die Klassen *std_MotorwayImpl*, *std_CityImpl*, *std_ MotorwayStationImpl* und *std_MotorwayRestaurantImpl* halten ihre Attribute in ursprünglicher Form. Die Klassen *trans_MotorwayImpl*, *trans_CityImpl*, *trans_ MotorwayStationImpl* und *trans_MotorwayRestaurantImpl* besitzen die Attribute entsprechend in versionierter Form.

5.2.3 Erzeugen von Graphen und Graphenelementen

Beim Erzeugen eines neuen Graphen, beim Laden eines bereits existierenden Graphen und beim Erzeugen von Graphenelementen muss sichergestellt werden, dass die jeweils für das Transaktionskonzept vorgesehenen Klassen bei der Initialisierung von Graphen- und Graphenelementinstanzen zum Einsatz kommen.

5.2.3.1 Anpassungen der Klasse GraphFactoryImpl

Für die Instanziierung von Graphen und Graphenelementen ist im JGraLab die *GraphFactory* verantwortlich (siehe Abschnitt 2.3, Seite 21). Für die Integration des Transaktionskonzepts sind zunächst Anpassungen an der abstrakten Implementationsklasse *GraphFactoryImpl* notwendig (siehe Listing 5.7).

```
1 public abstract class GraphFactoryImpl implements GraphFactory {
2   protected HashMap<Class<? extends Graph>,
3     Constructor<? extends Graph>> graphMap;
4   protected HashMap<Class<? extends Edge>,
5     Constructor<? extends Edge>> edgeMap;
```

```
6   protected HashMap<Class<? extends Vertex>,
7     Constructor<? extends Vertex>> vertexMap;
8   protected HashMap<Class<? extends Graph>,
9     Constructor<? extends Graph>> graphTransactionMap;
10  protected HashMap<Class<? extends Vertex>,
11    Constructor<? extends Vertex>> vertexTransactionMap;
12  protected HashMap<Class<? extends Edge>,
13    Constructor<? extends Edge>> edgeTransactionMap;
14
15  public void setGraphImplementationClass(
16    Class<? extends Graph> originalClass,
17    Class<? extends Graph> implementationClass) { ... }
18
19  public void setVertexImplementationClass(
20    Class<? extends Vertex> originalClass,
21    Class<? extends Vertex> implementationClass) { ... }
22
23  public void setEdgeImplementationClass(
24    Class<? extends Edge> originalClass,
25    Class<? extends Edge> implementationClass) { ... }
26
27  public void setGraphTransactionImplementationClass(
28    Class<? extends Graph> originalClass,
29    Class<? extends Graph> implementationClass) { ... }
30
31  public void setVertexTransactionImplementationClass(
32    Class<? extends Vertex> originalClass,
33    Class<? extends Vertex> implementationClass) { ... }
34
35  public void setEdgeTransactionImplementationClass(
36    Class<? extends Edge> originalClass,
37    Class<? extends Edge> implementationClass) { ... }
38
39  public Graph createGraph(Class<? extends Graph> graphClass, String id,
40    int vMax, int eMax) { ... }
41
42  public Vertex createVertex(Class<? extends Vertex> vertexClass, int id,
43    Graph g) { ... }
44
45  public Edge createEdge(Class<? extends Edge> edgeClass,
46    int id, Graph g) { ... }
47
48  public Graph createGraphWithTransactionSupport(
49    Class<? extends Graph> graphClass,
50    String id, int vMax, int eMax) { ... }
51
52  public Vertex createVertexWithTransactionSupport(
53    Class<? extends Vertex> vertexClass,
54    int id, Graph g) { ... }
```

```
55
56    public Edge createEdgeWithTransactionSupport(
57      Class<? extends Edge> edgeClass,
58      int id, Graph g) { ... }
59  }
```

Listing 5.7: Anpassungen der Klasse *GraphFactoryImpl*

Zunächst werden die (neuen) Maps *graphTransactionMap* (Zeile 9), *vertexTransactionMap* (Zeile 11) und *edgeTransactionMap* (Zeile 13) eingeführt. Diese speichern jeweils die Zuordnungen zwischen den aus einem Schema generierten Interfaces und den zugehörigen Konstruktoren der generierten Implementationsklassen für das Transaktionskonzept.

Die Zuordnungen werden jeweils mit Hilfe der Methoden *setGraphTransactionImplementationClass* (Zeile 27), *setVertexTransactionImplementationClass* (Zeile 31) und *setEdgeTransactionImplementationClass* (Zeile 35) gesetzt.

Für die Instanziierung von Graphen-, Knoten- und Kanteninstanzen für das Transaktionskonzept werden jeweils die Methoden *createGraphWithTransactionSupport* (Zeile 48), *createVertexWithTransactionSupport* (Zeile 52) bzw. *createEdgeWithTransactionSupport* (Zeile 56) verwendet.

5.2.3.2 Anpassungen der Klasse MotorwayMapSchemaFactory

Für das Beispielschema *MotorwayMapSchema* (siehe Abbildung 1.4, Seite 11) muss die zugehörige generierte *GraphFactory*-Implementationsklasse *MotorwayMapSchemaFactory* angepasst werden (siehe Listing 5.8).

```
1  public class MotorwayMapSchemaFactory extends GraphFactoryImpl {
2
3    protected void fillTable() {
4      // standard implementation classes
5      setGraphImplementationClass(
6        de.uni_koblenz.motorwaymap.MotorwayMap.class,
7        de.uni_koblenz.motorwaymap.impl.std.MotorwayMapImpl.class);
8      ...
9      // implementation classes for transaction concept
10     setGraphTransactionImplementationClass(
11       de.uni_koblenz.motorwaymap.MotorwayMap.class,
12       de.uni_koblenz.motorwaymap.impl.trans.MotorwayMapImpl.class);
13     setVertexTransactionImplementationClass(
14       de.uni_koblenz.motorwaymap.Motorway.class,
15       de.uni_koblenz.motorwaymap.impl.trans.MotorwayImpl.class);
```

```
16      setVertexTransactionImplementationClass(
17        de.uni_koblenz.motorwaymap.MotorwayStation.class,
18        de.uni_koblenz.motorwaymap.impl.trans.MotorwayStationImpl.class);
19      setVertexTransactionImplementationClass(
20        de.uni_koblenz.motorwaymap.MotorwayRestaurant.class,
21        de.uni_koblenz.motorwaymap.impl.trans.MotorwayRestaurantImpl.class);
22      setVertexTransactionImplementationClass(
23        de.uni_koblenz.motorwaymap.City.class,
24        de.uni_koblenz.motorwaymap.impl.trans.CityImpl.class);
25      setEdgeTransactionImplementationClass(
26        de.uni_koblenz.motorwaymap.HasExit.class,
27        de.uni_koblenz.motorwaymap.impl.trans.HasExitImpl.class);
28      setEdgeTransactionImplementationClass(
29        de.uni_koblenz.motorwaymap.Offers.class,
30        de.uni_koblenz.motorwaymap.impl.trans.OffersImpl.class);
31    }
32  }
```

Listing 5.8: Anpassungen der Klasse *MotorwayMapSchemaFactory*

So muss die Klasse *MotorwayMapSchemaFactory* dafür sorgen, dass die Zuordnungen zwischen den generierten Interfaces aus dem Paket *de.uni_koblenz.motorwaymap* und den entsprechenden für das Transaktionskonzept generierten Implementationsklassen aus dem Paket *de.uni_koblenz.motorwaymap.impl.trans* vorgenommen werden. Dies geschieht innerhalb der Methode *fillTable* (Zeile 3) ab Zeile 10 mit Hilfe der in Listing 5.7 eingeführten Methoden *setGraphTransactionImplementationClass*, *setVertexTransactionImplementationClass* und *setEdgeTransactionImplementationClass*.

5.2.4 Erzeugen neuer Graphen mit Transaktionsunterstützung

Wie bereits in Abschnitt 2.2.3 (Seite 20) erläutert, ist die Klasse *MotorwayMapSchema* (siehe Abbildung 2.4, Seite 20) für das Erzeugen und Laden von Graphen des Typs *MotorwayMap* (*ohne* Transaktionsunterstützung) verantwortlich. Listing 5.9 zeigt einen Ausschnitt der Implementierung der Klasse *MotorwayMapSchema* mit den Methoden *createMotorwayMap* (Zeile 9) und *loadMotorwayMap* (Zeile 14). Bei der Initialisierung der *Singleton*-Instanz der Klasse *MotorwayMapSchema* wird im Konstruktor in Zeile 6 eine Instanz der GraphFactory-Klasse *MotorwayMapSchemaFactory* erzeugt.

```
1   public class MotorwayMapSchema extends SchemaImpl {
2     static MotorayMapSchema theInstance = null;
3     ...
4     private MotorayMapSchema() {
5       ...
6       graphFactory = new MotorayMapSchemaFactory();
7     }
8     ...
```

```
9     public MotorwayMap createMotorwayMap(String id, int vMax, int eMax) {
10      return (MotorwayMap) graphFactory.createGraph(MotorwayMap.class, id,
11        vMax, eMax);
12    }
13    ...
14    public MotorwayMap loadMotorwayMap(String filename, ProgressFunction pf)
15    throws GraphIOException {
16      Graph graph = GraphIO.loadGraphFromFile(filename, pf);
17      ...
18      return (MotorwayMap) graph;
19    }
20  }
```

Listing 5.9: Ausschnitt aus der Implementierung der Klasse *MotorwayMapSchema*

Listing 5.10 zeigt die Anpassungen der Klasse *MotorwayMapSchema*, um auch das Erzeugen und Laden von Graphen vom Typ *MotorwayMap mit* Transaktionsunterstützung zu erlauben.

```
1   public class MotorwayMapSchema extends SchemaImpl {
2     static MotorwayMapSchema theInstance = null;
3     ...
4     private MotorwayMapSchema() {
5       ...
6       graphFactory = new MotorayMapSchemaFactory();
7     }
8     ...
9     public MotorwayMap createMotorwayMap(String id, int vMax, int eMax) {
10      return (MotorwayMap) graphFactory.createGraph(MotorwayMap.class, id,
11        vMax, eMax);
12    }
13    ...
14    public MotorwayMap createMotorwayMapWithTransactionSupport(
15      String id, int vMax, int eMax) {
16      return (MotorwayMap) graphFactory.createGraphWithTransactionSupport(
17        MotorwayMap.class, id, vMax, eMax);
18    }
19    ...
20    public MotorwayMap loadMotorwayMap(String filename, ProgressFunction pf)
21    throws GraphIOException {
22      Graph graph = GraphIO.loadGraphFromFile(filename, pf);
23      ...
24      return (MotorwayMap) graph;
25    }
26    ...
27    public MotorwayMap loadMotorwayMapWithTransactionSupport(
28      String id, int vMax, int eMax) {
29      Graph graph = GraphIO.loadGraphFromFileWithTransactionSupport(
30        filename, pf);
```

```
31      ...
32      return (MotorwayMap) graph;
33    }
34  }
```

Listing 5.10: Anpassungen der Klasse *MotorwayMapSchema*

Für das Erzeugen neuer Grapheninstanzen vom Typ *MotorwayMap* mit Transaktionsunterstützung wird die Klasse *MotorwayMapSchema* durch die Methode *createMotorwayMapWithTransactionSupport* (Zeile 14) erweitert. Diese ruft in Zeile 16 die in Listing 5.7 eingeführte Methode *createGraphWithTransactionSupport* der abstrakten Klasse *GraphFactoryImpl* auf.

5.2.5 Laden von Graphen mit Transaktionsunterstützung

Für das Laden von Grapheninstanzen vom Typ *MotorwayMap* mit Transaktionsunterstützung wird die Klasse *MotorwayMapSchema* durch die Methode *loadMotorwayMapWithTransactionSupport* (Zeile 27) erweitert. Diese ruft in Zeile 29 die (neu einzuführende) statische Methode *loadGraphFromFileWithTransactionSupport* der Klasse *GraphIO* auf.

Die weiteren Details zu den notwendigen Anpassungen der Klasse *GraphIO* zum Laden von Grapheninstanzen mit Transaktionsunterstützung werden im Rahmen dieser Diplomarbeit nicht erläutert.

5.2.6 Repräsentation einer Transaktion

Eine Transaktion wird im JGraLab durch das Interface *Transaction* repräsentiert, welches in Abbildung 5.2 dargestellt wird.

Transaction definiert die folgenden Methoden:

- **bot()**, **commit()** und **abort()** zur Durchführung der entsprechenden Steuerungsoperationen *BOT*, *COMMIT* bzw. *ABORT* einer Transaktion.
- **getID()** zur Ermittlung der Transaktions-ID.
- **isReadOnly()** zur Abfrage, ob es sich um eine *Read-Only*-Transaktion handelt.
- **getState()** zur Ermittlung des aktuellen Zustands der Transaktion.
- **defineSavepoint()** und **restoreSavepoint(Savepoint savepoint)** zum Setzen eines Sicherungspunkts bzw. zum Zurücksetzen des Transaktionszustands auf einen gegebenen Sicherungspunkt (*savepoint*).

- **removeSavepoint(Savepoint savepoint)** zum Entfernen des übergebenen Sicherungspunkts (*savepoint*).

- **getSavepoints()** zur Ermittlung aller angelegten Sicherungspunkte für eine Transaktion.

- **isInConflict()** zur Durchführung der Validierungsphase, also der Prüfung auf mögliche durch die Transaktion verursachte Konflikte.

- **getGraph()** zur Ermittlung der Grapheninstanz, für welche die Transaktion gültig ist.

- **getThread()** zur Ermittlung des Java-Threads, in dem die Transaktion zu einem Zeitpunkt aktiv ist. In Abschnitt 5.2.7.1 wird die Notwendigkeit dieser Methode erläutert.

- **isValid()** zur Abfrage, ob die Transaktion noch gültig ist. Eine Transaktion T_i ist nicht (mehr) gültig, falls sich T_i im Zustand *TransactionState.COMMITTED* oder *TransactionState.ABORTED* befindet (siehe Abschnitt 5.2.10).

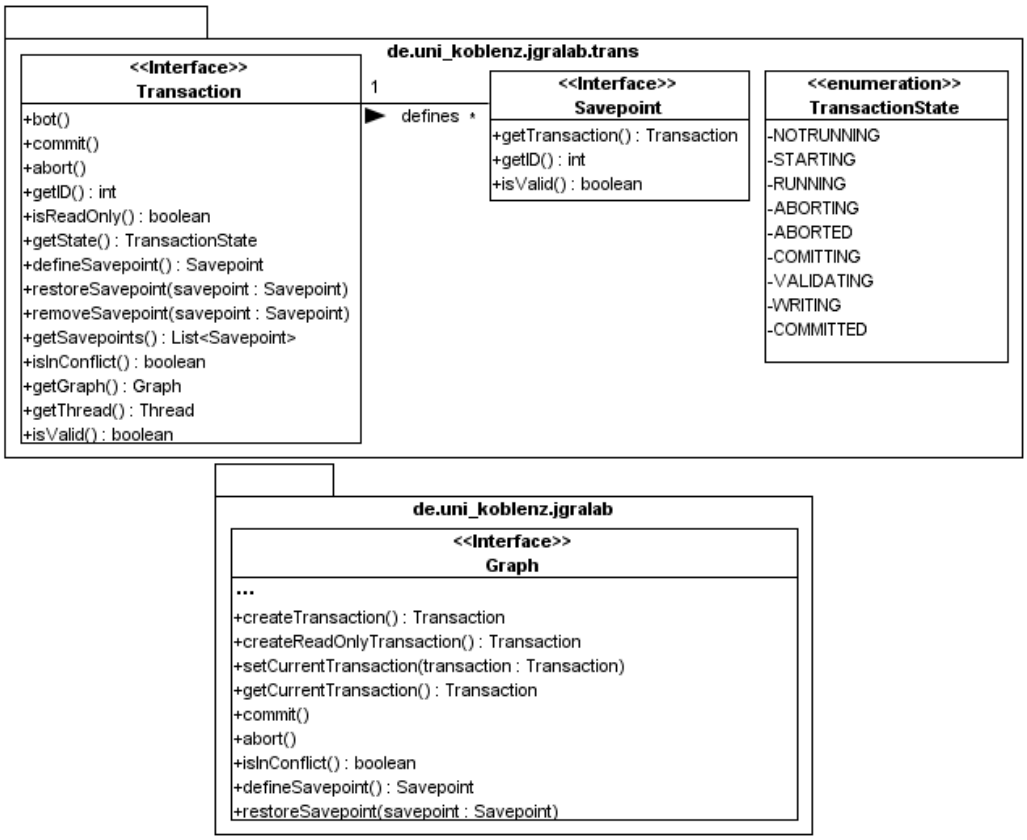


Abbildung 5.2: Repräsentation einer Transaktion

Erläuterungen zu dem erweiterten Interface *Graph*, der Enumeration *TransactionState* und dem Interface *Savepoint* erfolgen im Abschnitt 5.2.7.1, Abschnitt 5.2.10 bzw. Abschnitt 5.2.16.

Die in Abbildung 5.2 angegebenen Interfaces stellen die vollständige Schnittstelle dar, welche dem Anwender von JGraLab für die Verwendung des Transaktionskonzepts zur Verfügung gestellt wird. Die im Folgenden darüber hinaus definierten Interfaces und Klassen sind ausschließlich für die interne Umsetzung des Transaktionskonzepts relevant.

5.2.7 Definition von Abhängigkeiten für das Transaktionskonzept

Abbildung 5.3 zeigt die definierten Abhängigkeiten zwischen Java-Threads, Transaktionen und Graphen, welche im Folgenden näher erläutert werden.

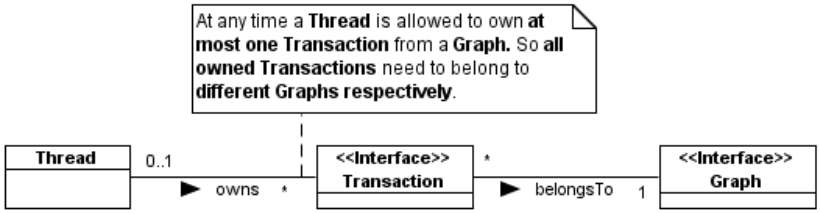


Abbildung 5.3: Abhängigkeiten zwischen Threads, Transaktionen und Graphen

5.2.7.1 Abhängigkeiten zwischen Java-Threads und Transaktionen

Innerhalb eines Java-Threads dürfen zu einem Zeitpunkt beliebig viele Transaktionen aktiv sein, die jedoch jeweils zu unterschiedlichen Graphen gehören müssen. Es dürfen also zu einem Zeitpunkt nicht zwei Transaktionen T_1 und T_2 existieren, für die T_1.getGraph() = T_2.getGraph und T_1.getThread() = T_2.getThread() gilt. So wird sichergestellt, dass zu einem Zeitpunkt für eine Grapheninstanz eindeutig bestimmt werden kann, welche Transaktion in einem Thread aktiv ist. Eine Transaktion kann während ihrer Ausführungszeit in beliebig vielen unterschiedlichen Java-Threads ablaufen. Eine Transaktion darf jedoch zu einem Zeitpunkt in höchstens einem Thread aktiv sein.

5.2.7.2 Abhängigkeit zwischen Transaktionen und Grapheninstanzen

Darüberhinaus darf eine Transaktion nur im Kontext der Grapheninstanz verwendet werden, für welche sie erzeugt wurde. Der Versuch eine „fremde“ Transaktion für eine Gra-

pheninstanz *g* zu verwenden, löst eine entsprechende Exception aus. Eine Transaktion T_i wird als „fremde“ Transaktion für *g* bezeichnet, falls T_i*.getGraph()* != *g* gilt.

5.2.8 Erstellung und Verwaltung von Transaktionen

In diesem Abschnitt wird die *Erstellung* und *Verwaltung* von Transaktionen erläutert.

5.2.8.1 Schnittstelle für den Anwender

Dem Anwender von JGraLab stehen dafür die Methoden des (erweiterten) Interfaces *Graph* (siehe Abbildung 5.2) zur Verfügung, die im Folgenden erläutert werden. Alle Methodenaufrufe werden intern an den *Transaktionsmanager* (siehe Abschnitt 5.2.8.2) delegiert, welcher dem Anwender verborgen bleibt.

- **createTransaction()** und **createReadOnlyTransaction()** ermöglichen die Erzeugung einer *Read-Write-* bzw. einer *Read-Only*-Transaktion für die jeweilige Grapheninstanz. Diese wird für den aktuellen Thread als (neue) aktive Transaktion gesetzt.

- Mit Hilfe der Methode **setCurrentTransaction(Transaction transaction)** kann die übergebene Transaktion (*transaction*) als (neue) aktive Transaktion im aktuellen Thread für die jeweilige Grapheninstanz gesetzt werden.

- **getCurrentTransaction()** liefert die zu einem Zeitpunkt im aktuellen Thread aktive Transaktion für die jeweilige Grapheninstanz oder *null*, falls der jeweiligen Grapheninstanz zum Zeitpunkt des Methodenaufrufs *keine* Transaktion im aktuellen Thread zugeordnet ist.

- Die Methoden **commit()**, **abort()**, **isInConflict()**, **defineSavepoint()** und **restoreSavepoint(Savepoint savepoint)** delegieren ihre Aufrufe an die zu einem Zeitpunkt im aktuellen Thread gültige Transaktion T_i für die jeweilige Grapheninstanz *g* (T_i = g.getCurrentTransaction()). Ist *g* im aktuellen Thread (noch) keine Transaktion zugeordnet (g.getCurrentTransaction() = null), wird eine Exception geworfen.

5.2.8.2 Der Transaktionsmanager

Für die interne Erzeugung und Verwaltung von Transaktionsinstanzen ist der Transaktionsmanager verantwortlich. Die zugehörige Klasse *TransactionManagerImpl* wird in

Abbildung 5.4 dargestellt. Jeder Grapheninstanz *g* ist jeweils eine Instanz des Transaktionsmanagers zugeordnet, welcher die Verwaltung der für *g* zugehörigen Transaktionen übernimmt.

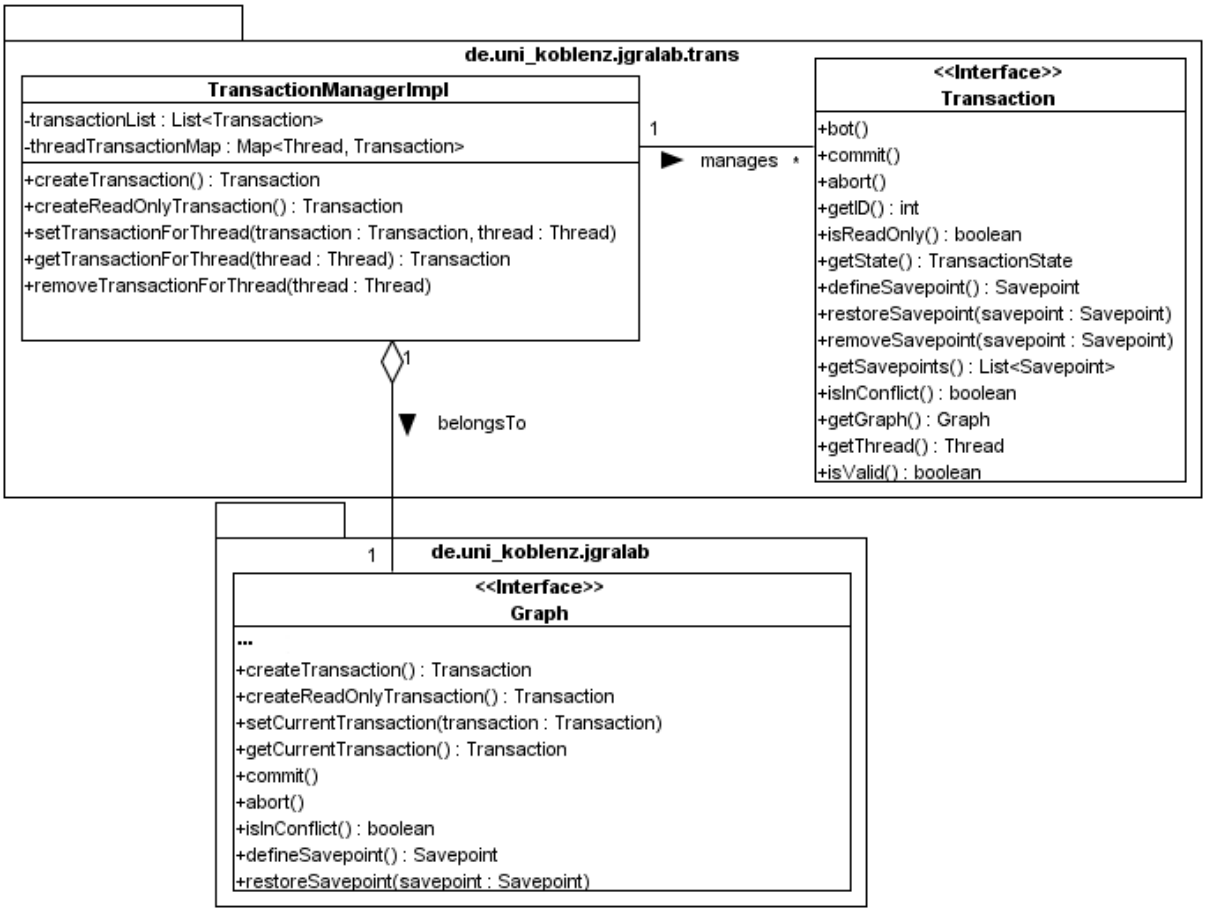


Abbildung 5.4: Repräsentation des Transaktionsmanagers

TransactionManagerImpl besitzt die folgenden Methoden:

- Die Methoden **createTransaction()** und **createReadOnlyTransaction()** erzeugen eine neue *Read-Write*- bzw. *Read-Only*-Transaktion.

- **setTransactionForThread(Transaction transaction, Thread thread)** setzt die übergebene Transaktionsinstanz (*transaction*) als für den übergebenen Thread (*thread*) aktive und gültige Transaktion. Handelt es sich bei *transaction* um eine für *g* „fremde“ Transaktion (*transaction.getGraph() != g*) muss eine entsprechende Exception geworfen werden. Ist *transaction* vor dem Methodenaufruf einem anderen Thread *oldThread* != *thread* zugeordnet, wird diese Bindung aufgehoben, so dass der Aufruf *getTransactionForThread(oldThread)* den Wert *null* ergibt.

- **getTransactionForThread(Thread thread)** liefert zum übergebenen Thread (*thread*) die zugehörige Transaktion oder *null*, falls diesem Thread (noch) keine Transaktion der Grapheninstanz *g* zugeordnet wurde.

- **removeTransactionForThread(Thread thread)** löst die zu einem Zeitpunkt gültige Zuordnung zwischen dem übergebenen Thread (*thread*) und der zugehörigen Transaktion auf. Dies kann beispielsweise nach dem erfolgreichen *COMMIT* oder nach dem *ABORT* einer Transaktion notwendig sein.

Der Transaktionsmanager besitzt mit dem Feld *transactionList* eine Liste aller (noch) gültigen Transaktionen für die jeweilige Grapheninstanz. Im Feld *threadTransactionMap* wird die Zuordnung zwischen den Threads und den Transaktionen der Grapheninstanz gespeichert. Der Transaktionsmanager muss dabei sicherstellen, dass einem Thread zu einem Zeitpunkt höchstens eine Transaktion der jeweiligen Grapheninstanz zugeordnet ist.

5.2.9 Vorgehensweisen zur Nutzung des Transaktionskonzepts

Für die Nutzung des Transaktionskonzepts sind die folgenden Vorgehensweisen denkbar:

1. Jede Transaktion läuft in einem eigenen Java-Thread ab. Dies bedeutet, dass für jede Transaktion ein neuer Java-Thread erzeugt wird, dem eine Referenz auf die jeweilige Grapheninstanz übergeben werden muss. Mit der Methode *createTransaction* wird die Transaktionsinstanz für den aktuellen Java-Thread erzeugt. Es entfällt das manuelle Setzen von Transaktionen in Java-Threads mittels *setCurrentTransaction*.

   Bei dieser Vorgehensweise steigt die Systembelastung mit der wachsenden Anzahl der im System aktiven Transaktionen.

2. Alle Transaktionen laufen im gleichen Java-Thread ab. Dies erfordert das manuelle Umschalten von Transaktionen im entsprechenden Java-Thread mittels *setCurrentTransaction*.

   Da nur ein Thread existiert, ist hier die Systembelastung (relativ) gering(er).

3. Für die praktische Nutzung des Transaktionskonzepts sind Szenarien wahrscheinlicher, in denen mehrere Threads existieren, welche jeweils mehrere Transaktionen unterschiedlicher Grapheninstanzen besitzen. Innerhalb eines Threads wird (bei Bedarf) mittels *setCurrentTransaction* zwischen den Transaktionen einer Grapheninstanz geschaltet.

   Die Regulierung der Systembelastung wird dem Anwender von JGraLab überlassen.

5.2.10 Transaktionszustände

Die für eine Transaktion T_i annehmbaren Transaktionszustände werden in der Enumeration *TransactionState* aus Abbildung 5.5 definiert. Hierbei ist anzumerken, dass im Vergleich zu der Definition der Transaktionszustände in Abschnitt 4.1.2 (Seite 58) weitere Zwischenzustände eingeführt werden. *Starting*, *Aborting* und *Committing* signalisieren die (noch nicht abgeschlossene) Ausführung der *bot*-, *abort*- bzw. *commit*-Methode durch T_i. *Validating* und *Writing* geben an, dass sich T_i in der Validierungs- bzw. in der Schreibphase befindet.

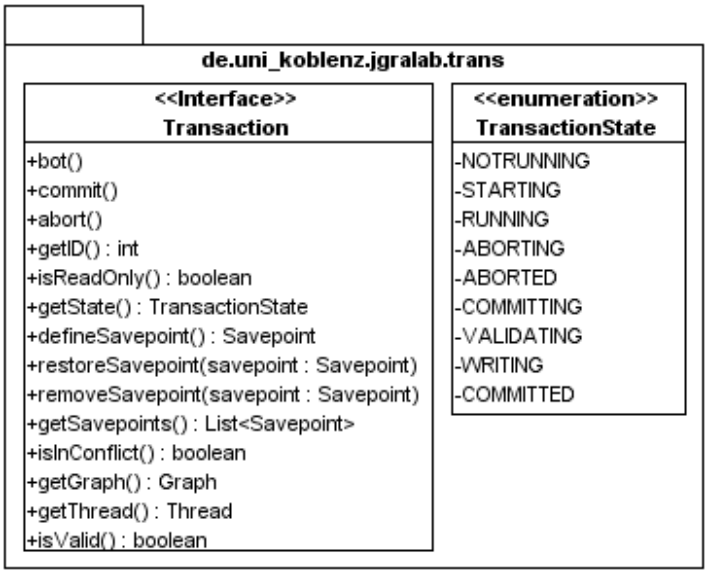


Abbildung 5.5: Die Enumeration *TransactionState*

Abbildung 5.6 stellt die Zustandsübergänge innerhalb einer Transaktion T_i dar. Nach Initialisierung der Transaktionsinstanz befindet sich T_i im Zustand *NotRunning*. Sobald T_i ihre *bot*-Methode aufruft, wechselt T_i in den Zustand *Starting*. Falls sich mindestens eine parallel laufende Transaktion T_j in der Schreibphase (*Writing*) befindet, muss T_i im Zustand *Starting* warten (*wait*).

Befindet sich keine parallel laufende Transaktion T_j mehr im Zustand *Writing*, geht T_i in den Zustand *Running* über. T_i ist es nur in diesem Zustand erlaubt, Lese- und Schreiboperationen auf der zugehörigen Grapheninstanz auszuführen. Um diesbezüglich den Umgang mit Transaktionen für den Anwender von JGraLab zu vereinfachen, sollte die *bot*-Methode *implizit* bei der Initialisierung einer Transaktionsinstanz ausgeführt werden.

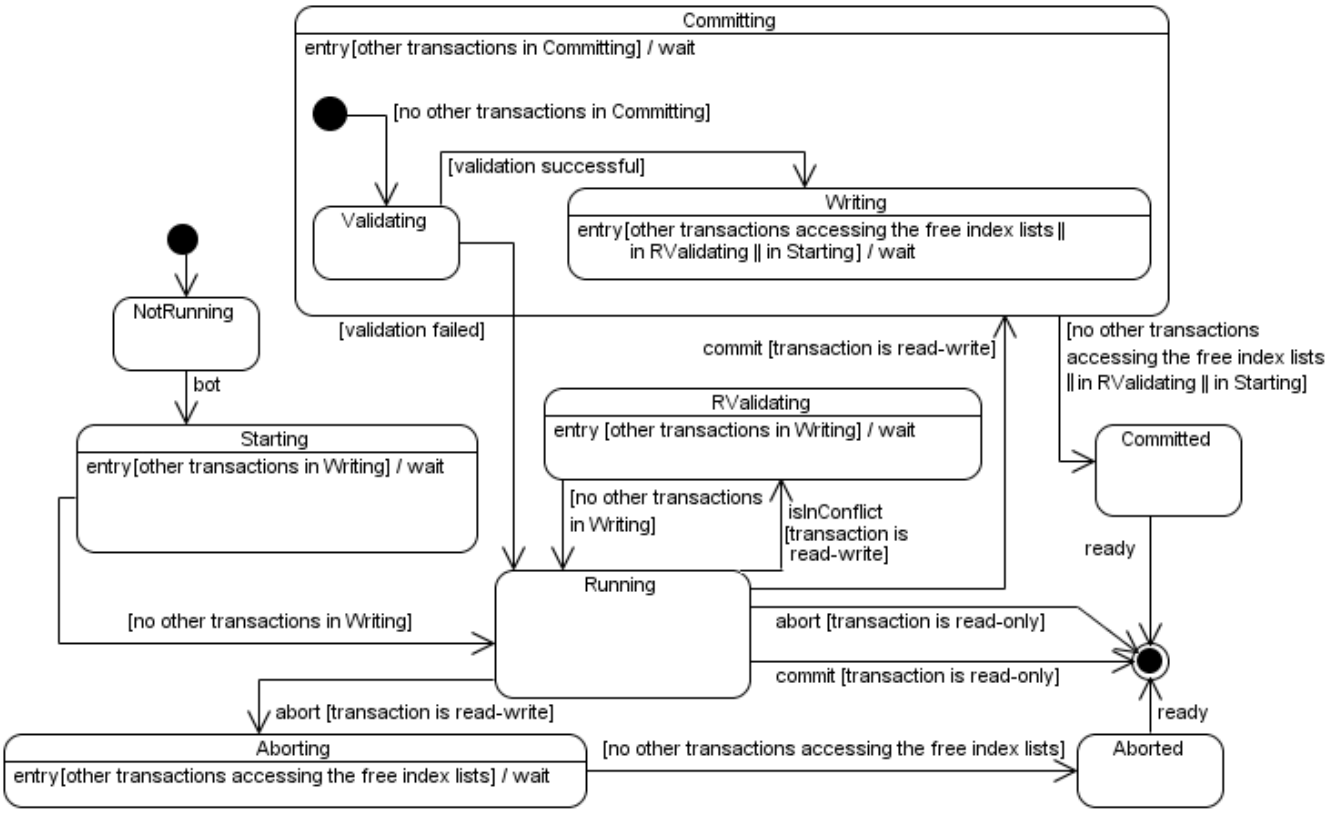


Abbildung 5.6: Zustandsdiagramm für eine Transaktion

Wird im Zustand *Running* die *abort*-Methode ausgeführt, wechselt (die *Read-Write*-Transaktion) T_i in den Zustand *Aborting*. Zu beachten ist dabei, dass die von T_i alloziierten Knoten- und Kanten-IDs wieder freigegeben werden und somit der Zugriff auf die jeweiligen Freispeicherlisten synchronisiert werden muss. Dies kann das Verzögern von T_i im Zustand *Aborting* zur Folge haben. Wurden die Freispeicherlisten erfolgreich aktualisiert, wechselt T_i in den Zustand *Aborted* und wird aus dem System entfernt. Handelt es sich bei T_i um eine *Read-Only*-Transaktion, wird T_i bei Ausführung der *abort*-Methode im Zustand *Running* direkt aus dem System entfernt.

Wird für (die *Read-Write*-Transaktion) T_i im Zustand *Running* die Methode *isInConflict* ausgeführt, wird die Validierung für T_i durchgeführt, falls sich keine parallel laufende Transaktion T_j im Zustand *Writing* befindet. In Abbildung 5.6 wird dies durch den Zustand *RValidating* dargestellt, welcher (genauso wie der Zustand *Validating*) den Wert *TransactionState.VALIDATING* repräsentiert. Ist die Validierung abgeschlossen, kehrt T_i in den Zustand *Running* zurück.

Führt (die *Read-Write*-Transaktion) T_i im Zustand *Running* die *commit*-Methode aus, wechselt diese in den (zusammengesetzten) Zustand *Committing*. Führt mindestens eine weitere parallel laufende Transaktion T_j ihr *COMMIT* aus, muss T_i im Zustand *Committing* warten. Handelt es sich bei T_i um eine *Read-Only*-Transaktion, wird T_i bei Ausführung der *commit*-Methode im Zustand *Running* direkt aus dem System entfernt.

Der *COMMIT* einer *Read-Write*-Transaktion T_i setzt sich aus der Validierungs- und der Schreibphase zusammen. Existiert keine andere parallel laufende Transaktion T_j mehr, die sich im Zustand *Committing* befindet, betritt T_i zunächst die Validierungsphase. Werden dabei Konflikte festgestellt, wechselt die Transaktion T_i wieder in den Zustand *Running*. Sind keine Konflikte aufgetreten, wird die Schreibphase durchgeführt. Die Durchführung der Schreibphase ist nur möglich, falls sich keine weiteren parallel laufenden Transaktionen T_j im Zustand *Starting* oder *RValidating* befinden. Auch hier müssen analog zum Zustand *Aborting* die Zugriffe auf die Freispeicherlisten synchronisiert ablaufen. Dies kann das Verzögern von T_i im Zustand *Writing* zur Folge haben. Anschließend wechselt T_i in den Zustand *Committed* und wird aus dem System entfernt.

5.2.11 Synchronisierung im Kontext von Transaktionen

In diesem Abschnitt wird die Umsetzung der im vorangegangenen Abschnitt 5.2.10 angesprochenen Synchronisationsmaßnahmen erläutert. Zur Synchronisierung werden dabei zwei Java-Mechanismen verwendet. Zum einen werden Synchronisationsobjekte eingesetzt, mit denen Anweisungsblöcke durch das Schlüsselwort *synchronized* exklusiv gesperrt werden können, so dass zu einem Zeitpunkt nur höchstens ein Thread den jeweiligen geschützten Anweisungsblock betreten darf [Ull07]. Zum anderen werden Sperren vom Typ *ReadWriteLock* eingesetzt, um Konflikte beim parallelen Ausführen von Anweisungsblöcken zu vermeiden. Ein ReadWriteLock *rwl* besitzt sowohl eine Lese- (*readLock*) als auch eine Schreibsperre (*writeLock*). Dabei kann mehreren Threads zur gleichen Zeit die Lesesperre von *rwl* zugewiesen werden, während die Vergabe der Schreibsperre von *rwl* exklusiv erfolgt [Ull07]. Die Regeln zur Vergabe der Lese- und Schreibsperre für ein ReadWriteLock entsprechen den Regeln zur Vergabe von Lese- und Schreibsperren bei der sperrbasierten Synchronisation (siehe Tabelle 3.10, Seite 50).

Die Verwaltung der *ReadWriteLocks* zur Synchronisierung der *bot()*-, *commit()*- und *isInConflict()*-Methoden für Transaktionen einer Grapheninstanz *g* erfolgt im zugehörigen Transaktionsmanager. Dabei werden:

- der ReadWriteLock *botWritingSync*,
- der ReadWriteLock *commitSync* und
- der ReadWriteLock *commitValidatingSync* eingeführt.

Listing 5.11 zeigt einen Ausschnitt aus der Implementationsklasse *TransactionImpl* des Interfaces *Transaction*, anhand dessen die Umsetzung der Synchronisierung im Folgenden erläutert wird.

Um sicherzustellen, dass eine Transaktion T_i nicht ihr *BOT* durchführt, während sich eine andere Transaktion T_j parallel in der Schreibphase befindet, wird der ReadWriteLock *botWritingSync* verwendet. In der Methode *bot* (Zeile 6) wird eine Lesesperre (*readLock()*) gesetzt (Zeile 8). Zudem wird die Schreibphase in der Methode *commit* (Zeile 15) mit einer Schreibsperre (*writeLock()*) versehen (Zeile 23). Die Atomarität der Schreibphase und des BOTs ist dadurch für jede Transaktion T_i garantiert. Dabei können mehrere Transaktionen parallel ihr BOT durchführen, solange sich keine weitere Transaktion zur gleichen Zeit in der Schreibphase befindet.

Durch die Schreibsperre des ReadWriteLocks *commitSync* in Zeile 17 innerhalb der Methode *commit* wird sichergestellt, dass der *COMMIT* einer jeden Transaktion T_i atomar ausgeführt wird. Es können also zu einem Zeitpunkt keine zwei Transaktionen T_i und T_j existieren, die parallel ihr *COMMIT* durchführen.

```
1  public class TransactionImpl implements Transaction {
2    private TransactionManagerImpl transactionManager;
3    private GraphImpl graph;
4    private TransactionState state;
5    ...
6    public void bot() {
7      if (state == TransactionState.NOTRUNNING) {
8        transactionManager.getBotWritingSync().readLock().lock();
9        // set persistentVersionAtBot
10       transactionManager.getBotWritingSync().readLock().unlock();
11       state = TransactionState.RUNNING;
12     }
13   }
14
15   public void commit() {
16     state = TransactionState.COMMITTING;
17     transactionManager.getCommitSync().writeLock.lock();
18     if (!internalIsInConflict()) {
19       state = TransactionState.RUNNING;
20       // throw Exception
21     }
22     transactionManager.getCommitValidatingSync().writeLock().lock();
23     transactionManager.getBotWritingSync().writeLock().lock();
24     // execute writing-phase
25     if (deletedVertices.size() > 0) {
26       synchronized (graph.getFreeVertexList()) {
27         ...
28       }
29     }
```

```
30      if (deletedEdges.size() > 0) {
31        synchronized (graph.getFreeEdgeList()) {
32          ...
33        }
34      }
35      transactionManager.getBotWritingSync().writeLock().unlock();
36      transactionManager.getCommitValidatingSync().writeLock().unlock();
37      transactionManager.getCommitSync().writeLock.unlock();
38      state = TransactionState.COMMITTED;
39    }
40
41    public boolean isInConflict() {
42      state = TransactionState.VALIDATING;
43      boolean result = false;
44      transactionManager.getCommitValidatingSync().readLock().lock();
45      result = internalIsInConflict();
46      transactionManager.getCommitValidatingSync().readLock().unlock();
47      state = TransactionState.RUNNING;
48      return result;
49    }
50
51    private boolean internalIsInConflict() {
52      state = TransactionState.VALIDATING;
53      // return true if conflict, false otherwise
54      ...
55    }
56
57    public void abort() {
58      state = TransactionState.ABORTING;
59      if (addedVertices.size() > 0) {
60        synchronized (graph.getFreeVertexList()) {
61          ...
62        }
63      }
64      if (addedEdges.size() > 0) {
65        synchronized (graph.getFreeEdgeList()) {
66          ...
67        }
68      }
69      state = TransactionState.ABORTED;
70    }
71    ...
72  }
```

Listing 5.11: Umsetzung der Synchronisierung

Zudem muss sichergestellt werden, dass während der Schreibphase einer Transaktion T_i keine weitere Transaktion T_j parallel die Methode *isInConflict* (Zeile 41) durchführen

kann und vice versa. Dafür wird in Zeile 44 für den ReadWriteLock *commitValidating-Sync* die Lesesperre innerhalb der Methode *isInConflict* gesetzt. In der Methode *commit* wird die Schreibphase mit der Schreibsperre für *commitValidatingSync* geschützt (Zeile 22). Analog zur Methode *bot* können mehrere Transaktionen parallel ihre *isInConflict*-Methode durchführen, solange sich keine weitere Transaktion zur gleichen Zeit in der Schreibphase befindet.

Der synchronisierte Zugriff auf die Freispeicherliste der Knoten (*freeVertexList*) und die Freispeicherliste der Kanten (*freeEdgeList*) in den Methoden *commit* (Zeile 15) und *abort* (Zeile 57) wird durch entsprechende *synchronized*-Blöcke in den Zeilen 26 und 60 bzw. in den Zeilen 31 und 65 sichergestellt. Als Synchronisationsobjekte kommen dabei die jeweiligen Freispeicherlisten zum Einsatz.

5.2.12 Umsetzung der Versionierung

Um die Isolation von Transaktionen sicherzustellen, wird im JGraLab die Mehrversionen-Synchronisation (in Kombination mit der optimistischen Synchronisation) verwendet (siehe Abschnitt 4.2.3, Seite 85). So hat jede Transaktion T_i für alle Datenobjekte *x* lediglich Zugriff auf die persistenten Versionen x_p, die zum *BOT*-Zeitpunkt von T_i gültig waren und auf die für T_i erzeugten temporären Versionen x_t. Die Details zur Umsetzung der Versionierung und zur Verwaltung von temporären und persistenten Versionen der Datenobjekte für eine Grapheninstanz werden im Folgenden erläutert.

5.2.12.1 Die Umsetzung der Versionenzähler

Zur Umsetzung des persistenten Versionenzählers *persistentVersionCounter* (siehe Abschnitt 4.2.3.2, Seite 86) wird das bereits vorhandene *long*-Attribut *graphVersion* der Klasse *std_GraphImpl* (siehe Abschnitt 5.1.2.6, Seite 120) verwendet. Es muss hierfür sichergestellt werden, dass der Wert von *graphVersion* nur bei persistenten Änderungen inkrementiert wird und bei temporären Änderungen unverändert bleibt.

Zur Umsetzung des temporären Versionenzählers *temporaryVersionCounter* (siehe Abschnitt 4.2.3.2, Seite 86) für jede Transaktion T_i wird ein *long*-Attribut *temporaryVersionCounter* in der Klasse *TransactionImpl* eingeführt.

5.2.12.2 Das generische Interface VersionedDataObject<E>

Für die Umsetzung der Versionierung (wie in Abschnitt 4.2.3 auf Seite 85 beschrieben) wird für alle zu versionierenden Datenobjekte ein einheitliches generisches Interface *VersionedDataObject<E>* definiert (siehe Abbildung 5.7).

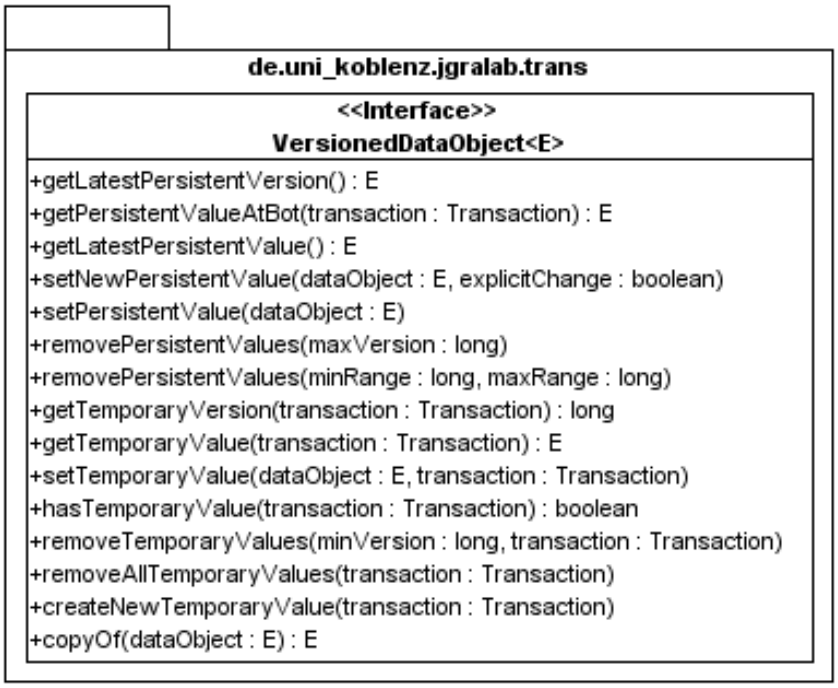


Abbildung 5.7: Das generische Interface *VersionedDataObject<E>*

VersionedDataObject<E> deklariert dabei die folgenden Methoden für ein Datenobjekt *x* vom Typ *E*:

- **getLatestPersistentVersion()** liefert die Versionsnummer der letzten *explizit* geänderten persistenten Version von *x*. Diese Methode wird zur Umsetzung der Konfliktüberprüfung benötigt. Durch den Vergleich der von *getLatestPersistentVersion* zurückgelieferten Versionsnummer und dem Wert T_i.*persistentVersionAtBot* können persistente Änderungen an *x* seit dem *BOT* einer Transaktion T_i festgestellt werden (siehe Abschnitt 4.2.3.7, Seite 91).

- **getPersistentValueAtBot(Transaction transaction)** liefert die persistente Version von *x*, welche zum *BOT*-Zeitpunkt der übergebenen Transaktion *transaction* gültig war. Diese Methode wird unter anderem für *Read-Only*-Transaktionen benötigt, welche lediglich Zugriff auf die zu ihrem *BOT*-Zeitpunkt gültige persistente Version von *x* haben dürfen.

- **getLatestPersistentValue()** liefert die letzte persistente Version von *x*. Diese Methode wird bei der Konfliktüberprüfung zur Erkennung von *Lost Updates* und *Phantomen* benötigt (siehe Abschnitt 4.2.3.7, Seite 91).

- **setNewPersistentValue(E dataObject, boolean explicitChange)** erzeugt eine neue persistente Version von *x* mit dem Wert von *dataObject*. Der Parameter *explicitChange* gibt dabei an, ob *x.persistentVersion* aktualisiert werden soll oder nicht. Die Aktualisierung sollte beispielsweise bei der *impliziten* Änderung des Nachfolge- oder Vorgängerknotens eines Knotens *v* in *Vseq* entfallen und nur bei *expliziten* Änderungen erfolgen (siehe Abschnitt 4.2.3.3, Seite 87). Davon unabhängig muss der neuen persistenten Version von *x* eine neue Versionsnummer zugeordnet werden. Näheres dazu findet sich in Abschnitt 5.2.12.3.

- **setPersistentValue(E dataObject)** aktualisiert die zum Zeitpunkt des Methodenaufrufs gültige persistente Version von *x* mit dem Wert von *dataObject*. Diese Methode ist für den Fall notwendig, dass eine Transaktion T_i in ihrer Schreibphase bereits eine neue persistente Version x_p von *x* mittels *setNewPersistentValue* erzeugt hat und x_p im weiteren Verlauf der Schreibphase aktualisiert werden muss.

- **removePersistentValues(long maxVersion)** entfernt alle *persistenten* Versionen von *x* mit einer Versionsnummer *n* < *maxVersion*. Diese Methode ist für die *Garbage Collection* (siehe Abschnitt 4.2.3.13, Seite 107) notwendig. Sie sollte nur für den Fall aufgerufen werden, dass die betroffenen persistenten Versionen von *x* von keiner (noch) aktiven Transaktion referenziert werden.

- **removePersistentValues(long minRange, long maxRange)** dient ebenfalls zur *Garbage Collection* von nicht mehr benötigten persistenten Versionen. Dabei werden alle persistenten Versionen des Datenobjekts *x* mit einer Versionsnummer *n* > *minRange* und *n* < *maxRange* entfernt.

- **getTemporaryVersion(Transaction transaction)** liefert für *transaction* die Versionsnummer der zu dem Zeitpunkt gültigen temporären Version von *x*. Diese Methode ist notwendig, um nach dem Anlegen eines Sicherungspunkts *s* für eine Transaktion T_i feststellen zu können, ob eine neue temporäre Version von *x* für T_i erstellt werden muss, um die Gültigkeit von *s* zu garantieren (siehe Abschnitt 4.3.3.1, Seite 109).

- **getTemporaryValue(Transaction transaction)** liefert für die übergebene Transaktion *transaction* die zu einem Zeitpunkt gültige temporäre Version des Datenobjekts *x*.

- **setTemporaryValue(E dataObject, Transaction transaction)** aktualisiert die zum Zeitpunkt des Methodenaufrufs gültige temporäre Version von *x* der Transaktion *transaction* mit dem Wert von *dataObject*.

- **hasTemporaryValue(Transaction transaction)** überprüft, ob für die übergebene Transaktion *transaction* bereits eine temporäre Version von *x* erstellt wurde.

- **removeTemporaryValues(long minVersion, Transaction transaction)** entfernt alle temporären Versionen von *x* mit einer Versionsnummer *n* > *minVersion* für die Transaktion *transaction*. Diese Methode ist notwendig, um nicht mehr benötigte temporäre Versionen von *x* nach dem Zurücksetzen der Transaktion *transaction* auf einen Sicherungspunkt *s* zu löschen. *minVersion* sollte beim Methodenaufruf dem Wert *s.versionAtSavepoint* entsprechen.

- **removeAllTemporaryValues(Transaction transaction)** entfernt alle für die übergebene Transaktion *transaction* angelegten temporären Versionen von *x*. Diese Methode dient der *Garbage Collection* nach dem erfolgreichen *COMMIT* oder *ABORT* von *transaction*.

- **createNewTemporaryValue(Transaction transaction)** erzeugt für *transaction* eine neue temporäre Version von x_t mit einer neuen Versionsnummer. Hierbei stellt x_t eine Kopie von *getTemporaryValue(transaction)* dar, falls *hasTemporaryValue(transaction) = true* gilt. Ergibt *hasTemporaryValue(transaction) = false* ist x_t eine Kopie von dem Datenobjekt *getPersistentValueAtBot(transaction)*.

- **copyOf(E dataObject)** liefert zu dem übergebenen Datenobjekt *dataObject* eine (identische) Kopie zurück. Das Kopieren von Datenobjekten wird detailliert in Abschnitt 5.2.12.4 thematisiert.

5.2.12.3 Die generische Klasse VersionedDataObjectImpl<E>

Die generische abstrakte Klasse *VersionedDataObjectImpl<E>* (siehe Abbildung 5.8) implementiert das Interface *VersionedDataObject<E>* und übernimmt die (getrennte) zentrale Verwaltung der temporären und persistenten Versionen eines Datenobjekts *x* vom Typ *E*.

Ein wichtiger Aspekt ist dabei der möglichst geringe und effiziente Speicherplatzverbrauch, so dass die in Abschnitt 4.2.3.1 (Seite 85) beschriebenen Vorgehensweisen zur Erzeugung und Verwaltung von temporären und persistenten Versionen im Folgenden unter diesem Gesichtspunkt optimiert und angepasst werden.

Verwaltung von temporären Versionen

Bei der Verwaltung von temporären Versionen des Datenobjekts *x* werden die Maps *temporaryValueMap* und *temporaryVersionMap* verwendet.

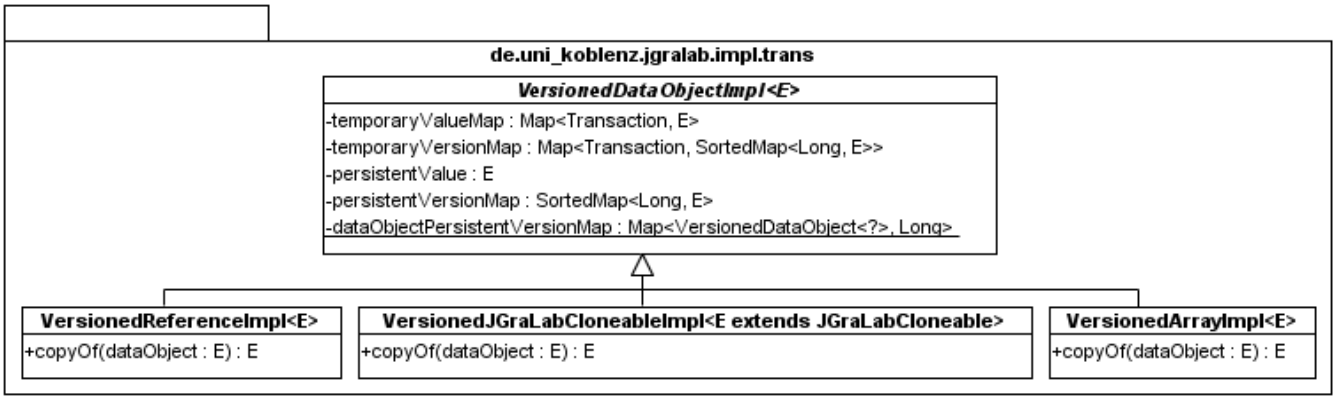


Abbildung 5.8: Die generische Klasse *VersionedDataObjectImpl<E>*

Die Map *temporaryValueMap* ordnet einer Transaktion T_i den zugehörigen gültigen temporären Wert x_t zu. Die Zuordnung einer Versionsnummer für x_t im Kontext von T_i entfällt dabei. Solange T_i keinen Sicherungspunkt definiert, wird für T_i ausschließlich *temporaryValueMap* verwendet, da das Mitführen von Versionsnummern nur notwendig ist, falls T_i mindestens einen Sicherungspunkt besitzt.

Die Map *temporaryVersionMap* ordnet einer Transaktion T_i eine (nach der Versionsnummer) *sortierte* Map von temporären Versionen zu. Letztere Map ordnet einer temporären Versionsnummer die zugehörige temporäre Version von *x* zu. Der letzte Schlüssel *k* dieser Map entspricht der zu einem Zeitpunkt gültigen temporären Versionsnummer und der zu *k* zugehörige Wert der zu einem Zeitpunkt gültigen temporären Version von *x* für T_i.

Definiert T_i einen Sicherungspunkt *s* und gilt für ein durch T_i geändertes Datenobjekt *x*, dass der zugehörigen temporären Version x_t noch keine Versionsnummer zugeordnet wurde (*temporaryVersionMap.contains(*T_i*) = false*) muss für $versionMap_{T_i}$ = *temporaryVersionMap.get(*T_i*)* die Operation $versionMap_{T_i}$*.put(s.versionAtSavepoint, temporaryValueMap.get(*T_i*))* ausgeführt werden. Der in der Map *temporaryValueMap* für T_i gespeicherte Wert von x_t wird also in Verbindung mit der im Sicherungspunkt *s* gespeicherten Versionsnummer in die Map *temporaryVersionMap* verschoben, um die Gültigkeit von *s* zu garantieren. Zudem wird die Operation *temporaryValueMap.remove(*T_i*)* ausgeführt.

Tritt für ein Datenobjekt *x* im Kontext von T_i der Fall auf, dass für T_i nur noch eine temporäre Version x_t in *temporaryVersionMap* existiert ($versionMap_{T_i}$*.size() = 1*) und T_i keinen Sicherungspunkt (mehr) besitzt, werden die Operationen *temporaryValueMap.put(*T_i*,* $versionMap_{T_i}$*.get(*$versionMap_{T_i}$*.lastKey()))* und *temporaryVersionMap.remove(*T_i*)* durchgeführt. Da hier das Mitführen einer Versionsnummer für x_t nicht (mehr) notwendig ist, wird x_t von der Map *temporaryVersionMap* in die Map *temporaryValueMap* verschoben.

Generell kann man auf die Verwendung von *temporaryValueMap* zur Speicherung von temporären Versionen eines Datenobjekts *x* verzichten und stattdessen für jede Transaktion T_i lediglich die Map *temporaryVersionMap* verwenden. Allerdings kann durch die Nutzung der Map *temporaryValueMap* Speicherplatz gespart werden, da das Mitführen von (temporären) Versionsnummern entfällt. Dieser Aspekt kann vor allem bei einer hohen Anzahl von zu versionierenden Datenobjekten relevant sein. Daraus folgt auch, dass das Anlegen von Sicherungspunkten einen deutlich höheren Speicherplatzbedarf zur Folge haben kann, da hier das Mitführen von temporären Versionsnummern erforderlich ist.

Verwaltung der temporären Versionen in TransactionImpl

Eine weitere Maßnahme zur Minimierung des Speicherplatzbedarfs wird in Abbildung 5.9 dargestellt. Hier werden die Maps *temporaryValueMap* und *temporaryVersionMap* in die Klasse *TransactionImpl* verschoben. Diese werden für alle zu versionierenden Datenobjekte einer Transaktion T_i verwendet, so dass T_i lokal alle zugehörigen temporären Versionen verwaltet.

Dadurch wird die Objektgröße für eine (langlebige) Instanz der Klasse *VersionedDataObjectImpl<E>* verringert, da die Instanziierungen der Maps *temporaryValueMap* und *temporaryVersionMap* entfallen. Da eine (Read-Write-)Transaktion T_i nach ihrem *COMMIT* oder *ABORT* vollständig aus dem System entfernt wird, kann hier der durch *temporaryValueMap* und *temporaryVersionMap* für T_i belegte Speicherplatz vollständig wieder freigegeben werden. Solange T_i keine Sicherungspunkte definiert, ist eine Instanziierung der Map *temporaryVersionMap* nicht notwendig. Handelt es sich bei T_i um eine Read-Only-Transaktion, kann komplett auf eine Instanziierung der Maps verzichtet werden.

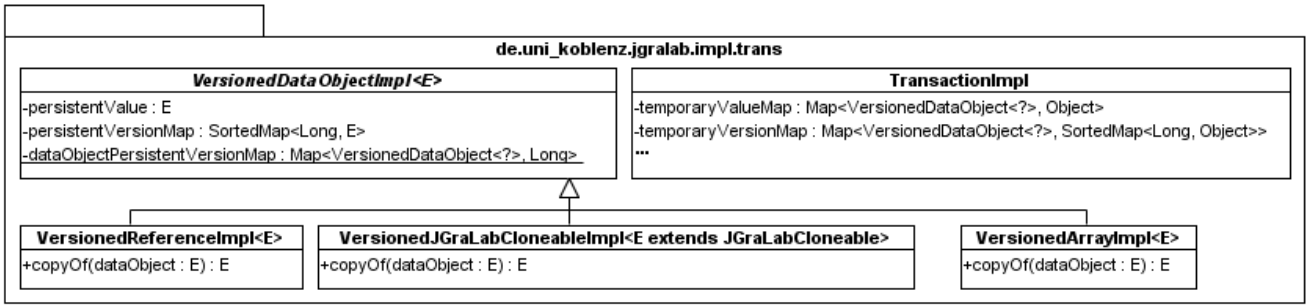


Abbildung 5.9: Verwaltung der temporären Versionen in der Klasse *TransactionImpl*

Die Verwendung von *temporaryValueMap* und *temporaryVersionMap* in Abbildung 5.9 im Vergleich zur Verwendung in Abbildung 5.8 führt zu einer höheren Schlüsselanzahl und zu einer höheren Zugriffszeit (vor allem bei einer hohen Anzahl von geänderten Datenobjekten in einer Transaktion) auf die jeweiligen Maps. Im Folgenden wird aufgrund der effizienteren Speicherverwendung die in Abbildung 5.9 dargestellte Modellierung angenommen. Die in Abbildung 5.9 abgebildete Klasse *TransactionImpl* wird in Abschnitt 5.2.13 erweitert.

Verwaltung von persistenten Versionen

Für die Verwaltung von persistenten Versionen des Datenobjekts *x* werden das Feld *persistentValue* und die Map *persistentVersionMap* benutzt (siehe Abbildung 5.9). Sowohl *persistentValue* als auch *persistentVersionMap* wird zu Beginn der Wert *null* zugeordnet.

persistentValue wird verwendet, falls zu einem Zeitpunkt nur eine persistente Version von *x* existiert, so dass das Mitführen einer persistenten Versionsnummer entfallen kann. Für diesen Fall gilt, dass *persistentVersionMap* dem Wert *null* entspricht.

Die (nach der Versionsnummer sortierte) Map *persistentVersionMap* ordnet einer persistenten Versionsnummer die entsprechende persistente Version von *x* zu. Sie wird benötigt, falls mindestens zwei persistente Versionen für *x* verwaltet werden müssen. Soll eine neue persistente Version x_p in der Schreibphase einer Transaktion T_i erzeugt werden und existiert bisher nur eine persistente Version für *x*, die aber noch von mindestens einer anderen Transaktion T_j referenziert wird, muss *persistentVersionMap.put(T_j.persistentVersionAtBot, persistentValue)* ausgeführt werden. Der in *persistentValue* gespeicherte Wert für *x* wird also in *persistentVersionMap* mit einer zugehörigen Versionsnummer eingefügt. *persistentValue* wird daraufhin auf den Wert *null* gesetzt. Die neue persistente Version x_p wird durch das Ausführen von *persistentVersionMap.put(persistentVersionCounter++, x_p)* erzeugt.

Tritt nach dem *COMMIT* einer Transaktion T_i durch die anschließende *Garbage Collection* von nicht mehr referenzierten persistenten Versionen für *x* der Fall auf, dass *persistentVersionMap.size() = 1* ist, wird *persistentValue = persistentVersionMap.get(persistentVersionMap.lastKey())* ausgeführt und *persistentVersionMap* auf den Wert *null* zurückgesetzt. Da nur noch eine persistente Version für *x* existiert, ist das Mitführen einer persistenten Versionsnummer für *x* nicht (mehr) notwendig.

Analog zu der Verwaltung von temporären Versionen kann durch die Verwendung von *persistentValue* Speicherplatz gespart werden, da das Mitführen einer persistenten Versionsnummer entfällt.

Ermittlung der aktuellen persistenten Versionsnummer

Existiert zu einem Zeitpunkt nur eine persistente Version eines Datenobjekts *x*, entspricht die für *x* gültige persistente Versionsnummer *x.persistentVersion* (siehe Abschnitt 4.2.3.2, Seite 86) dem Wert T_i*.persistentVersionAtBot* der Transaktion T_i, die *x.persistentVersion* (zum Beispiel in der Validierungsphase) ermitteln will.

Existieren mindestens zwei persistente Versionen von *x* und handelt es sich bei *x* um ein Datenobjekt, für das nicht zwischen impliziten und expliziten (persistenten) Änderungen unterschieden werden muss, entspricht *x.persistentVersion* immer dem letzten Schlüssel in *persistentVersionMap* (*x.persistentVersion = persistentVersionMap.lastKey()*).

Für ein Datenobjekt *y*, für das eine Unterscheidung zwischen impliziten und expliziten (persistenten) Änderungen erforderlich ist, sind andere Mechanismen zur Speicherung und Ermittlung von *y.persistentVersion* notwendig, falls mindestens zwei persistente Versionen für *y* existieren. Eine implizite (persistente) Änderung erfolgt für ein Datenobjekt *y*, falls die Methode *setNewPersistentVersion(E dataObject, boolean explicitChange)* für *y* mit dem Wert *false* für den Parameter *explicitChange* aufgerufen wird. Sowohl bei einer expliziten als auch bei einer impliziten (persistenten) Änderung von *y* durch die Ausführung der Methode *setNewPersistentVersion* muss eine neue persistente Version y_p mittels *persistentVersionMap.put(newPersistentVersionCounter, dataObject)* mit *newPersistentVersionCounter = persistentVersionCounter +1* angelegt werden. Nach einer expliziten Änderung von *y* muss gelten, dass *y.persistentVersion = newPersistentVersionCounter* ist. Nach einer impliziten Änderung von *y* bleibt *y.persistentVersion* jedoch unverändert. Im JGraLab sind implizite Änderungen für die Referenzen *nextVertex* und *prevVertex* der Klasse *trans_VertexImpl*, für die Referenzen *nextEdge* und *prevEdge* der Klasse *trans_EdgeImpl* und für die Referenzen *nextIncidence* und *prevIncidence* der Klassen *trans_EdgeImpl* und *trans_ReversedEdgeImpl* relevant (siehe Abbildung 5.1, Seite 125).

Um auch für solche Datenobjekte den zu einem Zeitpunkt zugehörigen gültigen Wert von *persistentVersion* ermitteln zu können, ist eine Erweiterung der Klasse *VersionedDataObjectImpl<E>* um ein Attribut *currentPersistentVersion* vom Typ *long* denkbar, in dem jeweils der Wert von *persistentVersion* gespeichert wird. Nachteilig ist dabei, dass dieses Attribut nicht für alle Datenobjekte benötigt wird und somit unnötigter Speicherplatz reserviert werden würde. Aus diesem Grund wird in Abbildung 5.8 eine *statische* Map *dataObjectPersistentVersion* in der Klasse *VersionedDataObjectImpl<E>* eingeführt. *dataObjectPersistentVersion* ordnet einem versionierten Datenobjekt (falls notwendig) die aktuell gültige persistente Versionsnummer zu. Wird eine implizite (persistente) Änderung für ein Datenobjekt *y* durchgeführt und gilt *dataObjectPersistentVersion.containsKey(y) = false*, ist die Erzeugung eines neuen Eintrags für *y* in *dataObjectPersistentVersion* erforderlich. Gilt *dataObjectPersistentVersion.containsKey(y) = true*, bleibt der zugehörige Wert von *y* in *dataObjectPersistentVersion* erhalten. Bei einer *expliziten* (persistenten) Änderung von *y* wird *dataObjectPersistentVersion.remove(y)* ausgeführt. Für jedes Datenobjekt *x* ist ein zugehöriger Eintrag in *dataObjectPersistentVersion* nur notwendig, falls mindestens zwei persistente Versionen von *x* verwaltet werden und mindestens eine (aktive) Transaktion T_i existiert, für die T_i*.isValid() = true* und *dataObjectPersistentVersion(x) <=* T_i*.persistentVersionAtBot < x.persistentVersionMap.lastKey()* gilt.

Existiert zu einem Zeitpunkt nur eine persistente Version eines Datenobjekts *x*, liefert die Methode *getLatestPersistentVersion()* (siehe Abbildung 5.7) den Wert T_i*.persistentVersionAtBot* der Transaktion T_i, welche die Methode aufgerufen hat. Existieren mindestens zwei persistente Versionen von *x* wird *dataObjectPersistentVersion.get(x)* zurückgegeben,

falls *dataObjectPersistentVersion.containsKey(x) = true* gilt. Für *dataObjectPersistentVersion.containsKey(x) = false* wird *x.persistentVersionMap.lastKey()* zurückgeliefert.

Die Notwendigkeit der ebenfalls in Abbildung 5.9 dargestellten generischen Klassen *VersionedReferenceImpl<E>*, *VersionedJGraLabCloneableImpl<E extends JGraLabCloneable>* und *VersionedArrayImpl<E>* wird in Abschnitt 5.2.12.4 erläutert.

5.2.12.4 Das Kopieren von Datenobjekten

Im Folgenden wird das Kopieren von temporären und persistenten Versionen für Datenobjekte im JGraLab erläutert.

Das Erstellen von Kopien in Java

Für das Kopieren von Objekten wird in Java üblicherweise die *clone*-Methode der Klasse *Object* im Zusammenhang mit dem (leeren) Interface *Cloneable* verwendet. Sollen Objekte einer Klasse *C* kopiert werden können, muss *C* einerseits die in der Klasse *Object* als *protected* deklarierte *clone*-Methode als *public*-Methode überschreiben. Andererseits muss die Klasse *C* das Interface *Cloneable* implementieren. Das Interface *Cloneable* dient dabei als sogenanntes *Marker*-Interface. Es handelt sich also um ein leeres Interface, welches keine eigenen Methoden (auch nicht die *clone*-Methode) deklariert und daher lediglich signalisiert, dass das Klonen von Objekten von einer Klasse *C* unterstützt wird [Krü07].

Beim Kopieren von Objekten wird immer als erstes die *clone*-Methode der Klasse *Object* aufgerufen. Diese Methode ist nicht in Java, sondern in einer (dem Betriebssystem entsprechenden) nativen Programmiersprache umgesetzt. Sie stellt sicher, dass genügend Speicher für die Objektkopie reserviert wird. Zudem führt sie eine *bitweise* Kopie des Objekts durch. Dadurch wird nur eine sogenannte *flache* Kopie (engl. *shallow copy*) eines Objekts erzeugt. In einer flachen Kopie werden die im zu kopierenden Objekt vorhandenen Referenzen ebenfalls kopiert. Die referenzierten (Unter-)Objekte selbst werden jedoch nicht mitkopiert [Krü07].

Sollen auch zusätzlich rekursiv Kopien der referenzierten (Unter-)Objekte erzeugt werden, spricht man von einer *tiefen* Kopie (engl. *deep copy*). Allerdings wird das Anlegen von tiefen Kopien in Java nicht standardmäßig durch die Kombination *clone*-Methode und *Cloneable*-Interface unterstützt. Für das Anlegen von tiefen Kopien in Java gibt es zwei Möglichkeiten:

1. Um tiefe Kopien von Objekten einer Klasse *C* erzeugen zu können, muss man durch das Überschreiben der (involvierten) *clone*-Methode(n) sicherstellen, dass alle in *C* referenzierten Objekte (rekursiv) mitkopiert werden.

2. Man bedient sich der *Serialisierung*. In Java dient diese in der Regel dazu, Objekte in serialisierter Form in eine Datei zu schreiben oder in serialisierter Form vorliegende Objekte aus einer Datei im Hauptspeicher zu rekonstruieren. Die Serialisierung kann aber auch für das Erstellen von tiefen Kopien eines Objekts verwendet werden. Dafür müssen die serialisierten Daten statt in und aus Dateien einfach im und vom Hauptspeicher gespeichert bzw. geladen werden. Die Klassen der zu serialisierenden Objekte müssen dabei das *Marker*-Interface *Serializable* implementieren [Krü07].

Die 1. Variante ist vor allem dadurch problematisch, weil das Kopieren von Objekten in Java nicht vereinheitlicht ist. Es existieren also Klassen, die das Interface *Cloneable* nicht implementieren und somit das Anlegen von Objektkopien mittels der *clone*-Methode nicht unterstützen. So bietet zum Beispiel die Klasse *String* nur einen *Copy-Konstruktor* zur Objektkopie an. Die Klasse *Enum* verhindert sogar das Anlegen von Kopien, indem sie die überschriebene *clone*-Methode als *final* deklariert und lediglich eine *CloneNotSupportedException* auslöst [Krü07].

Die 2. Variante bietet eine quasi-einheitliche Methode zur Objektkopie an, da fast alle in der Standardbibliothek enthaltenen Klassen in Java das Interface *Serializable* implementieren. Allerdings ist die Serialisierung nicht ausreichend performant für die Anwendung im Kontext des Transaktionskonzepts. Aus diesem Grund wird im Folgenden die Verwendung der *clone*-Methode (falls möglich) angenommen.

Das Erstellen von Kopien im JGraLab

Für die Versionierung im JGraLab im Kontext des Transaktionskonzepts müssen zwei Arten von Kopien unterschieden werden:

1. Zum einen müssen *Referenzen* auf Objekte versioniert werden. Zum Beispiel ist für eine Knoteninstanz die Verwaltung von (temporären und persistenten) Versionen der Referenz *nextVertex* auf den Nachfolgeknoten in *Vseq* in der Klasse *trans_VertexImpl* notwendig. Hier werden also nicht die Objekte selbst, sondern nur Referenzen auf Objekte versioniert.

2. Zum anderen werden Kopien von Objektwerten benötigt, zum Beispiel bei der Versionierung eines *List*-Attributs in einer Knoteninstanz.

Für den 1. Fall wird die Klasse *VersionedReferenceImpl<E>* verwendet (siehe Abbildung 5.9). Die Umsetzung der Methode *copyOf* wird in Listing 5.12 gezeigt. *copyOf* liefert also lediglich eine Kopie der übergebenen Referenz auf das Datenobjekt.

```
1 public class VersionedReferenceImpl<E>
2 extends VersionedDataObjectImpl<E> {
3
4   public E copyOf(E dataObject) {
5     return dataObject;
6   }
7 }
```

Listing 5.12: Die Methode *copyOf* in der Klasse *VersionedReferenceImpl<E>*

Für den 2. Fall wird im Folgenden analysiert, welche Datentypen (Klassen) von der Versionierung betroffen sind und welche Mechanismen jeweils für das Erstellen von Kopien in Java vorgesehen sind [Krü07]. Zudem wird auch untersucht, ob das Erstellen von Objekt-Kopien in einzelnen Fällen für das Transaktionskonzept überhaupt erforderlich ist.

Array — Für ein Array lässt sich mit der *clone*-Methode eine flache Kopie erstellen. Im JGraLab ist dies für die Arrays *vertex*, *edge* und *revEdge* der Klasse *trans_GraphImpl* erforderlich.

Die Erstellung von flachen Array-Kopien ist im Kontext des Transaktionskonzepts ausreichend.

String — Kopien von *String*-Objekten lassen sich standardmäßig nur über einen *Copy*-Konstruktor erstellen. Im JGraLab treten *String*-Attribute in den Implementationsklassen von Graphen-, Knoten- und Kanteninstanzen im generierten Code auf.

Wichtig ist zu erwähnen, dass *String*-Objekte sogenannte *Immutable*-Objekte sind [Krü07]. Dies bedeutet, dass der Wert eines *String*-Objekts nach dessen Initialisierung nicht mehr verändert werden kann. Will man also einen neuen Wert für ein *String*-Attribut setzen, muss ein neues *String*-Objekt mit dem entsprechenden Wert erzeugt werden.

Im Kontext der Versionierung bedeutet dies, dass für *String*-Objekte das Anlegen von Objektkopien nicht erforderlich ist. Die Kopie der Objektreferenz ist im Kontext des Transaktionskonzepts ausreichend.

Wrapperklassen primitiver Datentypen — Auch Objekte von Wrapperklassen (*Integer*, *Long*, *Double* und *Boolean*) primitiver Datentypen sind nach deren Initialisierung *unveränderlich*. Objektkopien von Wrapperklassen sind in Java standardmäßig nicht vorgesehen.

Analog zum Datentyp *String* reicht auch hier das Kopieren der Objektreferenzen für das Transaktionskonzept aus.

Enum — Für *Enum*-Klassen ist das Kopieren von Objekten (wie bereits erwähnt) nicht vorgesehen. Auch Enum-Objekte sind *unveränderlich*.

Für das Transaktionskonzept reicht das Kopieren der Objektreferenzen aus.

Record — Im JGraLab wird ein *Record* als Klasse umgesetzt. Diese enthält *public*-Attribute, welche die einzelnen *Record*-Komponenten darstellen.

Für das Transaktionskonzept muss bei der Codegenerierung sichergestellt werden, dass jede betroffene Klasse vollständig tiefe Kopien erstellen kann.

List, Set und Map — Alle (generischen) Klassen der Java-Standardbibliothek im Paket *java.util*, welche die generischen Interfaces *List<E>*, *Set<E>* oder *Map<K,V>* implementieren (also die Klassen *ArrayList<E>*, *LinkedList<E>* und *Vector<E>*, *HashSet<E>*, *LinkedHashSet<E>* und *TreeSet<E>* bzw. *HashMap<K,V>* und *TreeMap<K,V>*) erlauben das Erstellen von flachen Objekt-Kopien mit Hilfe der *clone*-Methode.

Im Kontext der Versionierung sind flache Kopien nicht ausreichend. Es müssen (vollständig) tiefe Kopien erstellt werden, so dass auch die in den Listen bzw. Mengen enthaltenen Objekte (rekursiv) mitkopiert werden.

Für den Datentyp *String*, den *Wrapperklassen primitiver Datentypen* und den Datentyp *Enum* ist das Kopieren von Referenzen für die Versionierung ausreichend. Für die betroffenen Datentypen kann also auch die Klasse *VersionedReferenceImpl<E>* verwendet werden.

Das Interface JGraLabCloneable

Für die Datentypen *Record*, *List*, *Set* und *Map* ist zunächst die Definition eines neuen Interfaces *JGraLabCloneable* notwendig (siehe Abbildung 5.10). *JGraLabCloneable* definiert dabei *explizit* die im Interface *Cloneable* fehlende *clone*-Methode.

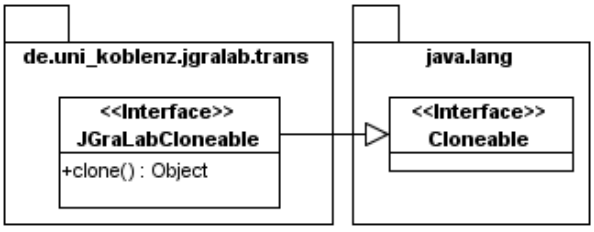


Abbildung 5.10: Das Interface *JGralabCloneable*

Eigene Implementationsklassen für List, Set und Map

Für die Datentypen *List*, *Set* und *Map* sind zudem eigene (im Kontext von Attributen ausschließlich zu nutzende) generische Implementierungen für das Transaktionskonzept erforderlich. Abbildung 5.11 führt zu diesem Zwecke die Klassen *JGraLabList<E>*, *JGraLabSet<E>* und *JGraLabMap<K,V>* ein. Diese implementieren jeweils das Interface *JGraLabCloneable*.

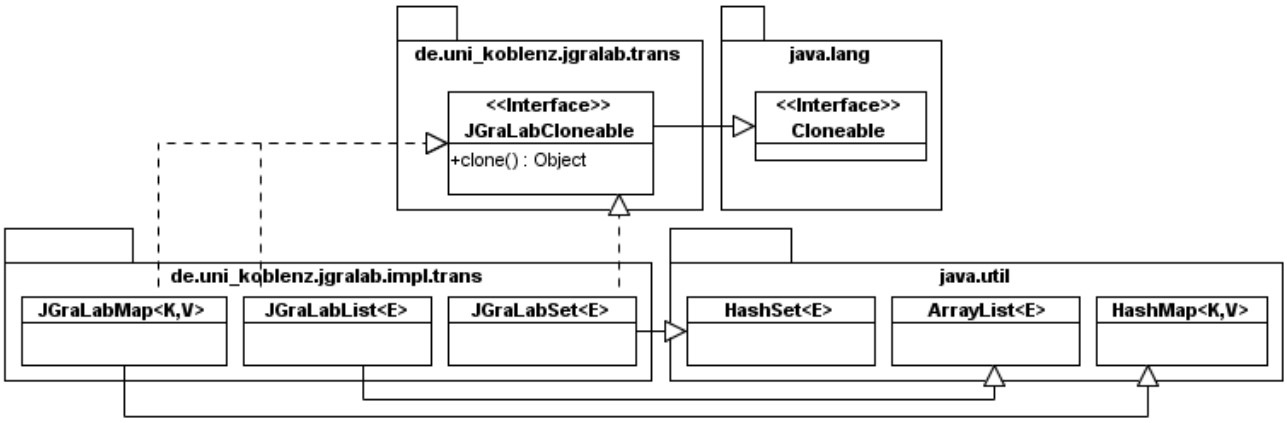


Abbildung 5.11: Die Klassen *JGraLabList<E>*, *JGraLabSet<E>* und *JGraLabMap<K,V>*

Listing 5.13 zeigt die Implementierung der *clone*-Methode in der Klasse *JGraLabList<E>*. In den Klassen *JGraLabSet<E>* und *JGraLabMap<K,V>* wird die *clone*-Methode analog umgesetzt.

```
1  public class JGraLabList<E> extends ArrayList<E>
2  implements JGraLabCloneable {
3  ...
4    public Object clone() {
5      JGraLabList<E> jgralabList = new JGraLabList<E>();
6      for (E element : this) {
7        if (element instanceof JGraLabCloneable)
8          jgralabList.add((E) ((JGraLabCloneable) element).clone());
9        else
10         jgralabList.add(element);
11     }
12     return jgralabList;
13   }
14 }
```

Listing 5.13: Die Implementierung der *clone*-Methode in der Klasse *JGraLabList<E>*

Zunächst wird in Zeile 5 eine neue Instanz der Klasse *JGraLabList<E>* erzeugt, welche die Objektkopie repräsentiert. Anschließend wird über die Originalliste iteriert. Für jedes Element (*element*) wird überprüft, ob es das Interface *JGralabCloneable* implementiert (Zeile 7). Ist dies der Fall, kann es sich im Kontext des Transaktionskonzepts nur um ein Element vom Typ *List*, *Set*, *Map* oder *Record* handeln. Auf dem Element wird die *clone*-Methode ausgeführt. Die gelieferte Kopie wird in die Liste eingefügt. Implementiert das Element das Interface *JGraLabCloneable* nicht, genügt das Hinzufügen einer Referenzkopie (Zeile 10), da der Datentyp *Array* im Kontext der Attribute im JGraLab nicht auftritt. Die gezeigte Implementierung liefert im Kontext des Transaktionskonzepts eine vollständig tiefe Kopie eines Listenobjekts.

Beim Zugriff auf ein Attribut *a* vom Typ *JGraLabList<E>*, *JGraLabSet<E>* oder *JGraLabMap<K,V>* durch eine Transaktion T_i mittels der zugehörigen *getter*-Methode erhält T_i eine Referenz auf die entsprechende gültige (temporäre oder persistente) Version von a. Hierbei muss sichergestellt werden, dass T_i nur zulässige Operationen auf dem Attribut *a* ausführt, so dass die Korrektheit der Versionierung für *a* erhalten bleibt. Falls T_i eine *Read-Only*-Transaktion ist, dürfen beispielsweise keine Schreiboperationen auf *a* durch T_i durchgeführt werden (zum Beispiel durch die Methoden *add* bzw. *put* oder *remove*). Handelt es sich bei T_i um eine *Read-Write*-Transaktion, erhält T_i beim ersten Zugriff auf *a* mittels der zugehörigen *getter*-Methode eine Referenz auf die persistente Version a_p, die zum *BOT*-Zeitpunkt von T_i gültig war. Führt T_i nun eine Schreiboperation (zum Beispiel *remove*) auf a_p während ihrer Lesephase aus, muss a_p unverändert bleiben und stattdessen eine neue temporäre Version a_t für T_i erzeugt werden. Diese Anpassungen erfordern das Überschreiben der Methoden der Klassen *ArrayList<E>*, *HashSet<E>* und *HashMap<K,V>* in *JGraLabList<E>*, *JGraLabSet<E>* bzw. *JGraLabMap<K,V>*.

Anpassungen für den Datentyp Record

Als Beispiel für ein *Record* dient die Klasse *Address* aus Listing 5.14. *Address* besitzt die Felder *firstName* (Zeile 2) und *surName* (Zeile 3) vom Typ *String* und ein Feld *phoneNumberList* (Zeile 4) vom Typ *JGraLabList<String>*.

Auch die Klasse *Address* implementiert das Interface *JGraLabCloneable* und muss demnach die *clone*-Methode umsetzen. Hier wird zunächst die *clone*-Methode der Klasse *Object* aufgerufen (Zeile 7). Anschließend wird lediglich für das Feld *phoneNumberList* eine Objektkopie erstellt (Zeile 8), da für die Felder *firstName* und *surName* eine Kopie der Referenz ausreichend ist.

Auch bei einem Attribut *a* vom Typ *Record* muss sichergestellt werden, dass durch die Ausführung von Schreiboperationen auf *a* die Korrektheit der Versionierung für *a* erhalten bleibt. Lesende und schreibende Zugriffe auf die einzelnen Elemente eines Records - welche ursprünglich durch *public*-Attribute repräsentiert werden - erfolgen daher im Kontext des Transaktionskonzepts durch entsprechende getter- und setter-Methoden. Direkte (nicht zu kontrollierende) Zugriffe auf die einzelnen Recordelemente werden also verhindert.

```
1  public class Address implements JGralabCloneable {
2    protected String firstname;
3    protected String surname;
4    protected JGraLabList<String> phoneNumberList;
5
6    public String getFirstName() { ...}
7    public String getSurName() { ...}
8    public JGraLabList<String> getPhoneNumberList() { ...}
9
10   public void setFirstName(String firstName) { ...}
11   public void setSurName(String surName) { ...}
12   public void setPhoneNumberList(JGraLabList<String>
13     phoneNumberList) { ...}
14   ...
15   public Object clone() {
16     Address address = (Address) this.clone();
17     address.phoneNumberList = phoneNumberList.clone();
18     return address;
19   }
20 }
```

Listing 5.14: Beispielklasse für den Datentyp *Record*

Zur Versionierung der Datentypen *JGraLabList<E>*, *JGraLabSet<E>*, *JGraLabMap<K,V>* und *Record* wird die in Abbildung 5.9 gezeigte generische Klasse *VersionedJGraLabCloneableImpl<E extends JGraLabCloneable>* verwendet. Listing 5.15 zeigt die Umsetzung der zugehörigen *copyOf*-Methode. Hier ist es ausreichend, auf dem zu kopierenden Datenobjekt *dataObject* die *clone*-Methode aufzurufen (Zeile 7).

```
1  public class VersionedJGraLabCloneableImpl<E extends JGraLabCloneable>
2  extends VersionedDataObjectImpl<E> {
3
4    public E copyOf(E dataObject) {
5      if(dataObject == null)
6        return null}
7      return (E) dataObject.clone();
8    }
9  }
```

Listing 5.15: Die Methode *copyOf* in der Klasse *VersionedJGraLabCloneableImpl<E extends JGraLabCloneable>*

Der Datentyp Array

Für den Datentyp *Array* wird die Klasse *VersionedArrayImpl<E>* (siehe Abbildung 5.9) verwendet. Wie bereits erwähnt, unterstützt *Array* das Erstellen von flachen Kopien mittels der *clone*-Methode. Listing 5.16 zeigt die Umsetzung der *copyOf*-Methode.

```
1  public class VersionedArrayImpl<E> extends VersionedDataObjectImpl<E> {
2
3    public E copyOf(E dataObject) {
4      if(dataObject == null)
5        return null;
6      try {
7        Method cloneMethod = dataObject.getClass().getSuperclass()
8          .getDeclaredMethod("clone");
9        cloneMethod.setAccessible(true);
10       return (E) cloneMethod.invoke(dataObject, new Object[]{});
11     } catch(Exception e) {
12       throw new GraphException("This should not happen,
13         if E is an Array-type.");
14     }
15   }
16 }
```

Listing 5.16: Die Methode *copyOf* in der Klasse *VersionedArrayImpl<E>* mit Reflection

Da in Java der Datentyp *Array* nicht als Typparameter für generische Klassen verwendet werden kann - also die Angabe VersionedArrayImpl<E[]> nicht erlaubt ist - und die *clone*-Methode für den generischen Typ *E* nicht (direkt) zur Verfügung steht, muss die *clone*-Methode per *Java-Reflection* aufgerufen werden (Zeile 7).

5.2.12.5 Implementationsklassen des Transaktionskonzepts mit Versionierung

Abbildung 5.12 zeigt die Umsetzung der Versionierung aus Abschnitt 4.2.3.3 (Seite 87) in den Implementationsklassen aus den Paketen *de.uni_koblenz.jgralab.impl.trans* und *de.uni_koblenz.motorwaymap.impl.trans* (siehe Abbildung 5.1, Seite 125).

5.2.12.6 Gültigkeitsüberprüfung für Iteratoren

Die temporären Versionen der Datenobjekte *vertexListVersion*, *edgeListVersion* und *incidenceListVersion* (siehe Abbildung 5.12) einer Transaktion T_i dienen der Gültigkeitsüberprüfung eines Iterators der Klasse *VertexIterable*, *EdgeIterable* bzw. *IncidenceIterable* (siehe Abschnitt 5.1.2, Seite 115) innerhalb von T_i.

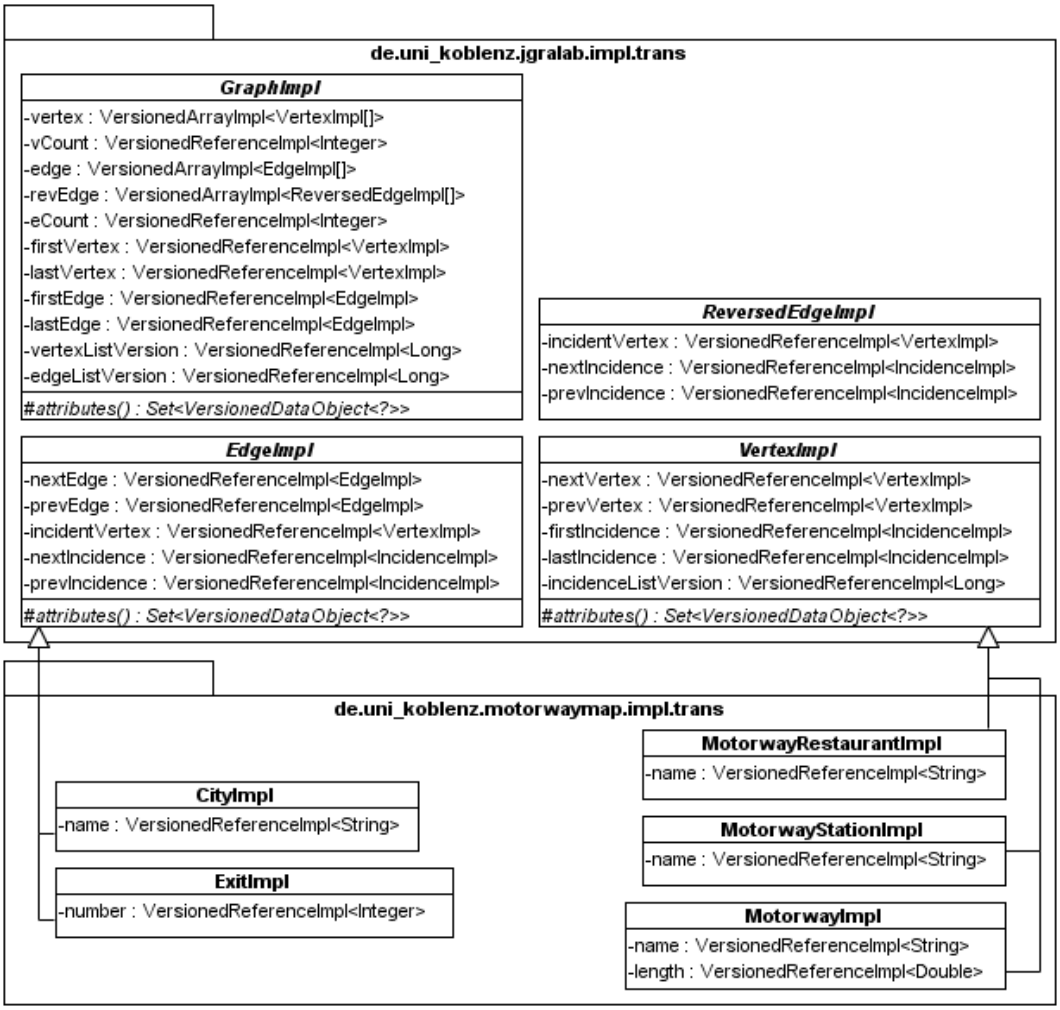


Abbildung 5.12: Implementationsklassen des Transaktionskonzepts mit Versionierung

5.2.13 Konflikterkennung

Jede Transaktion muss die für die Konflikterkennung notwendigen Mengen mitführen (siehe Abschnitt 4.2.3, Seite 85). Diese werden in der Klasse *TransactionImpl* verwaltet (siehe Abbildung 5.13).

Hierfür werden zusätzlich die Enumerations *ListPosition* und *VertexPosition* eingeführt. *ListPosition* ist notwendig, um für ein Graphenelement oder eine Inzidenz in der jeweiligen Liste (*Vseq*, *Eseq* bzw. Λseq(v) eines Knotens *v*) speichern zu können, ob sich das Vorgängerelement (*PREV*) oder das Nachfolgeelement (*NEXT*) explizit geändert hat. *VertexPosition* wird benötigt, um sich für eine Kante merken zu können, ob der *Startknoten* (*ALPHA*), der *Endknoten* (*OMEGA*) oder beide (*ALPHAOMEGA*) geändert wurden.

Für die Konflikterkennung ist zudem die Einführung der abstrakten Methode *attributes()* in den Klassen *trans_GraphImpl*, *trans_EdgeImpl* und *trans_VertexImpl* erforderlich (siehe Abbildung 5.12), welche in den jeweiligen generierten Graphen-, Kanten- und Knotenklassen implementiert werden muss. Diese Methode liefert eine Menge zurück, welche Referenzen auf alle versionerten Attribute der jeweiligen Instanz enthält.

Mit dem Feld *temporaryVersionCounter* vom Typ *long* wird der temporäre Versionenzähler für jede Transaktion T_i umgesetzt. Zudem speichert jede Transaktion T_i im Feld *persistentVersionAtBot* den zum jeweiligen *BOT*-Zeitpunkt gültigen Wert von *persistentVersionCounter*.

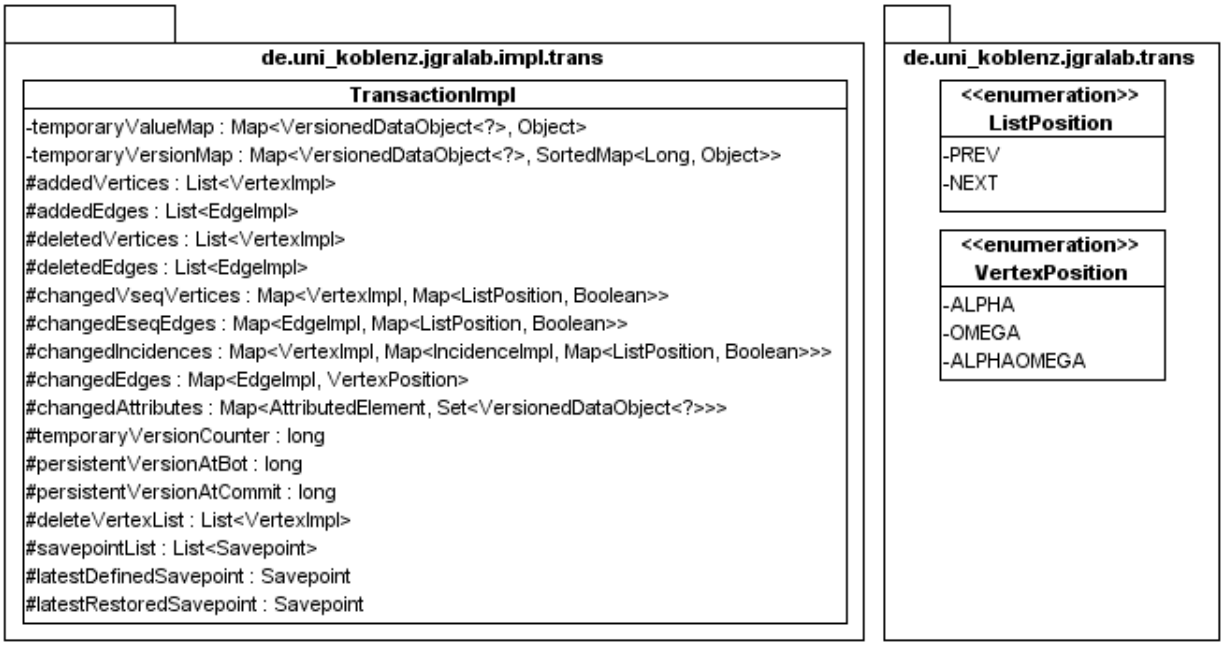


Abbildung 5.13: Die Klasse *TransactionImpl*

5.2.14 Korrektes Ausführen der Schreibphase

Im Feld *persistentVersionAtCommit* wird der Wert von *persistentVersionCounter* vor der Durchführung der Schreibphase durch eine Transaktion T_i gespeichert. So lässt sich für jedes durch T_i geänderte Datenobjekt *x* feststellen, ob eine neue persistente Version x_p erzeugt werden muss (x.*getLatestPersistentVersion()* <= T_i.*persistentVersionAtCommit*). Wurde während der Schreibphase bereits eine neue persistente Version x_p erstellt (gilt also x.*getLatestPersistentVersion()* > T_i.*persistentVersionAtCommit*), werden weitere persistente Änderungen für *x* auf (der schon vorhandenen Version) x_p durchgeführt.

5.2.15 Korrekte Ausführung des Löschens von Knoten

Um das Löschen von Knoten für jede Transaktion T_i korrekt ausführen zu können (siehe Abschnitt 5.1.2, Seite 115), besitzt die Klasse *TransactionImpl* die Liste *deleteVertexList* (siehe Abbildung 5.13). Diese enthält die zu löschenden Knoteninstanzen bei Löschoperationen einer einzelnen Knoteninstanz oder einer Kompositionskante.

5.2.16 Umsetzung von Sicherungspunkten

Jede Transaktion T_i hält durch die Liste *savepointList* in der Klasse *TransactionImpl* Referenzen auf die für T_i gültigen Sicherungspunkte. Im Feld *lastDefinedSavepoint* wird eine Referenz auf den zuletzt für T_i definierten Sicherungspunkt und im Feld *lastRestoredSavepoint* eine Referenz auf den zuletzt wiederhergestellten Sicherungspunkt gespeichert (siehe Abbildung 5.13).

Die Umsetzung eines Sicherungspunkts wird in Abbildung 5.14 dargestellt. Die Klasse *SavepointImpl* enthält (flache) Kopien der in der zugehörigen Transaktion mitgeführten Mengen zur Konfliktüberprüfung, die jeweils zum Initialisierungszeitpunkt des Sicherungspunkts gültig waren.

Im Feld *versionAtSavepoint* merkt sich ein Sicherungspunkt den zum Initialisierungszeitpunkt gültigen Wert von T_i.*temporaryVersionCounter* der zugehörigen Transaktion T_i.

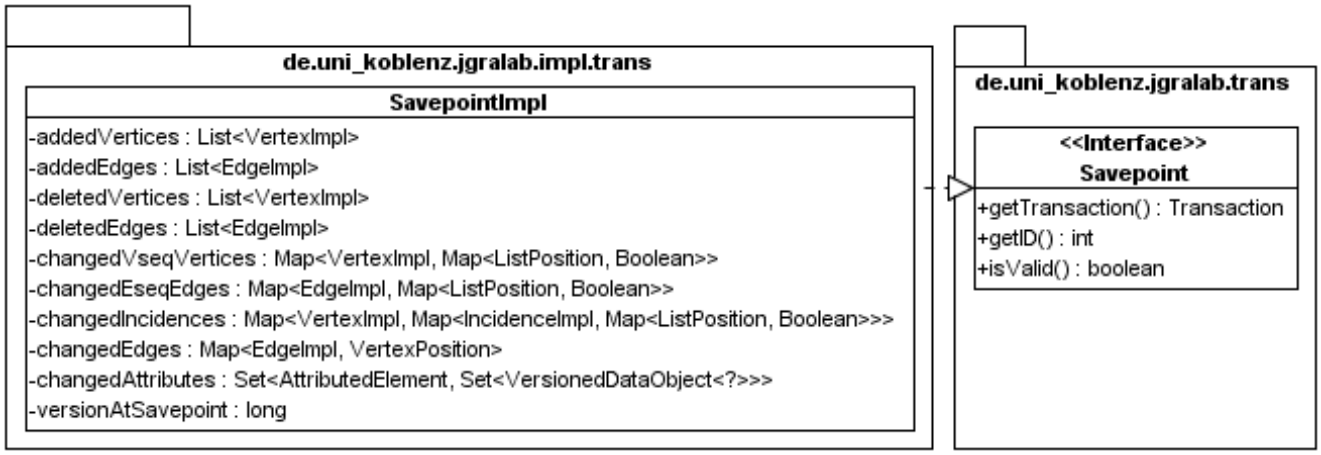


Abbildung 5.14: Die Klasse *SavepointImpl*

Das Interface *Savepoint* ist für die Anwender von JGraLab zur Repräsentation eines Sicherungspunkts relevant. Es definiert die folgenden Methoden:

- Mit der Methode **getTransaction()** kann für einen Sicherungspunkt die zugehörige Transaktion ermittelt werden.

- Die Methode **getID()** erlaubt die Abfrage der *ID* des Sicherungspunkts.

- Die Methode **isValid()** signalisiert, ob ein Sicherungspunkt noch gültig ist. Ein Sicherungspunkt *s* ist nicht (mehr) gültig, falls *s.getTransaction().savepointList.contains(s) = false* gilt.

6 Speicherplatz- und Zeitmessungen

In diesem Kapitel werden Speicherplatz- und Zeitmessungen erläutert, die (unter identischen Bedingungen) zum Vergleich auf Graphen vom Typ *MotorwayMap* (siehe Abbildung 2.3, Seite 19) *ohne* und *mit* Transaktionsunterstützung ausgeführt werden.

6.1 Rahmenbedingungen

Zur Messung des Speicherplatzverbrauchs (**MEM**) wird *VisualVM*[6] in der Version 1.1 verwendet. Mit *VisualVM* kann unter anderem die CPU-Belastung und der Speicherplatzverbrauch (benutzter *Heap*-Speicher) eines Java-Prozesses analysiert werden. Der Speicherplatz wird durchgehend in Megabyte (MB) angegeben.

Zur Messung der Laufzeit (**RTIME**) werden Instanzen der Klasse *ProgressFunctionImpl* aus dem Paket *de.uni_koblenz.jgralab.impl* verwendet, die in den Messungen als Timer fungieren. Die Laufzeit wird durchgehend in Sekunden (s) angegeben.

Jede Messung wird dreimal durchgeführt, um die auftretenden Abweichungen bei den Testdurchläufen - vor allem bei den Zeitmessungen (≈ 3 %) - zu berücksichtigen. Als Ergebnisse werden jeweils der Durchschnitt der drei Durchgänge übernommen. Die Messergebnisse werden in Tabellen- und zum Teil auch in Diagrammform dargestellt. Die Tests für Graphen ohne Transaktionsunterstützung werden mit dem Bezeichner **G-STD**, die Tests für Graphen mit Transaktionsunterstützung mit dem Bezeichner **G-TRANS** gekennzeichnet. Wenn nicht anders angegeben, werden die Messungen in *G-TRANS* im Rahmen einer (Read-Write-)Transaktion durchgeführt.

Das Testsystem besitzt die folgende Konfiguration:

- Intel Core2 Duo CPU mit 2,20 GHz,
- 2 GB RAM,
- Windows XP Service Pack 3 und
- JVM 1.6.0_07 mit einer initialen JVM-Heap-Größe von 1500 MB.

[6] `https://visualvm.dev.java.net/` (abgerufen am 4. März 2009)

6.2 Erzeugen von Graphenelementen

In den ersten Tests werden ansteigende Mengen von Graphenelementen erzeugt, wobei jeweils das Verhältnis *2 Knoten : 3 Kanten* gilt. Die Messergebnisse sind in Tabelle 6.1 und in Tabelle 6.2 dargestellt. Bei *G-TRANS* werden die folgenden Messgrößen (*Measure*) betrachtet:

- *MEM-r* gibt den Speicherplatzverbrauch vor dem COMMIT der Transaktion an,
- *MEM-c* gibt den Speicherplatzverbrauch nach dem COMMIT der Transaktion an,
- *RTIME-r* gibt die Laufzeit der Transaktion bis zum COMMIT an,
- *RTIME-c* gibt die Laufzeit des COMMITs der Transaktion an und
- *RTIME-t* entspricht der Gesamtlaufzeit der Transaktion (also der Summe von *RTIME-r* und *RTIME-c*).

Ab der 3. Spalte wird in der ersten Zeile die Anzahl der in den einzelnen Testläufen jeweils erzeugten Graphenelemente notiert.

	Measure	*1000*	*2500*	*5000*	*10000*	*25000*	*50000*	*100000*
G-STD	*MEM (MB)*	0,788	0,933	1,201	1,320	2,477	4,450	8,510
	RTIME (s)	0,015	0,032	0,047	0,062	0,110	0,188	0,375
G-TRANS	*MEM-r (MB)*	0,842	1,561	2,753	4,972	10,540	20,924	40,971
	MEM-c (MB)	0,832	1,117	1,644	2,774	6,951	13,322	25,432
	RTIME-r (s)	0,797	0,969	1,187	1,500	2,640	4,328	8,828
	RTIME-c (s)	0,156	0,250	0,406	0,812	1,938	3,813	6,765
	RTIME-t (s)	0,953	1,319	1,593	1,312	4,578	8,141	15,593

Tabelle 6.1: Erzeugen von Graphenelementen ohne und mit Transaktionsunterstützung - Teil 1

	Measure	*250000*	*500000*	*1000000*	*1500000*	*2000000*
G-STD	*MEM (MB)*	20,171	42,421	84,215	127,067	164,248
	RTIME (s)	0,954	1,718	3,156	4,937	7,312
G-TRANS	*MEM-r (MB)*	101,209	202,761	401,599	607,831	n.a.
	MEM-c (MB)	63,726	128,256	214,412	349,791	n.a.
	RTIME-r (s)	25,016	47,250	95,438	161,531	n.a.
	RTIME-c (s)	17,000	34,609	69,890	143,938	n.a.
	RTIME-t (s)	42,016	81,859	165,328	305,469	n.a.

Tabelle 6.2: Erzeugen von Graphenelementen ohne und mit Transaktionsunterstützung - Teil 2

6.2.1 Speicherplatzverbrauch

Sowohl Speicherplatzverbrauch als auch Laufzeitverhalten sind bei *G-TRANS* erwartungsgemäß deutlich höher bzw. schlechter als bei *G-STD*. Zur besseren Übersicht wird zunächst der Speicherplatzverbrauch in Abbildung 6.1 in Diagrammform dargestellt. Hier wird bei *G-TRANS* der Speicherplatzverbrauch nach dem *COMMIT* (*MEM-c*) betrachtet, um einen brauchbaren Vergleich zu liefern.

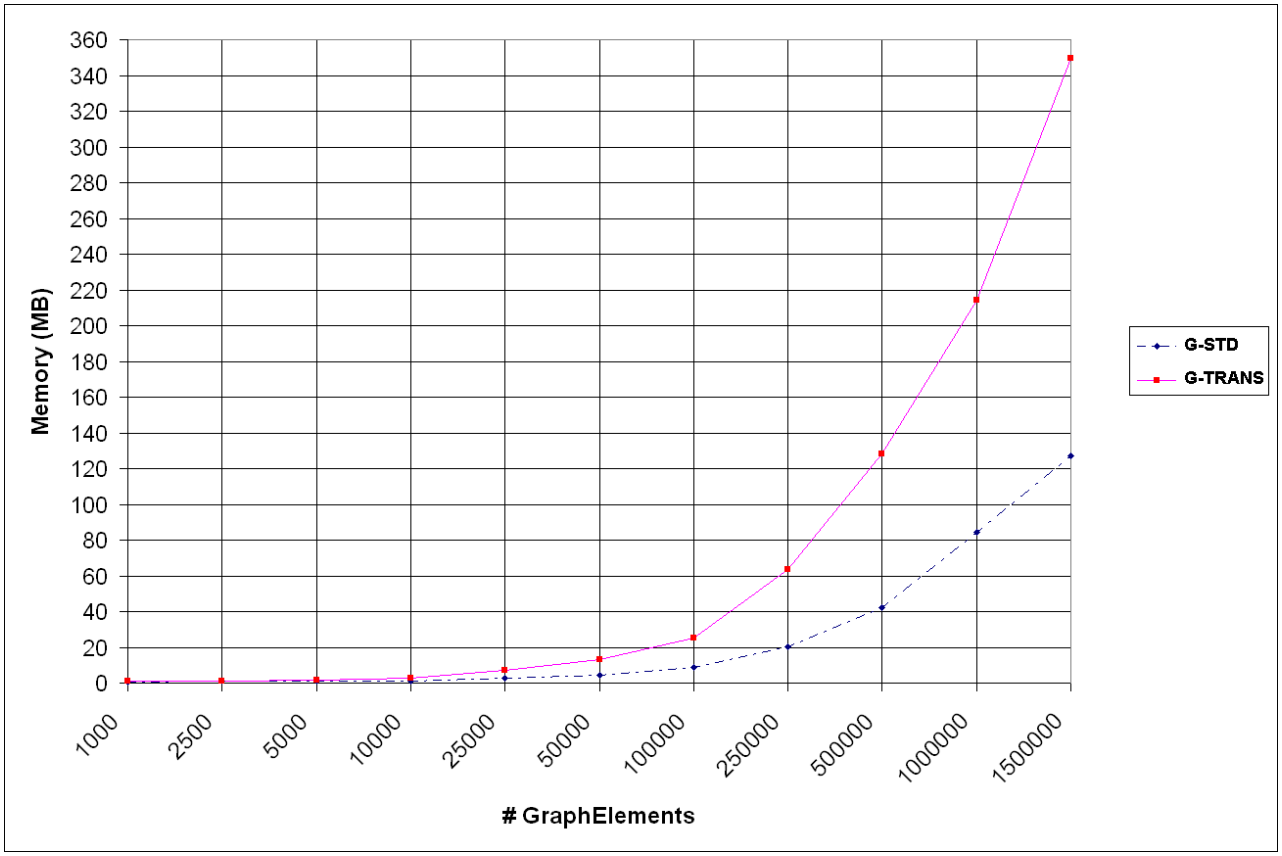


Abbildung 6.1: Speicherplatzverbrauch ohne und mit Transaktionsunterstützung

Der erhöhte Speicherplatzverbrauch bei *G-TRANS* lässt sich vor allem durch die höhere Anzahl von Objekten erklären, die im Vergleich zu *G-STD* bei gleicher Anzahl von Graphenelementen erzeugt werden müssen. Abbildung 6.2 zeigt einen mit *VisualVM* erstellten *Heap-Dump*, der für *G-TRANS* bei 250000 erzeugten Graphenelementen gilt. Insgesamt existieren hier 1500004 Instanzen der Klasse *VersionedReferenceImpl*, die jeweils mit einer Objektgröße von *24 Bytes* insgesamt ≈ 36 MB (64,4 % des gesamten Speicherplatzverbrauchs) beanspruchen.

Class Name	Instances [%]	Instances	Size
de.uni_koblenz.jgralab.impl.trans.**VersionedReferenceImpl**		1500004 (78%)	36000096 (64,4%)
de.uni_koblenz.jgralabtest.schemas.motorwaymap.impl.trans.**ExitImpl**		150000 (7,8%)	6600000 (11,8%)
de.uni_koblenz.jgralabtest.schemas.motorwaymap.impl.trans.**ReversedExitImpl**		150000 (7,8%)	4200000 (7,5%)
de.uni_koblenz.jgralabtest.schemas.motorwaymap.impl.trans.**CityImpl**		50000 (2,6%)	3000000 (5,4%)
de.uni_koblenz.jgralabtest.schemas.motorwaymap.impl.trans.**MotorwayImpl**		50000 (2,6%)	2200000 (3,9%)
de.uni_koblenz.jgralab.impl.trans.**EdgeImpl[]**		1 (0%)	1048588 (1,9%)
de.uni_koblenz.jgralab.impl.trans.**ReversedEdgeImpl[]**		1 (0%)	1048588 (1,9%)
de.uni_koblenz.jgralab.impl.trans.**VertexImpl[]**		1 (0%)	524300 (0,9%)

Abbildung 6.2: Heap-Dump für G-TRANS bei 250000 erzeugten Graphenelementen

In Tabelle 6.2 wird für *G-TRANS* bei einer Anzahl von 2000000 Graphenelementen *n.a.* (not available) angegeben. Da der Speicherplatzverbrauch vor dem *COMMIT* der Transaktion in etwa doppelt so groß ist wie nach dem *COMMIT*, ist hier der reservierte JVM-Heap-Speicher (1500 MB) nicht ausreichend, wodurch eine *OutOfMemoryException* von der JVM geworfen wird.

6.2.2 Laufzeitverhalten

Abbildung 6.3 stellt die Messergebnisse des Laufzeitverhaltens in Diagrammform dar. Für *G-TRANS* wird dabei die Laufzeit bis zum *COMMIT* (*RTIME-r*) abgebildet, da beim Vergleich nur die Laufzeiten für das eigentliche Erzeugen der Graphenelemente relevant sind. Die steigenden Laufzeitdifferenzen bei erhöhter Anzahl von Graphenelementen lassen sich vor allem durch die hohe Anzahl von zu instanziierbaren Objekten für die Versionierung in *G-TRANS* erklären.

6.2.3 Messergebnisse bei parallel laufenden Transaktionen

Auch das parallele Ausführen mehrerer Transaktionen führt auf dem Testsystem nicht zu einem Performanzgewinn. Tabelle 6.3 zeigt die durchschnittliche Laufzeit einer Transaktion bis zum *COMMIT* (∅ *RTIME-r*) und die Gesamtlaufzeit (*RTIME-t*), um insgesamt 100000 Graphenelemente zu erstellen. Getestet werden 2, 4, 8, 16 und 32 parallele Transaktionen, wobei jede jeweils 50000, 25000, 12500, 6250 bzw. 3125 Graphenelemente erzeugt.

	Measure	2	4	8	16	32
G-TRANS	∅ RTIME-r (s)	10,101	10,588	11,741	10,375	10,891
	RTIME-t (s)	29,938	28,890	28,578	32,063	43,234

Tabelle 6.3: Parallele Ausführung von Transaktionen bei 100000 Graphenelementen

Da die Transaktionen zwar in unterschiedlichen Threads, jedoch auf dem gleichen Prozessorkern arbeiten, verschlechtern sich jeweils die Laufzeiten bis zum *COMMIT* im Vergleich zu den Ergebnissen in Tabelle 6.1. Da zu einem Zeitpunkt nur eine Transaktion ihr *COMMIT* durchführen kann, erhöht sich zusätzlich die Gesamtlaufzeit.

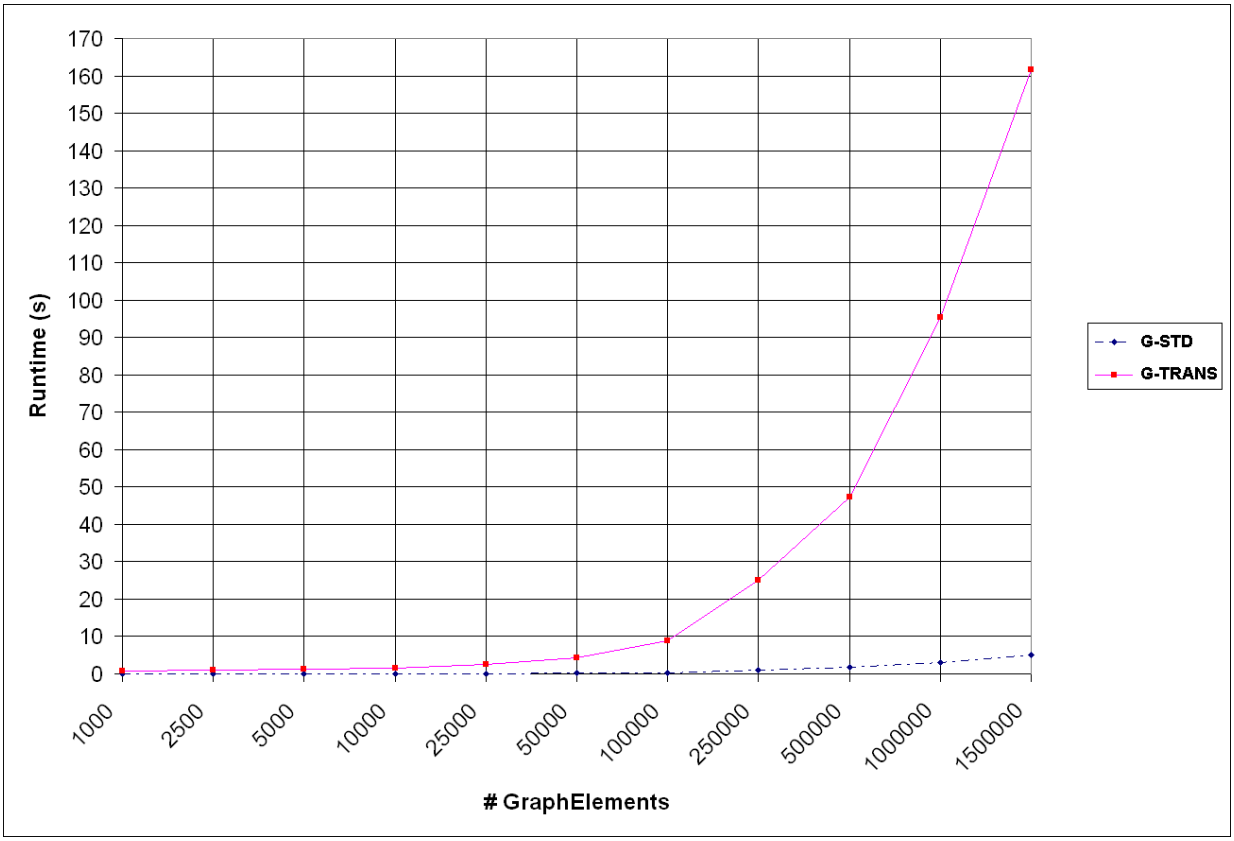


Abbildung 6.3: Laufzeit ohne und mit Transaktionsunterstützung

6.3 Laden von Graphen

Das Laden von Graphen ohne und mit Transaktionsunterstützung ist Bestandteil der zweiten Testläufe. Die Messergebnisse werden in Tabelle 6.4 und Tabelle 6.5 dargestellt. Hierbei werden Graphen mit jeweils aufsteigender Anzahl von Graphenelementen geladen. Da in *G-TRANS* keine Transaktion für das Laden von Graphen benötigt wird, sind auch Messergebnisse (des Speicherplatzverbrauchs) für Graphen mit 2000000 und 5000000 Graphenelementen möglich.

	Measure	1000	2500	5000	10000	25000	50000	100000
G-STD	MEM (MB)	1,046	1,041	1,308	1,824	2,580	5,173	9,428
	RTIME (s)	0,125	0,110	0,172	0,235	0,344	0,547	0,984
G-TRANS	MEM (MB)	0,768	1,268	1,741	3,186	7,157	13,940	27,174
	RTIME (s)	0,125	0,188	0,250	0,344	0,578	1,000	1,703

Tabelle 6.4: Laden von Graphen ohne und mit Transaktionsunterstützung - Teil 1

	Measure	250000	500000	1000000	1500000	2000000	5000000
G-STD	MEM (MB)	25,070	50,002	95,797	144,698	177,215	441,505
	RTIME (s)	2,219	4,250	8,297	13,172	18,109	44,000
G-TRANS	MEM (MB)	69,518	140,449	273,500	407,755	536,435	1345,604
	RTIME (s)	3,985	8,312	16,391	24,203	31,922	78,328

Tabelle 6.5: Laden von Graphen ohne und mit Transaktionsunterstützung - Teil 2

6.3.1 Laufzeitverhalten

Für das Laden von Graphen wird lediglich das Laufzeitverhalten näher betrachtet, da der Speicherplatzverbrauch bereits in Abschnitt 6.2 erläutert wurde. Abbildung 6.4 stellt vergleichend das Laufzeitverhalten beim Laden von Graphen in Diagrammform dar. Auch hier führt die höhere Anzahl von zu instanziierbaren Objekten in *G-TRANS* zu deutlich längeren Ladezeiten als in *G-STD* (vor allem ab 500000 Graphenelementen).

6.4 Speichern von Graphen

In den letzten Testdurchläufen wird das Speichern von Graphen ohne und mit Transaktionsunterstützung untersucht. Da beim Speichern unter anderem *Vseq*, *Eseq* und Λ*seq(v)* eines jeden Knotens $v \in V$ durchlaufen werden, können auch Rückschlüsse auf die Zugriffszeiten bei Graphenelementen gezogen werden. In *G-TRANS* wird das Speichern im Rahmen einer *Read-Only-Transaktion* durchgeführt. Die Messergebnisse werden in Tabelle 6.6 und in Tabelle 6.7 dargestellt.

	Measure	1000	2500	5000	10000	25000	50000	100000
G-STD	RTIME (s)	0,032	0,047	0,062	0,094	0,171	0,328	0,531
G-TRANS	RTIME (s)	0,047	0,063	0,109	0,188	0,343	0,625	1,203

Tabelle 6.6: Speichern von Graphen ohne und mit Transaktionsunterstützung - Teil 1

	Measure	250000	500000	1000000	1500000
G-STD	RTIME (s)	1,438	2,610	4,297	5,218
G-TRANS	RTIME (s)	2,953	5,953	11,750	17,766

Tabelle 6.7: Speichern von Graphen ohne und mit Transaktionsunterstützung - Teil 2

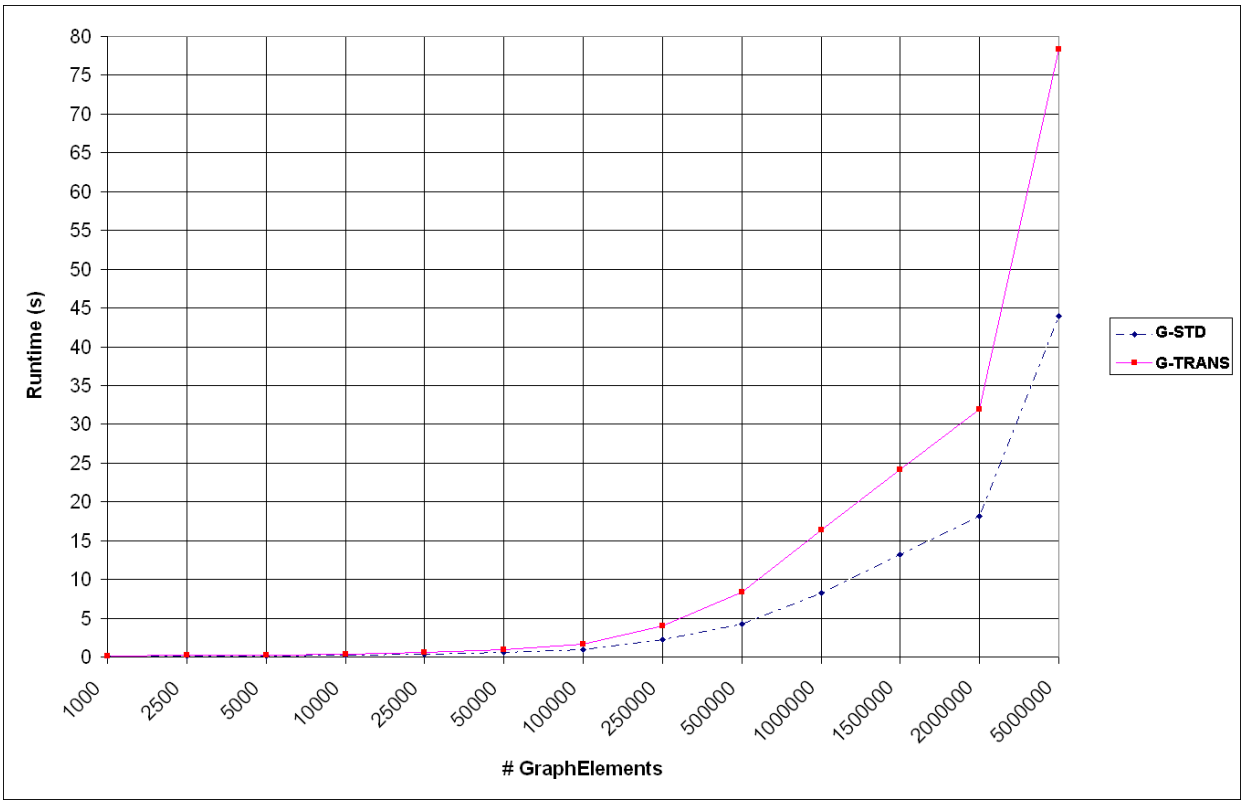


Abbildung 6.4: Laufzeit beim Laden ohne und mit Transaktionsunterstützung

6.4.1 Laufzeitverhalten

Abbildung 6.5 zeigt vergleichend das Laufzeitverhalten beim Speichern von Graphen in Diagrammform. Die schlechteren Zugriffszeiten bei *G-TRANS* sind bedingt durch die Versionierung der Felder in den einzelnen Graphenelementen. Während in *G-STD* beim Zugriff auf ein Feld innerhalb eines Graphenelements unmittelbar der entsprechende Wert zurückgeliefert werden kann, ist in *G-TRANS* für jedes (versionierte) Feld ein weiterer Zugriff auf ein Objekt der Klasse *VersionedDataObjectImpl* notwendig.

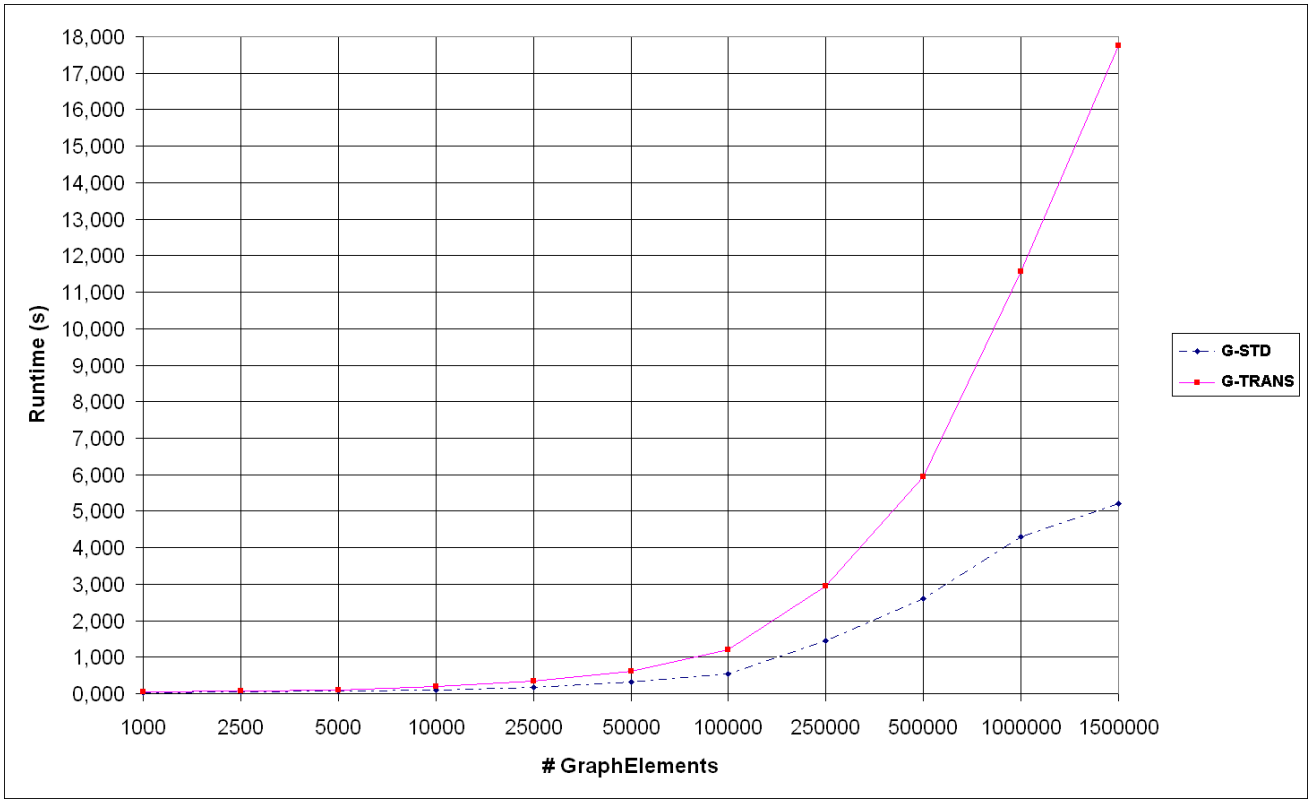


Abbildung 6.5: Laufzeit beim Speichern ohne und mit Transaktionsunterstützung

6.5 Bewertung

Durch die Verwaltung von temporären und persistenten Versionen für jedes zu versionierende Datenobjekt entsteht ein offensichtlicher erhöhter Speicherplatzbedarf bei der Repräsentation eines TGraphen im JGraLab. Um den Speicherzuwachs so gering wie möglich zu halten, wurde bereits bei der Modellierung eine möglichst effiziente Umsetzung der Versionierung angestrebt (siehe Abschnitt 5.2.12.3, Seite 145). Durch die Optimierungen sind Anpassungen an dem in Kapitel 4 beschriebenem formalem Modell des Transaktionskonzepts vorgenommen worden, wobei die korrekte Umsetzung des Verfahrens beibehalten wurde.

Die in diesem Kapitel durchgeführten Messungen zeigen, dass die Transaktionsunterstützung einerseits zu einem erhöhten Speicherplatzbedarf und andererseits zu schlechteren Laufzeitbedingungen führt. Dies lässt sich vor allem durch die erhöhte Objektanzahl erklären, welche durch die Versionierung bei der Transaktionsunterstützung entsteht. Da (Read-Write-)Transaktionen während ihrer Lesephase zudem temporäre Versionen für versionierte Datenobjekte anlegen müssen, steigt hier zusätzlich der Speicherplatzbedarf.

Die parallele Ausführung von Transaktionen (vor allem bei Graphen mit einer hohen Anzahl von Graphenelementen) kann dadurch zu (Speicherplatz-)Problemen führen.

Die Messergebnisse zeigen aber auch, dass sich für Graphen in der Größenordnung bis 250000 Graphenelemente akzeptable Performanzwerte ergeben. Da die Transaktionsunterstützung hauptsächlich in Anwendungen zum Einsatz kommen soll, in denen auf Graphen mit einer relativ geringen Anzahl von Graphenelementen (≤ 25000) gearbeitet wird, ist eine praktische Nutzung des Transaktionskonzepts hier problemlos möglich.

7 Fazit und Ausblick

In diesem Kapitel werden zunächst die in dieser Diplomarbeit ausgearbeiteten Ergebnisse zusammengefasst (siehe Abschnitt 7.1). Abschnitt 7.2 erläutert, welche der in Abschnitt 5.2 (Seite 123) angesprochenen Implementationsaspekte im Rahmen der Diplomarbeit umgesetzt worden sind. Abschließend wird ein Überblick über noch ausstehende oder zu bearbeitende Frage- und Problemstellungen im Kontext des entwickelten Transaktionskonzepts geliefert (siehe Abschnitt 7.3).

7.1 Zusammenfassung

Das im Rahmen der Diplomarbeit entwickelte Transaktionskonzept für das Graphenlabor *JGraLab* basiert vor allem auf Mechanismen der *optimistischen Synchronisation* (siehe Abschnitt 3.6.6, Seite 52) und der *Mehrversionen-Synchronisation* (siehe Abschnitt 3.6.7, Seite 55).

So durchläuft jede (Read-Write-)Transaktion T_i drei Phasen: die *Lesephase* (siehe Abschnitt 4.2.3.6, Seite 90), die *Validierungsphase* (siehe Abschnitt 4.2.3.7, Seite 91) und die *Schreibphase* (siehe Abschnitt 4.2.3.12, Seite 103). In der Lesephase führt T_i ihre Änderungen isoliert von allen anderen parallel laufenden Transaktionen T_j zunächst auf einem lokalen Puffer aus. Nach dem *COMMIT* von T_i erfolgt in der Validierungsphase die Überprüfung auf mögliche Konflikte (hauptsächlich *Lost Updates* und *Phantomprobleme*), welche durch die in T_i ausgeführten Änderungen aufgetreten sein können. Sind keine Konflikte für T_i feststellbar, kann T_i in die Schreibphase übergehen. In dieser werden die in T_i durchgeführten Änderungen persistent im jeweiligen Graphen übernommen.

Um die Isolation einer (Read-Write-)Transaktion T_i während ihrer Lesephase zu garantieren und den dafür notwendigen lokalen Puffer umsetzen zu können, wird die optimistische Synchronisation mit der Mehrversionen-Synchronisation kombiniert. So werden alle im JGraLab identifizierten Datenobjekte versioniert (siehe Abschnitt 4.2.3.3, Seite 87). Für jedes Datenobjekt *x* existieren *temporäre* und *persistente* Versionen (siehe Abschnitt 4.2.3.1, Seite 85). Temporäre Versionen entstehen durch die von einer Transaktion T_i durchgeführten Änderungen in der Lesephase. Persistente Versionen entstehen in der Schreibphase einer Transaktion T_i.

Desweiteren können für eine Transaktion T_i Sicherungspunkte während ihrer Lesephase definiert werden, welche sich den aktuellen (temporären) Graphenzustand für T_i merken (siehe Abschnitt 4.3.3, Seite 108). T_i kann während ihrer Lesephase auf einen definierten Sicherungspunkt *s* zurückgesetzt werden, so dass für T_i der mit *s* verbundene Graphenzustand wiederhergestellt wird.

Jede Transaktion T_i im JGraLab erfüllt die folgenden *ACID*-Eigenschaften (siehe Abschnitt 3.1, Seite 27):

- *Atomicity* (Atomarität): Die in T_i jeweils durchgeführten Änderungen werden nach dessen *COMMIT vollständig* persistent gespeichert, falls keine Konflikte in der Validierungsphase festgestellt werden konnten. Treten Konflikte auf, wird *keine* Änderung von T_i persistent übernommen.

- *Isolation*: Jede Transaktion T_i hat während ihrer Lesephase für jedes Datenobjekt *x* lediglich Zugriff auf die jeweilige persistente Version x_p, die zum *BOT*-Zeitpunkt von T_i gültig war und auf die für T_i erzeugten temporären Versionen x_t. Temporäre und persistente Änderungen und neu hinzugefügte Datenobjekte anderer parallel laufender Transaktionen T_j sind für T_i nicht sichtbar.

- *Durability* (Dauerhaftigkeit oder Persistenz): Persistente Änderungen nach dem *COMMIT* einer Transaktion T_i bleiben während dem Betrieb des Systems dauerhaft im Hauptspeicher erhalten.

Die Eigenschaft *Consistency* (Konsistenz) wird von Transaktionen im JGraLab nicht unterstützt.

7.2 Umsetzung des Transaktionskonzepts

Alle in Abschnitt 5.2 (Seite 123) angesprochenen Implementationsaspekte sind im Rahmen dieser Diplomarbeit umgesetzt worden. Dazu gehören unter anderem:

- Das Erstellen und Laden von Graphen mit und ohne Transaktionsunterstützung,

- die Generierung der objektorientierten Zugriffsschicht für Graphen mit Transaktionsunterstützung,

- die Umsetzung von Transaktionen mit Lesephase, Validierungsphase und Schreibphase,

- die Verwaltung von Transaktionen durch den Transaktionsmanager,

- die Umsetzung der Versionierung (Verwaltung von temporären und persistenten Versionen der Datenobjekte),

- das Definieren von Sicherungspunkten und das Zurücksetzen einer Transaktion auf einen definierten Sicherungspunkt und

- die *Garbage Collection* nach dem *COMMIT*, *ABORT* und dem Zurücksetzen einer Transaktion auf einen Sicherungspunkt.

Die Umsetzung des Transaktionskonzepts wurde mit bereits vorhandenen und eigens für das Transaktionskonzept in *JUnit*[7] erstellten *TestSuites* ausführlich getestet.

7.3 Ausblick

Folgende Frage- und Problemstellungen sind im Kontext des Transaktionskonzepts für zukünftige Bearbeitungen denkbar:

- Beim *COMMIT* wäre zusätzlich zu der Konfliktüberprüfung in der Validierungsphase auch eine Überprüfung auf Constraintverletzungen (beispielsweise das Missachten von Kardinalitäten in einem definierten Schema) denkbar.

- Nachdem im Rahmen des *COMMITs* einer Transaktion T_i ein Konflikt festgestellt wird, erfolgt der Abbruch des *COMMIT*-Versuchs und das Zurücksetzen von T_i in den Zustand *RUNNING*. Ein mögliche Erweiterung wäre eine (semi-)automatisierte Konfliktbehebung, welche nach Erkennung eines Konflikts die Transaktion T_i in einen konfliktfreien Zustand überführt.

Die in Kapitel 6 (Seite 163) durchgeführten Performanzmessungen zeigen, dass das umgesetzte Transaktionskonzept für die meisten Anwendungen, die auf Graphen mit einer relativ geringen Anzahl von Graphenelementen (≤ 250000) arbeiten, absolut geeignet ist. Die praktische Nutzung des Transaktionskonzepts im Rahmen von Projekten innerhalb des Instituts für Softwaretechnik ist geplant.

[7] `http://www.junit.org/` (abgerufen am 4. März 2009)

Literaturverzeichnis

[BER08] Bildhauer, Daniel ; Ebert, Jürgen ; Riediger, Volker ; Schwarz, Hannes ; Strauss, Sascha: *grUML - Eine UML-basierte Modellierungssprache für TGraphen (Version 0.92 vom 24.01.08)*. Institut für Softwaretechnik, Universität Koblenz-Landau, 2008

[DEL94] Dahm, Peter ; Ebert, Jürgen ; Litauer, Christoph: *Benutzerhandbuch EMS-Graphenlabor V3.0*. 1994. – `http://www.uni-koblenz.de/~ist/documents/Dahm1994DE3.pdf` (abgerufen am 4. März 2009)

[DW98] Dahm, Peter ; Widmann, Friedbert: *Das Graphenlabor*. Institut für Informatik, Universität Koblenz-Landau, 1998

[FFS04] Freeman, Eric ; Freeman, Elisabeth ; Sierra, Kathy ; Bates, Bert: *Head First Design Patterns*. 1. Auflage. O'Reilly, 2004

[GR94] Gray, J. ; Reuter, A.: *Transaction Processing: Concepts and Techniques*. 1. Auflage. Morgan Kaufmann, 1994

[HR99] Härder, T. ; Rahm, E.: *Datenbanksysteme - Konzepte und Techniken der Implementierung*. 1. Auflage. Springer, 1999

[Kah06] Kahle, Steffen: *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*. Diplomarbeit, Institut für Softwaretechnik, Informatik, Fachbereich 4, Universität Koblenz-Landau, Juni 2006

[KE04] Kemper, A. ; Eickler, A.: *Datenbanksysteme - Eine Einführung*. 5. Auflage. Oldenbourg, 2004

[Krü07] Krüger, G.: *Handbuch der Java-Programmierung*. 4. Auflage. Addison-Wesley, 2007

[KW06] Kemper, A. ; Wimmer, M.: *Übungsbuch: Datenbanksysteme*. 1. Auflage. Oldenbourg, 2006

[Mar06] MARCHEWKA, Katrin: *GReQL 2*. Diplomarbeit, Institut für Softwaretechnik, Informatik, Fachbereich 4, Universität Koblenz-Landau, 2006

[RHQ05] RUPP, Chris ; HAHN, Jürgen ; QUEINS, Stefan ; JECKLE, Mario ; ZENGLER, Barbara: *UML 2 glasklar*. 2. Auflage. Hanser, 2005

[Sed02] SEDGEWICK, Robert: *Algorithmen*. 2. Auflage. Addison-Wesley, 2002

[Ste06] STEFFENS, T.: *Kontextfreie Suche auf Graphen*. Diplomarbeit, Institut für Softwaretechnik, Informatik, Fachbereich 4, Universität Koblenz-Landau, Januar 2006

[Ull07] ULLENBOOM, C.: *Java ist auch eine Insel*. 6. Auflage. Galileo Computing, 2007

[Vos00] VOSSEN, G.: *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme*. 4. Auflage. Oldenbourg, 2000

[Wei88] WEIKUM, G.: *Transaktionen in Datenbanksystemen*. 1. Auflage. Addison-Wesley, 1988

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis