Institute for Software Technology

UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Entwicklung eines TGraph-basierten Modell-Vergleichsverfahren

Diplomarbeit

zur Erlangung des Grades eines
Diplom-Informatikers
im Studiengang Informatik

vorgelegt von

Matthias Sattel

Betreuer: Professor Dr. Jürgen Ebert, Institut für Softwaretechnik, Fachbereich 4

Dr. Volker Riediger, Institut für Softwaretechnik, Fachbereich 4

Dipl.-Inform. Tassilo Horn, Institut für Softwaretechnik, Fachbereich 4

Koblenz, im Juli 2010

Zusammenfassung

Im Rahmen dieser Diplomarbeit wird ein TGraph-basiertes Modell-Vergleichsverfahren entwickelt. Dieses Verfahren soll eine anwendbare Berechnung von Deltas für TGraph-basierte Modelle ermöglichen.

Modelle können durch TGraphen repräsentiert werden. Für die Erzeugung und Verarbeitung von TGraphen wurde am Institut für Softwaretechnik das **Java Graph Laboratory**, kurz JGraLab, entwickelt. Es handelt sich dabei um eine Java Klassenbibliothek, die eine hoch effiziente API zur Verarbeitung von TGraphen bereitstellt. [Ebe87, ERW08, ERSB08]

Basierend auf der JGraLab API wird ein System entwickelt, das unterschiedliche Werkzeuge zur Verwaltung von Modellen bereitstellt. Der Fokus dieser Diplomarbeit liegt dabei auf der Entwicklung eines Werkzeugs, das zu zwei gegebenen TGraphen ein Delta berechnet.

Abstract

Within this diploma thesis a TGraph-based process for Model-comparison is developed. This process should enable an applicable computation of Deltas for TGraph-based models.

Models can be represented using TGraphs. The **Java Graph Laboratory**, in short JGraLab, was developed at the Intsitute for Software Technology. It is a java class library that offers a highly efficient API for processing TGraphs.

Based on the JGraLab API a system is developed, that procures multiple tools for the purpose of Model-maintenance. The focus of this diploma thesis is on the developement of a tool, that computes a delta between two TGraphs.

Erklärung

Hiermit erkläre 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 verwendet habe.

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

Ort, Datum

Unterschrift

INHALTSVERZEICHNIS

KAPITEL 1

EINFÜHRUNG

Die Entwicklung von Software wird geprägt durch den vermehrten Einsatz von Modellen. Diese **Modelle unterliegen Veränderungen** durch den Entwicklungsprozess. Sie werden verfeinert, vereinfacht und variiert. Ein Modell liegt, über den gesamten Lebenszyklus einer Software betrachtet, in verschiedenen **Versionen und Varianten** vor.

Die Softwareentwicklung wird durch bewährte Techniken und Werkzeuge zur Verwaltung von textuellen Artefakten unterstützt. Diese können nicht adäquat auf Modelle angewandt werden [FW07, EKS09]. **Der Vergleich und die Versionierung von Modellen** sind aktuelle Forschungsschwerpunkte der Softwareentwicklung. Aktuelle Forschungsergebnisse werden auf Konferenzen, wie dem ICSE[1] Workshop on Comparison and Versioning of Software Models[2][3], präsentiert und diskutiert.

Problemstellung

Die vorliegende Arbeit befasst sich mit der Entwicklung eines TGraph-basierten Modell-Vergleichsverfahrens. Dieses soll den Vergleich und die Versionierung von Modellen ermöglichen.

Für die Repräsentation von Modellen können **TGraphen** eingesetzt werden. Das am Institut für Softwaretechnik entwickelte **Java Graph Laboratory**, kurz JGraLab, ermöglicht die Erzeugung und Verarbeitung von TGraphen. Im Verlauf dieser Diplomarbeit wird ein System entwickelt, das JGraLab um Techniken und Werkzeuge zum Vergleich und zur Versionierung von TGraphen erweitert.

[1] `http://www.icse-conferences.org/`
[2] `http://pi.informatik.uni-siegen.de/CVSM2008/`
[3] `http://pi.informatik.uni-siegen.de/CVSM2009/`

Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in die folgenden Kapitel.

Kapitel 2 Terminologie In diesem Kapitel werden die grundlegenden Begriffe definiert. Zuerst werden allgemeine Begriffe erläutert und die Terminologie der TGraphen aufgearbeitet. Anschließend werden neue Begriffe, wie z.B. das Mapping, die Differenz von TGraphen und das TGraph-Delta, definiert und anhand von Beispielen erörtert.

Kapitel 3 Analyse und Definition der Aufgabestellung Ausgehend von der Zielsetzung der Diplomarbeit wurde eine Produktidee und ein Produktkonzept entwickelt, die in diesem Kapitel beschrieben werden. Darauf aufbauend werden die Produktfunktionen und die Anforderungen an das System betrachtet.

Kapitel 4 Entwurf und Spezifikation In diesem Kapitel wird die Architektur des Systems behandelt. Dazu zählt der Entwurf und die Spezifikation der wichtigsten Bausteine des Systems.

Kapitel 5 Implementierung Die Umsetzung des Entwurfs und der Spezifikation wird in diesem Kapitel behandelt. Es ermöglicht einen Einblick in die Implementierung. Die Möglichkeiten zur Erweiterung und Anpassung des Systems werden ebenfalls in diesem Kapitel erläutert. Dieses Kapitel ist besonders interessant für Entwickler, die das System anpassen oder erweitern möchten.

Kapitel 6 Anwendung und Integration In diesem Kapitel wird die Verwendung der API und der Programme beschrieben. Die API ermöglicht eine Integration in andere TGraph-basierte Anwendungen. Ein Entwickler erfährt, wie er die Werkzeuge des Systems in eigene Anwendungen integrieren kann. Zudem wird die Verwendung der Programme erörtert. Diese können von einem Anwender bspw. von der Konsole aus genutzt werden. Ihre Anwendung wird anhand von Beispielen aufgezeigt.

Kapitel 7 Evaluation und Test In diesem Kapitel erfolgt eine Beurteilung des entwickelten Systems. Die Erprobung der Werkzeuge und die Erfüllung der Anforderungen wird behandelt.

Kapitel 8 Fazit Am Ende der vorliegenden Diplomarbeit erfolgt ein kurzer Abriß der Arbeit. Probleme werden erörtert, Schwachstellen im Entwurf und der

Implementierung werden offengelegt. Zudem soll dieses Kapitel auf offene Problemstellungen eingehen und mögliche Weiterentwicklungen betrachten.

KAPITEL 2

TERMINOLOGIE

2.1. Einleitung

In diesem Kapitel werden alle notwendigen Begriffe definiert. Die Begriffe werden anhand von Beispielen erläutert. Dazu zählen einfache Beispiele und ein durchweg verwendetes Fallbeispiel.

Fallbeispiel

Bei dem Fallbeispiel handelt es sich um UML-Statechart Diagramme. Sie werden für die Modellierung von Softwaresystemen verwendet und dienen hier der Erläuterung der wichtigsten Begriffe. In einigen Fällen wird zur Veranschaulichung nicht auf das Fallbeispiel zurückgegriffen, da dieses in manchen Fällen zur Erläuterung der Begriffe zu komplex ist. Im weiteren Verlauf der Arbeit wird immer wieder auf dieses Fallbeispiel zurückgegriffen.

2.2. Artefakte, Versionen und Modelle

Vergleichsverfahren werden unabhängig von ihrem Einsatzgebiet dafür genutzt, um Objekte, die einen konkreten Sachverhalt beschreiben, miteinander zu vergleichen. Diese Objekte müssen in maschinenlesbarer Form vorliegen und werden als **Artefakte** bezeichnet.

Definition 1 *Artefakt*

Ein Artefakt ist ein Objekt (i.w.S.), das einen konkreten Sachverhalt beschreibt und maschinenlesbar ist.[Ebe06]

Artefakte entstehen in den unterschiedlichsten Anwendungsgebieten der Informatik. Die vorliegende Arbeit befasst sich in erster Linie mit solchen, die in

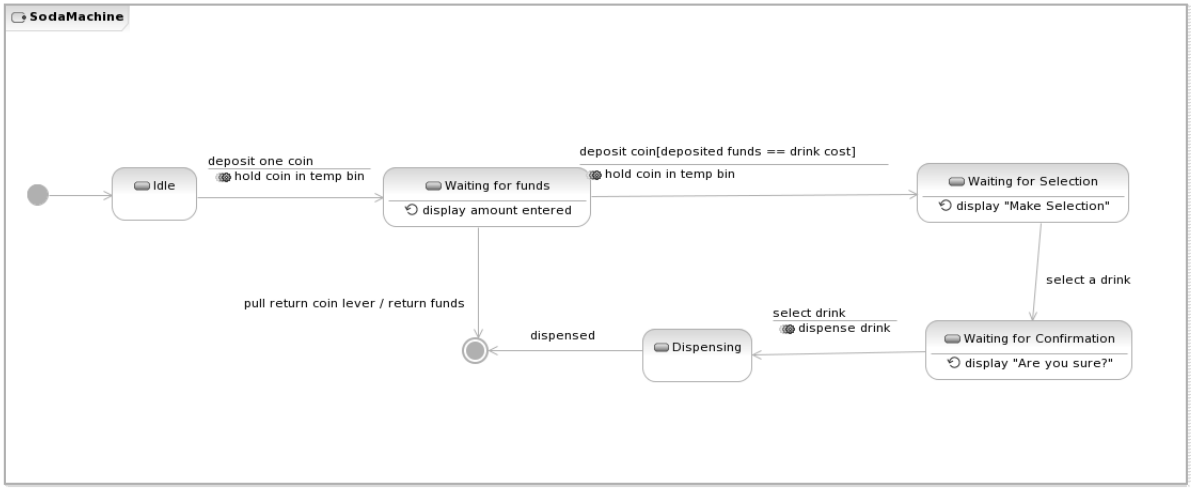


Abbildung 2.1.: Fallbeispiel: Eine Version eines UML Statechart-Diagramms

der Softwareentwicklung entstehen. Diese Artefakte unterliegen Veränderungen durch den Entwicklungsprozess. Zu unterschiedlichen Zeitpunkten können unterschiedliche Ausprägungen eines Artefakts existieren. Eine konkrete Ausprägung eines Artefakts wird als **Version** bezeichnet.

Definition 2 *Version*

Eine Version ist die konkrete Ausprägung eines Artefakts.[Ebe07]

Es ist üblich, dass ein Artefakt in verschiedenen Versionen vorliegt. Diese Versionen können im Laufe der Zeit durch Ersetzung einer bestehenden Version entstehen. In diesem Fall spricht man auch von **Revisionen**.

Definition 3 *Revision*

Eine Revision ist eine Version, die eine vorhandene Version ersetzen soll.[Ebe07]

So könnte z.B. die Version des Statechart-Diagramms aus Abbildung 2.2 als eine Revision des Diagramms aus Abbildung 2.1 aufgefasst werden.

Versionen eines Artefakts können zudem parallel existieren und verwendet werden. So zum Beispiel zur parallelen Entwicklung alternativer Software-Bausteine in der Software-Entwicklung. Diese Art von Versionen werden als **Varianten** bezeichnet.

Definition 4 *Variante*

Eine Variante ist eine Version, die die gleiche Funktion in einer anderen Form erfüllt.[Ebe07]

Ein Artefakt kann in unterschiedlichen Versionen vorliegen. Dies ist in der Softwareentwicklung, bei der Änderungsanforderungen ständig die Weiterentwick-

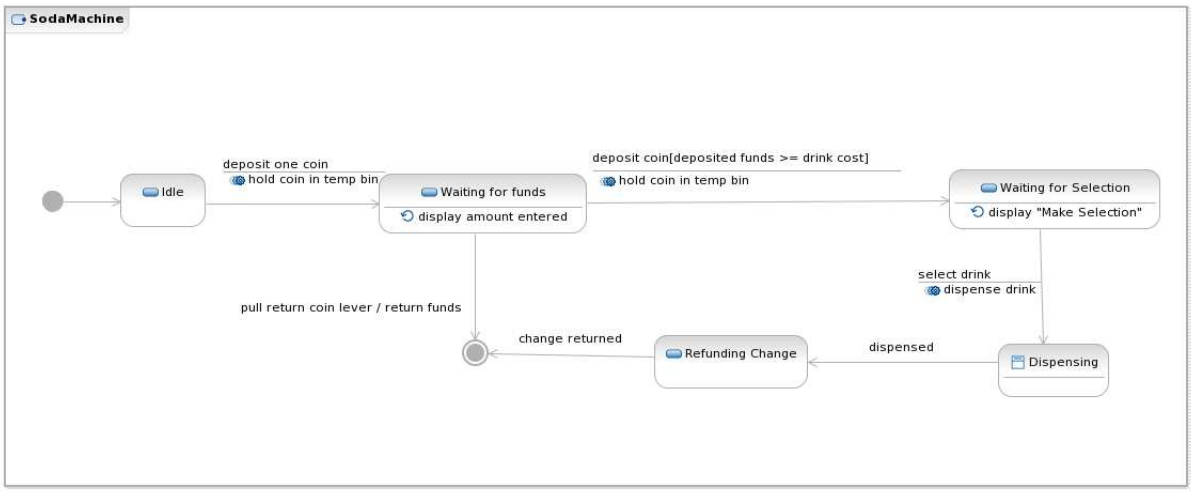


Abbildung 2.2.: Fallbeispiel: Eine weitere Version des UML Statechart-Diagramms aus Abbildung 2.1

lung vorantreiben und durch Variantenbildung neue Entwicklungszweige entstehen, üblich.

Die in den Abbildungen 2.1 und 2.2 dargestellten Diagramme könnten als Varianten aufgefasst werden. Beide erfüllen die gleiche Funktion, sie dienen als Modell eines Getränkeautomaten.

Versionen werden durch verschiedene **Werkzeuge** erzeugt und verarbeitet. In der Softwaretechnik zählen hierzu vor allem Werkzeuge, wie z.B. Editoren und Entwicklungsumgebungen. Diese ermöglichen es einem Anwender neue Versionen zu erstellen oder bestehende Versionen zu verändern.

2.2.1. Modelle

Modelle sind ein beliebtes Mittel um *Etwas* zu veranschaulichen, es einer Untersuchung zu unterziehen, gemeinsam mit mehreren Personen zu analysieren und darüber zu diskutieren. Man verwendet sie um *Etwas*, ein **Original**, darzustellen, zu veranschaulichen und aus verschiedenen *Blickwinkeln* zu betrachten.

Durch Abstraktion entsteht ein *Abbild*. Ein Modell ist somit das Abbild eines Originals. Diese Eigenschaft von Modellen wird als das **Abbildungs-Merkmal** bezeichnet.

> Jedes Modell steht für etwas anderes: sein Original. [Sta73]

Die Statechart-Diagramme in den Abbildungen 2.1 und 2.2 stehen beide für einen Getränkeautomaten. Die Diagramme sind Abbilder des Getränkeautomaten.

Bei der Abbildung werden die, für den Betrachter und dessen Intention, als überflüssig eingestuften Eigenschaften gefiltert. Zudem können Eigenschaften verän-

dert oder sogar hinzugefügt werden. Auch dieses Merkmal gilt für alle Modelle und wird als das **Reduktions-Merkmal** bezeichnet.

> Ein Modell weist nicht alle Eigenschaften des Originals auf, sondern nur einige - und auch die möglicherweise in veränderter, "ähnlicher" Form. [Sta73]

Man spricht hier auch von **präterierten Eigenschaften**. So werden z.B. in den Diagrammen des Fallbeispiels viele Eigenschaften des Getränkeautomaten nicht abgebildet, wie z.B. dessen optische Eigenschaften oder die Details der Benutzerschnittstelle. Diese Eigenschaften sind für das Modell unwichtig.

Bei der Abbildung kann der Betrachter dem Abbild zusätzliche Eigenschaften zusprechen, die nicht dem Original zugesagt werden können. Man nennt sie **abundante Eigenschaften**. Sie können aus unterschiedlichsten Gründen vom Modellierer dem Abbild hinzugefügt werden, beispielsweise um bestimmte Merkmale des Originals hervorzuheben.

Ein Modell wird immer in Hinblick auf ein bestimmtes Ziel erstellt. Jedes Modell weist somit ein **Pragmatisches-Merkmal** auf.

> Ein Modell hat den Zweck unter bestimmten Bedingungen und bezüglich bestimmter Fragestellungen das Original zu ersetzen. [Sta73]

Die beiden Diagramme des Fallbeispiels erfüllen den Zweck, die Zustände und Zustandsübergänge eines Getränkeautomaten darzustellen.

Es wurden die wichtigsten Eigenschaften von Modellen betrachtet, die für alle Formen von Modellen, also auch jene, welche wir in der Softwaretechnik zur Entwicklung von Softwaresystemen verwenden, gültig sind.

Definition 5 *Modell*

Ein Modell ist ein zielgerichtetes Abbild (im weiteren Sinne), das die Realität auf die problemrelevanten Aspekte vereinfacht und dazu ähnliche Beobachtungen ermöglicht, wie das Ausgangssystem.[Ebe06]

Die vorliegende Arbeit befasst sich mit Modellen, die in einer maschinenlesbaren Form vorliegen. Da jedes Modell einen konkreten Sachverhalt darstellt, handelt es sich bei maschinenlesbaren Modellen um Artefakte. Sie können wie jedes andere Artefakt auch in mehreren Versionen und Varianten vorkommen.

2.3. TGraphen

Zur Repräsentation von Artefakten können **TGraphen** verwendet werden [ERSB08], [BHE09]. Sie bilden eine allgemeine Klasse von Graphen. Jeder gerich-

tete oder ungerichtete Graph kann als TGraph repräsentiert werden. Zudem ermöglichen TGraphen die Attributierung und Typisierung von Graphelementen und der Graphen selbst.

TGraphen bilden eine effiziente und mächtige Datenstruktur zur Repräsentation von beliebigen Artefakten, und insbesondere von Modellen. Zu ihren wichtigsten charakteristischen Eigenschaften zählen:

Gerichtet. Kanten in TGraphen sind gerichtet. Zu jeder Kante existiert ein Startknoten und ein Zielknoten. Dennoch ist es möglich auch ungerichtete Graphen zu repräsentieren und die Traversierung des Graphen wird nicht durch die Richtung der Kanten eingeschränkt.

Typisiert. Jedem Graphenelement und dem Graphen wird ein Typ zugeordnet. Man unterscheidet die Typen nach Kanten-, Knoten- und Graphen-Typen. Zudem ermöglichen es TGraphen Typhierarchien zu bilden.

Attributiert. Jedem Graphenelement und dem Graphen können Attribute mit Werten zugeordnet werden.

Angeordnet. Die Graphenelemente eines TGraphen stehen in einer totalen Anordnung zueinander. Die Anordnungen werden in injektiven Sequenzen (iseq) gehalten.

Alle Eigenschaften von TGraphen werden in der folgenden Definition formal erfasst.

Definition 6 *TGraph*

Sei

- *Vertex das Universum der Knoten*
- *Edge das Universum der Kanten*
- *TypeId das Universum der Typbezeichner*
- *AttrId das Universum der Attributbezeichner*
- *Value das Universum der Attributwerte*

Seien zudem

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

Dann bildet das Tupel $TG = (V_{Seq}, E_{Seq}, \Lambda_{Seq}, type, value)$ *einen TGraphen, falls gilt:*

- $V_{Seq} \in iseq\, V$ *ist eine Anordnung von* V
- $E_{Seq} \in iseq\, E$ *ist eine Anordnung von* E
- $\Lambda_{Seq} : V \rightarrow iseq(E \times \{in, out\})$ *ist eine Inzidenzabbildung, für die gilt:*

$$\forall\, e \in E\ \exists! v, w \in V : (e, out) \in ran\, \Lambda_{Seq}(v) \wedge (e, in) \in ran\, \Lambda_{Seq}(w)$$

- $type : V \cup E \rightarrow TypeId$ *ist eine Typisierung*
- $value : V \cup E \rightarrow (AttrId \twoheadrightarrow Value)$ *ist eine Attributierung wobei gilt:*

$$\forall\, x, y \in V \cup E : type(x) = type(y) \Rightarrow dom(value(x)) = dom(value(y))$$

Anhand des Beispiels 1 auf der nächsten Seite können nun weitere Begriffe, siehe [DW98], erläutert werden.

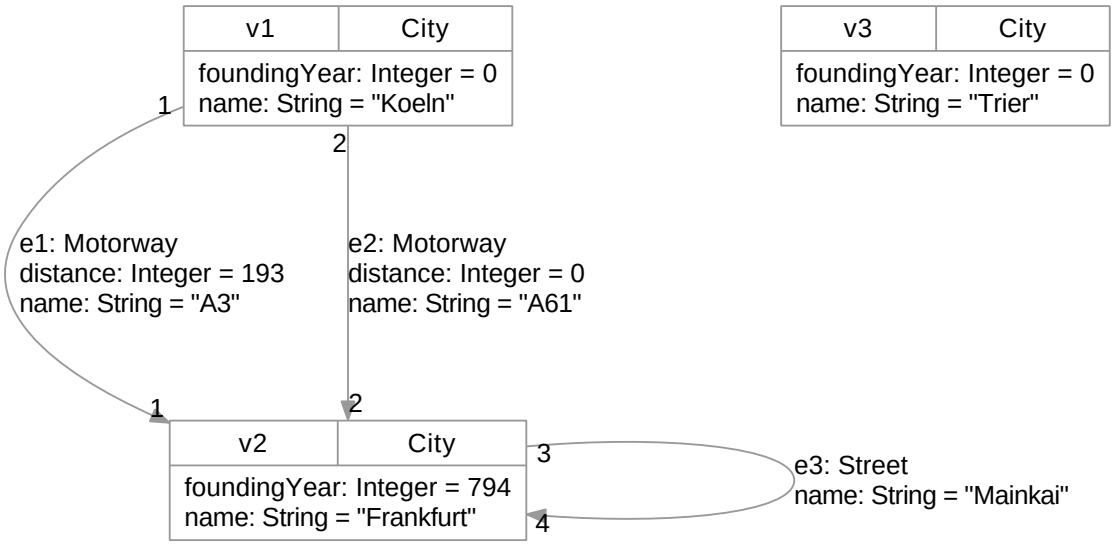
Ein Knoten $v \in V$ wird als **isoliert** bezeichnet, wenn der Knoten v zu keiner Kante $e \in E$ inzident ist und somit für den Knoten gilt: $\Lambda_{Seq}(v) = \langle\rangle$. In Beispiel 1 ist der Knoten $v3$ isoliert.

Die Abbildungen $\alpha : E \rightarrow V$ und $\omega : E \rightarrow V$ liefern zu einer Kante $e \in E$ den **Startknoten** $\alpha(e)$, bzw. den **Zielknoten** $\omega(e)$. In Beispiel 1 gilt für die Abbildungen: $\alpha = \{e1 \mapsto v1, e2 \mapsto v1, e3 \mapsto v2\}$ und $\omega = \{e1 \mapsto v2, e2 \mapsto v2, e3 \mapsto v2\}$.

Wenn der Startknoten und der Zielknoten einer Kante $e \in E$ identisch sind, $\alpha(e) = \omega(e)$, so wird die Kante als **Schlinge** bezeichnet. In Beispiel 1 kann die Kante $e3$ als Schlinge bezeichnet werden, da $\alpha(e3) = v2 = \omega(e3)$ gilt.

TGraphen können **Mehrfachkanten** enthalten. Mehrfachkanten sind Kanten mit identischem Startknoten und Zielknoten. Zwei Kanten $e1, e2 \in E$ werden als Mehrfachkanten bezeichnet, wenn gilt: $e1 \neq e2 \wedge \alpha(e1) = \alpha(e2) \wedge \omega(e1) = \omega(e2)$.

Beispiel 1 *CityGraph01*
Die folgende Abbildung zeigt einen TGraphen, bestehend aus drei Knoten und drei Kanten.



Dieser TGraph ist wie folgt formal definiert:
$CityGraph01 = (V_{01Seq}, E_{01Seq}, \Lambda_{01Seq}, type_{01}, value_{01})$ *mit:* $V_{01Seq} = \langle v1, v2, v3 \rangle$
$E_{01Seq} = \langle e1, e2, e3 \rangle$
$\Lambda_{01Seq} : V \rightarrow iseq(E \times \{in, out\})$ *mit*
$\Lambda_{01Seq} = \{v1 \mapsto \langle (e1, out), (e2, out) \rangle, v2 \mapsto \langle (e1, in), (e2, in), (e3, out), (e3, in) \rangle, v3 \mapsto \langle\rangle\}$
$type_{01} : V_{01Seq} \cup E_{01Seq} \rightarrow TypeId$ *mit*
$type_{01} = \{v1 \mapsto City, v2 \mapsto City, v3 \mapsto City, e1 \mapsto Motorway, e2 \mapsto Motorway, e3 \mapsto Street\}$
$value_{01} : V_{01Seq} \cup E_{01Seq} \rightarrow (AttrId \twoheadrightarrow Value)$ *mit*
$value_{01} = \{v1 \mapsto \{(name = \text{“}Koeln\text{“})\}, v2 \mapsto \{(name = \text{“}Frankfurt\text{“}), (foundingYear = 794)\}, v3 \mapsto \{(name = \text{“}Trier\text{“})\}, e1 \mapsto \{(name = \text{“}A3\text{“}), (distance = 193)\}, e2 \mapsto \{(name = \text{“}A61\text{“})\}, e2 \mapsto \{(name = \text{“}Mainkai\text{“})\}\}$

Die Inzidenzabbildung Λ_{Seq} liefert für einen Knoten $v \in V$ die Sequenz aller eingehenden und ausgehenden Kanten. Die Abbildung $\Lambda^{-}_{Seq} : V \rightarrow \text{iseq}(E \times \{in\})$ liefert für jeden Knoten v die **Sequenz der eingehenden Kanten**. In Beispiel 1 gilt: $\Lambda^{-}_{Seq} = \{v1 \mapsto \langle\rangle, v2 \mapsto \langle (e1, in), (e2, in), (e3, in) \rangle, v3 \mapsto \langle\rangle\}$. Analog hierzu liefert die Abbildung $\Lambda^{+}_{Seq} : V \rightarrow \text{iseq}(E \times \{out\})$ die **Sequenz der ausgehenden Kanten** für einen Knoten v. In dem Graphen aus Beispiel 1 gilt: $\Lambda^{-}_{Seq} = \{v1 \mapsto \langle (e1, out), (e2, out) \rangle, v2 \mapsto \langle (e3, out) \rangle, v3 \mapsto \langle\rangle\}$.

Zu jedem Knoten $v \in V$ bestimmt der **Grad des Knoten** $\delta(v)$ die Anzahl der mit v verbundenen Inzidenzen. Es gilt: $\delta : V \rightarrow \mathbb{N}$ mit $\delta(v) = \#\Lambda_{Seq}(v)$. Für den

Graphen in Beispiel 1 gilt $\delta = \{v1 \mapsto 2, v2 \mapsto 4, v3 \mapsto 0\}$. Der **Innengrad** eines Knoten gibt die Anzahl der eingehenden Kanten an: $\delta^-(v) = \#\Lambda^-_{Seq}(v)$. Für den Graphen in Beispiel 1 gilt $\delta^- = \{v1 \mapsto 0, v2 \mapsto 3, v3 \mapsto 0\}$. Der **Außengrad** eines Knoten gibt die Anzahl der ausgehenden Kanten an: $\delta^+(v) = \#\Lambda^+_{Seq}(v)$. Für den Graphen in Beispiel 1 gilt $\delta^+ = \{v1 \mapsto 2, v2 \mapsto 1, v3 \mapsto 0\}$.

Die TGraphenelemente und TGraphen selbst werden durch **eindeutige Bezeichner** gekennzeichnet. Sei *TGraph* das Universum aller TGraphen. Dann liefert die Abbildung $graphId : TGraph \rightarrow String$ zu jedem TGraphen dessen Bezeichner. Die Knoten eines TGraphen verfügen über eindeutige Bezeichner. Sei $TG = (V_{Seq}, E_{Seq}, \Lambda_{Seq}, type, value)$ ein TGraph. Die totale Injektion $vertexId : V_{Seq} \rightarrowtail \mathbb{N}$ weist jedem Knoten des TGraphen eine natürliche Zahl als Identifikationsmerkmal zu. Ebenso verfügen Kanten über ein eindeutiges Identifikationsmerkmal. Die totale Injektion $edgeId : E_{Seq} \rightarrowtail \mathbb{N}$ weist jeder Kante des TGraphen eine natürliche Zahl als Identifikationsmerkmal zu. Für den TGraphen aus Beispiel 1 gilt: $graphId(CityGraph01) =$ "*CityGraph*01", $vertexId = \{v1 \mapsto 1, v2 \mapsto 2, v3 \mapsto 3\}$ und $edgeId = \{e1 \mapsto 1, e2 \mapsto 2, e3 \mapsto 3\}$

Die TGraphen zu dem Statechart-Diagrammen des Fallbeispiels (Abbildungen 2.1 und 2.2) sind in den Abbildungen 2.3 und 2.3 dargestellt.

2.3.1. Das Java Graphenlabor

Für die Erzeugung und Verarbeitung von TGraphen wurde am Institut für Softwaretechnik[1] das Java Graphenlabor **Java Graph Laboratory**[2], kurz JGraLab, entwickelt. Es handelt sich dabei um eine Java Klassenbibliothek, die eine hoch effiziente API zur Verarbeitung von TGraphen bereitstellt. JGraLab ermöglicht die Erzeugung, den Zugriff und die Manipulation von TGraphen zur Laufzeit. Dadurch können TGraphen in Anwendungen als Datenstrukturen eingesetzt werden. [Ebe87, ERW08, ERSB08]

Das Java Graphenlabor verfügt über eine stetig wachsende Anzahl an Werkzeugen zur weiteren Verarbeitung von TGraphen. Die **Graph Repository Query Language 2** (GReQL2) ist eine Abfragesprache für TGraphen, die zur Extraktion von inhaltlichen, strukturellen oder aggregierten Informationen aus TGraphen genutzt werden kann[Mar06]. Mit der **Graph Repository Transformation Language** (GReTL) verfügt das Graphenlabor über eine Transformationssprache, die den operationalen Ansatz zur Modelltransformation umsetzt, siehe [Wei09] und [HE10]. Jeder TGraph kann durch JGraLab im dot-Format exportiert und anschließend mit Hilfe des Werkzeugs dot aus dem Graphviz[3]-Paket visualisiert werden, wie in Beispiel 1 zu sehen ist.

[1] `http://www.uni-koblenz.de/FB4/Institutes/IST` Stand: 1. Juli 2010
[2] `http://JGraLab.uni-koblenz.de/` Stand:1. Juli 2010
[3] `http://www.graphviz.org/`

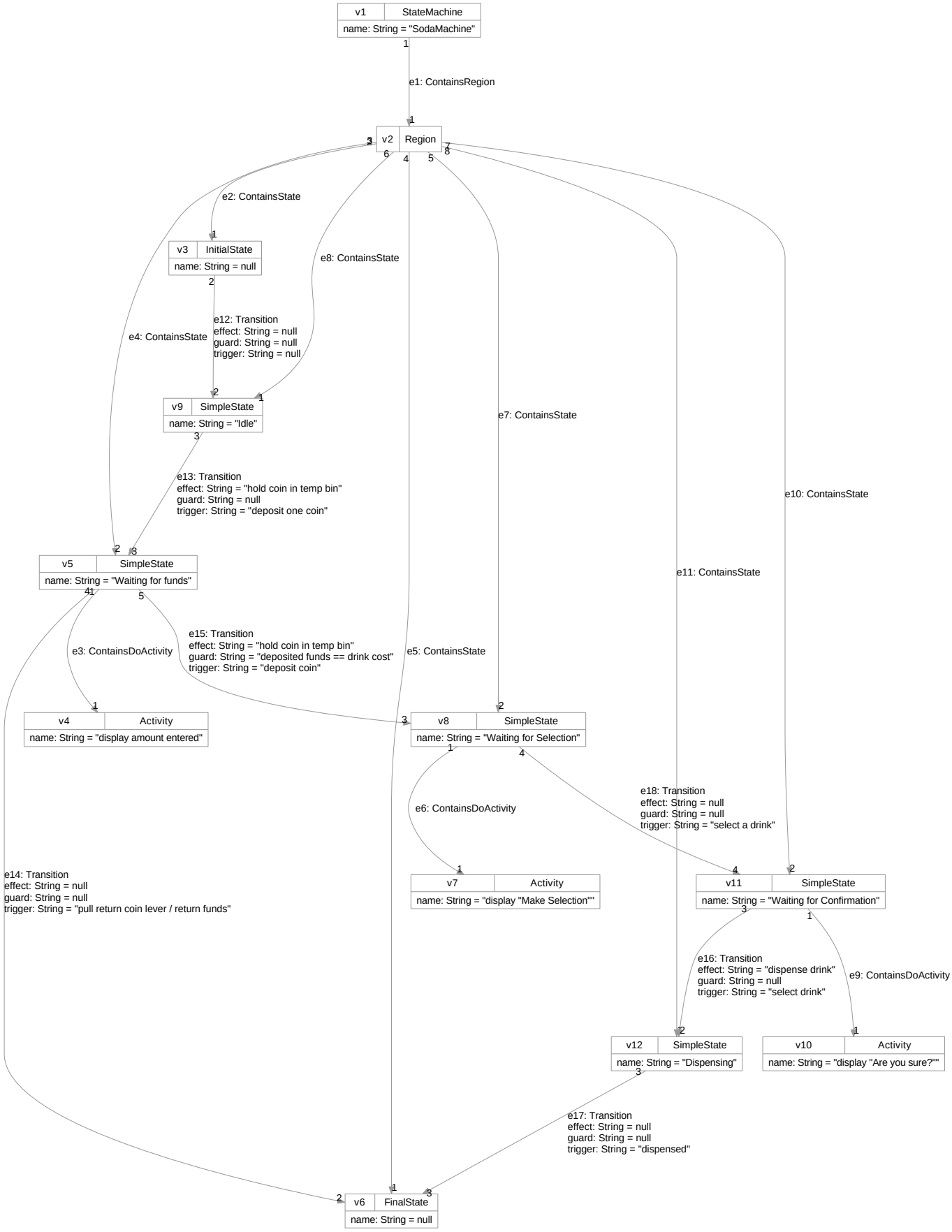


Abbildung 2.3.: Fallbeispiel: Die erste Version des Statechart-Diagramms aus Abbildung 2.1 repräsentiert durch einen TGraphen

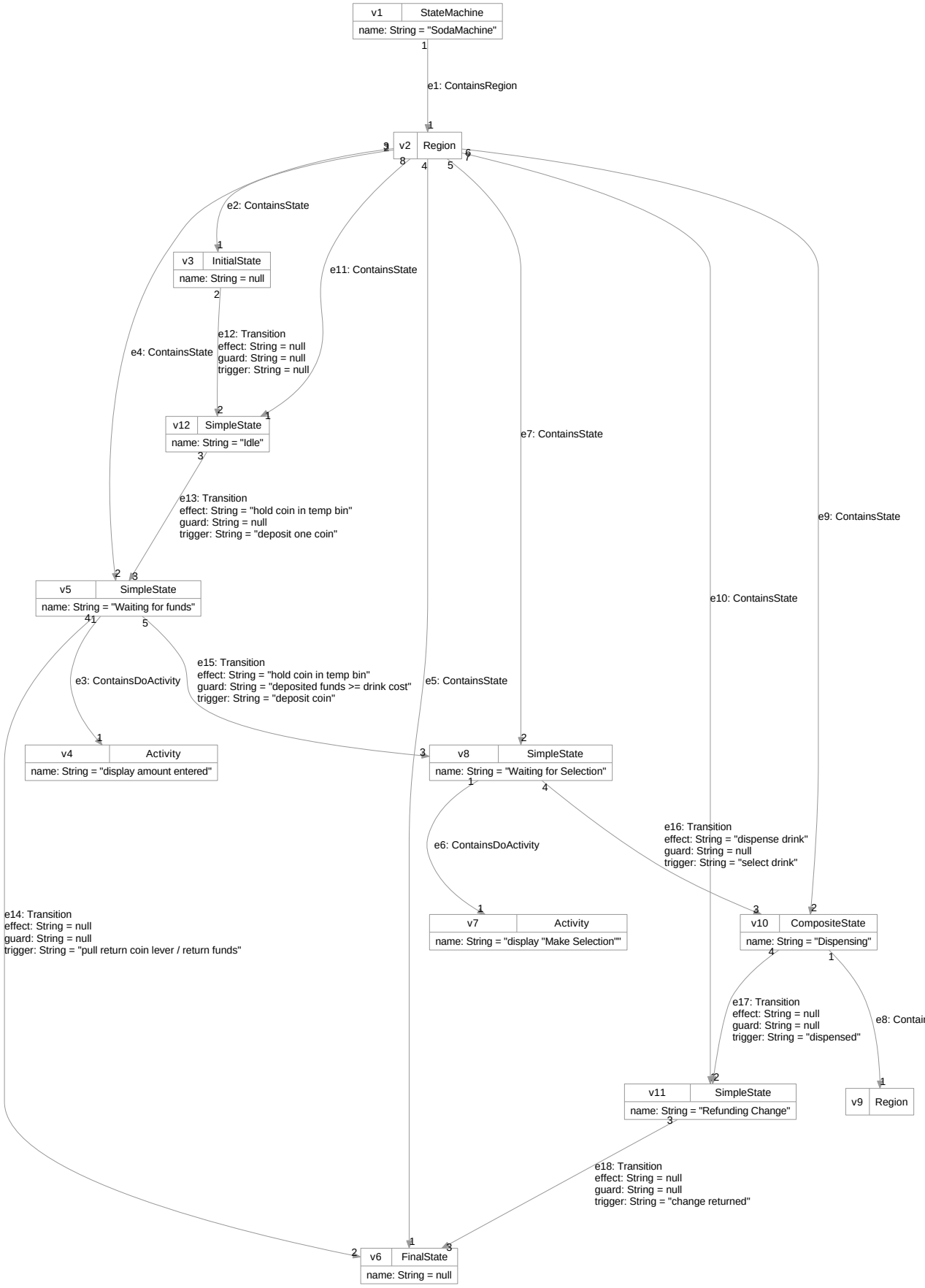


Abbildung 2.4.: Fallbeispiel: Die zweite Version des Statechart-Diagramms aus Abbildung 2.2, repräsentiert durch einen TGraphen

Viele weitere **Werkzeuge und Konzepte** runden den TGraph-Ansatz ab. Durch die fortlaufende Weiterentwicklung wird die Funktionalität, Stabilität und Effizienz von JGraLab stetig gesteigert.

2.3.2. TGraphen, Schemata und das Meta-Schema

Die Struktur und die Elemente eines TGraphen werden durch ein **TGraph-Schema** festgelegt. Jeder TGraph ist konform zu einem TGraph-Schema. Er darf nur die Knoten- und Kanten-Typen enthalten, die durch das Schema definiert sind. Zudem legt das Schema fest, welche Knoten- und Kanten-Typen miteinander in Verbindung stehen dürfen. Die Syntax eines TGraphen wird somit durch ein TGraph-Schema spezifiziert. Es regelt die Typisierung und Attributierung der TGraphen, die zu dem Schema konform sind.

Ein TGraph-Schema bildet ein **Meta-Modell** für eine Klasse von TGraphen. Jeder zu dem Schema konforme TGraph stellt eine Instanz des TGraph-Schemas dar. Jedes TGraph-Schema selbst ist eine Instanz des Meta-Schema. Das Meta-Schema spezifiziert die Syntax aller TGraph-Schemata.

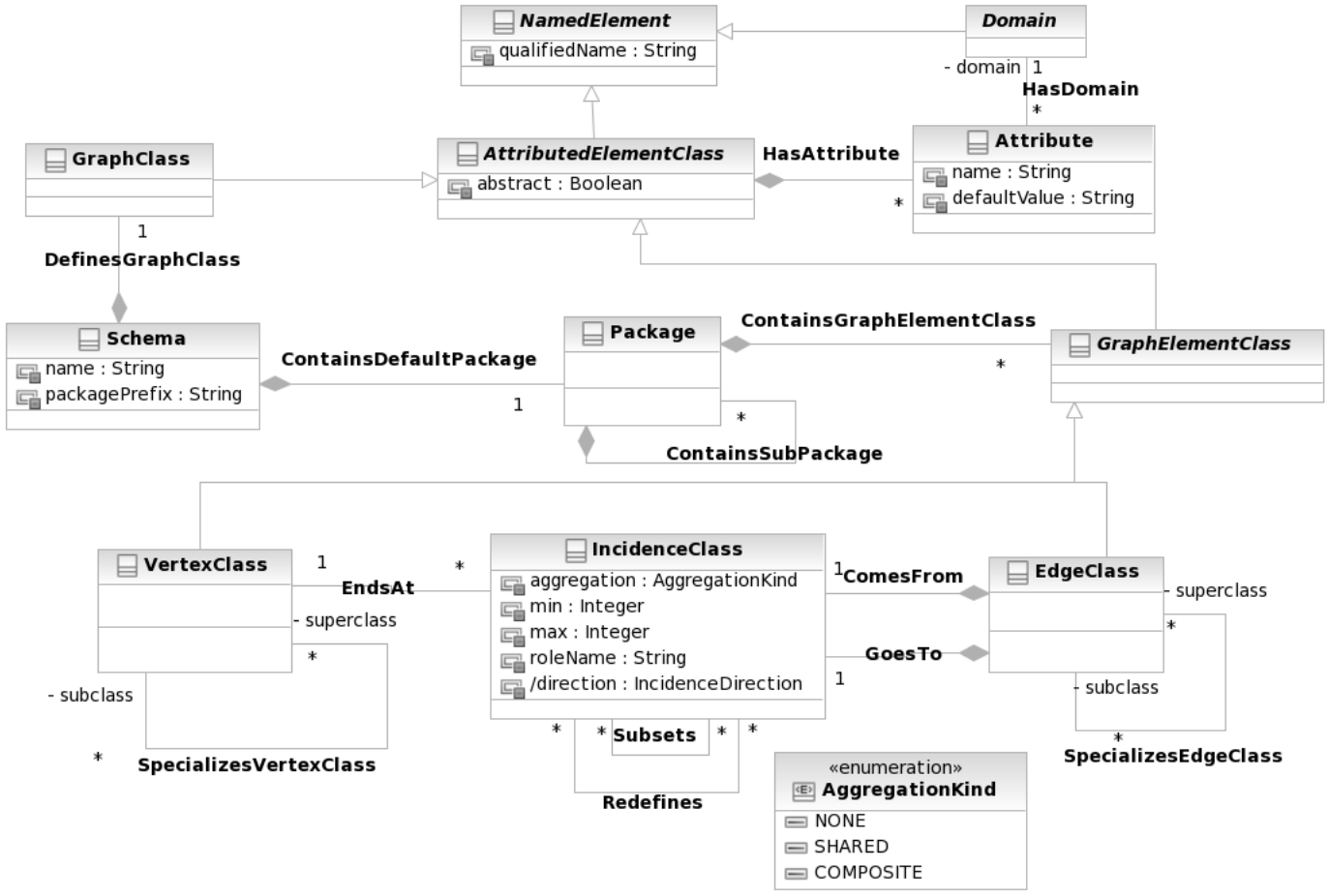


Abbildung 2.5.: Das grUML Meta-Schema (structure) in vereinfachter Darstellung

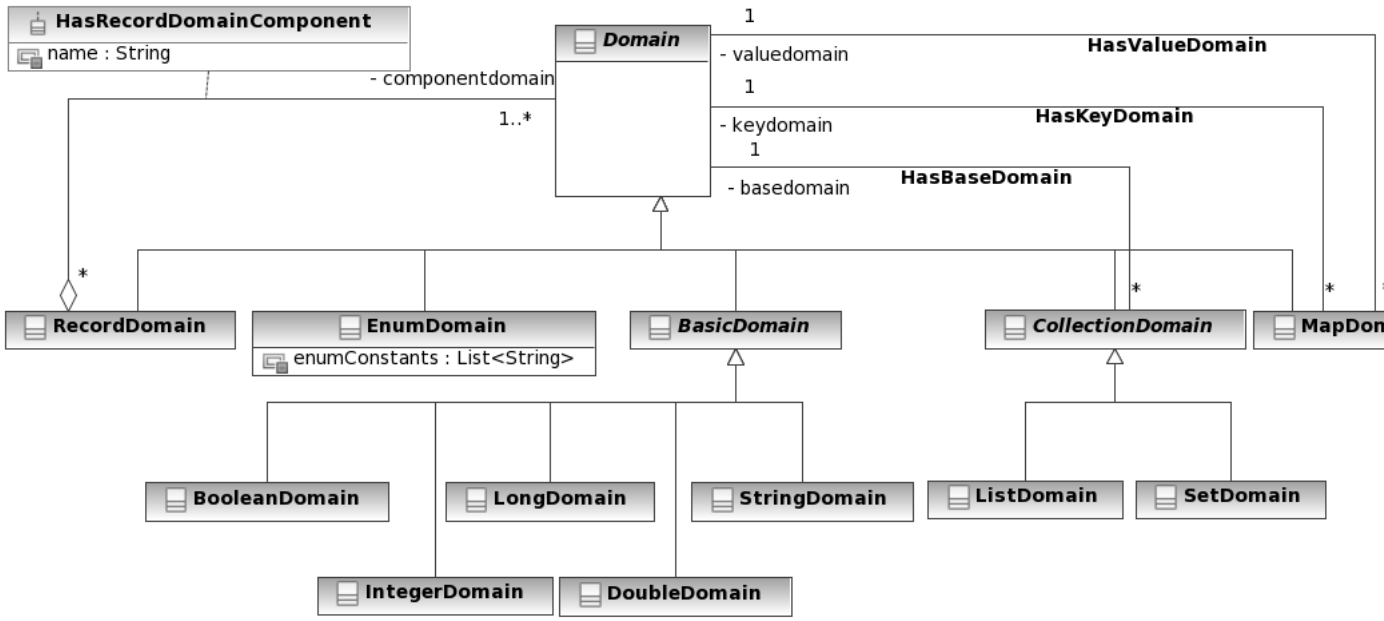


Abbildung 2.6.: Das grUML Meta-Schema (domains)

Zur Modellierung von TGraph-Schemata und dem Meta-Schema wird **graphUML**(grUML) verwendet. Es handelt sich dabei um ein Profil der UML2 Klassendiagramme. grUML ermöglicht es TGraph-Schemata mit Hilfe von UML2-Editoren zu modellieren.[BHR+10]. Das Meta-Schema, modelliert mit grUML, ist in den Abbildungen 2.5 und 2.6 zu sehen.

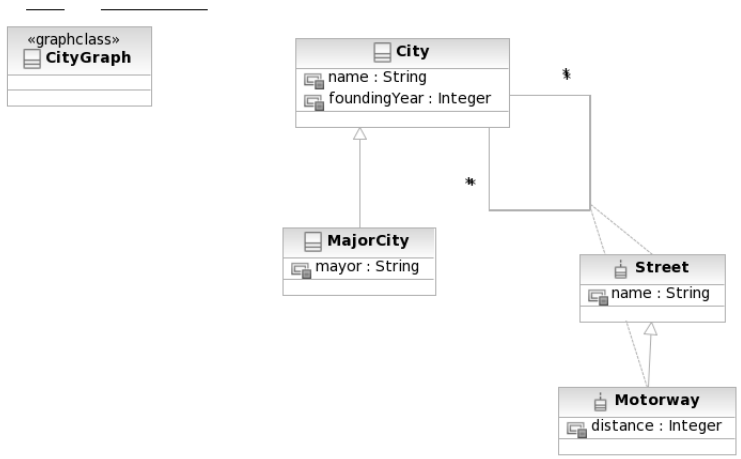


Abbildung 2.7.: Beispiel für ein TGraph-Schema.

In Abbildung 2.7 ist ein einfaches TGraph-Schema, modelliert mit grUML, dargestellt. Ein grUML Diagramm modelliert eine Klasse von TGraphen. Es spezifiziert die Struktur und die Bedingungen für eine Menge von TGraphen, die Instanzen der Klasse sind.[BHR+10]

In dem Beispiel in Abbildung 2.7 wird eine Klasse von TGraphen, CityGraph, modelliert.

Die Graphenelemente werden mit Hilfe von **Klassen** und **Assoziationen** modelliert. Klassen repräsentieren die Knoten-Typen. Kanten-Typen werden durch einfache Assoziationen, bei nicht attributierten Kanten-Typen, oder durch Assoziationsklassen modelliert. In dem Beispiel aus Abbildung 2.7 wird zwei Knoten-Typen City und MajorCity, und zwei Kanten-Typen Street und Motorway modelliert. Der Knoten-Typ MajorCity ist ein Subtyp von City. Ebenso ist der Kanten-Typ Motorway ein Subtyp von Street.

JGraLab ermöglicht den **Import** von im XMI Format exportierten grUML-Diagrammen und erzeugt daraus einen voll funktionsfähigen und effizienten Java Source Code zur Erzeugung und Manipulation von TGraphen, die zu dem modellierten Schema konform sind. Ein mit Hilfe des Werkzeugs RSA modelliertes TGraph-Schema kann im XMI Austauschformat von JGraLab mit Hilfe des Programms de.uni_koblenz.JGraLab.utilities.rsa2tg.Rsa2Tg importiert werden. Als Ausgabe erzeugt das Programm ein TGraphSchema im TG Datei Format.

Das **TG Datei Format**[4] ist ein speziell für die effiziente, persistente Speicherung von TGraphen ausgerichtetes Dateiformat. Es ermöglicht die Speicherung von Schemata und TGraphen. Eine TG Datei hat die Dateiendung .tg. Es enthält zumindest ein TGraph-Schema und optional einen zum Schema konformen TGraphen. Aus einer TG Datei kann JGraLab Java Source Code generieren, mit dem sich zu dem in der TG Datei gespeicherten Schema konforme TGraphen erzeugen und manipulieren lassen. Bereits generierte TGraphen können persistent in einer TG Datei gespeichert und wieder aus der TG Datei geladen werden.

Das Listing 3-1 zeigt eine solche TG Datei. Diese TG Datei wurde aus dem Modell in Abbildung 2.7 mit Hilfe des Werkzeugs RSA2TG erzeugt. Es enthält ein TGraph-Schema, jedoch keinen TGraphen.

```
1  TGraph 2;
2
3  Schema de.uni_koblenz.jgralab.diff.example.CityGraphSchema;
4
5  GraphClass CityGraph;
6
7  Package ;
8  Package structure;
9  EdgeClass Motorway: Street from City (0,*) to City (0,*) {
     distance: Integer};
10 VertexClass City {name: String, foundingYear: Integer};
11 EdgeClass Street from City (0,*) to City (0,*) {name: String};
12 VertexClass MajorCity: City {mayor: String};
```

Listing 3-1: Das TGraph-Schema aus Abbildung 2.7 im TG Datei Format

Ein TGraph kann in Form einer TG Datei gespeichert werden. Das folgende Listing zeigt eine TG Datei zu dem TGraphen aus Beispiel 1. Die Datei enthält sowohl den TGraphen, auch als das zugehörige TGraph-Schema.

[4] http://www.uni-koblenz.de/~ist/TG_Format

```
1  // JGraLab - The Java graph laboratory
2  //   Version : Dimetrodon
3  //   Revision: 2726
4  //   Build ID: 69
5
6  TGraph 2;
7  Schema de.uni_koblenz.jgralab.diff.example.CityGraphSchema;
8  GraphClass CityGraph;
9  Package structure;
10 VertexClass City { foundingYear: Integer,  name: String };
11 VertexClass MajorCity: City { mayor: String };
12 EdgeClass Motorway: Street from City (0,*) to City (0,*) {
      distance: Integer };
13 EdgeClass Street from City (0,*) to City (0,*) { name: String };
14 Graph "CityGraph01" 14 CityGraph (100 100 3 3);
15 Package structure;
16 1 City <1 2> 0 "Koeln";
17 2 City <-1 -2 3 -3> 794 "Frankfurt";
18 3 City <> 0 "Trier";
19 1 Motorway 193 "A3";
20 2 Motorway 0 "A61";
21 3 Street "Mainkai";
```

Listing 3-2: Der TGraph aus Beispiel 1 im TG Datei Format

Das Schema des Fallbeispiels

Das Schema zum Fallbeispiel ist in Abbildung 2.8 zu sehen. Mit dessen Hilfe können Statechart-Diagramme durch TGraphen repräsentiert und mit Hilfe der JGraLab-API verarbeitet werden. Listing 3-3 zeigt die entsprechende TG-Datei zum Schema.

```
1  TGraph 2;
2
3  Schema de.uni_koblenz.jgralab.diff.example.StatechartSchema;
4
5  GraphClass StatechartDiagram;
6
7  Package ;
8  Package structure;
9  VertexClass StateMachine {name: String};
10 abstract VertexClass State {name: String};
11 EdgeClass ContainsRegion from StateMachine (1,1) to Region (0,*)
       role regions aggregation composite;
12 abstract VertexClass PseudoState: State;
13 VertexClass InitialState: PseudoState;
14 VertexClass Choice: PseudoState;
15 abstract VertexClass History: PseudoState;
16 VertexClass EntryPoint: PseudoState;
17 VertexClass ExitPoint: PseudoState;
```

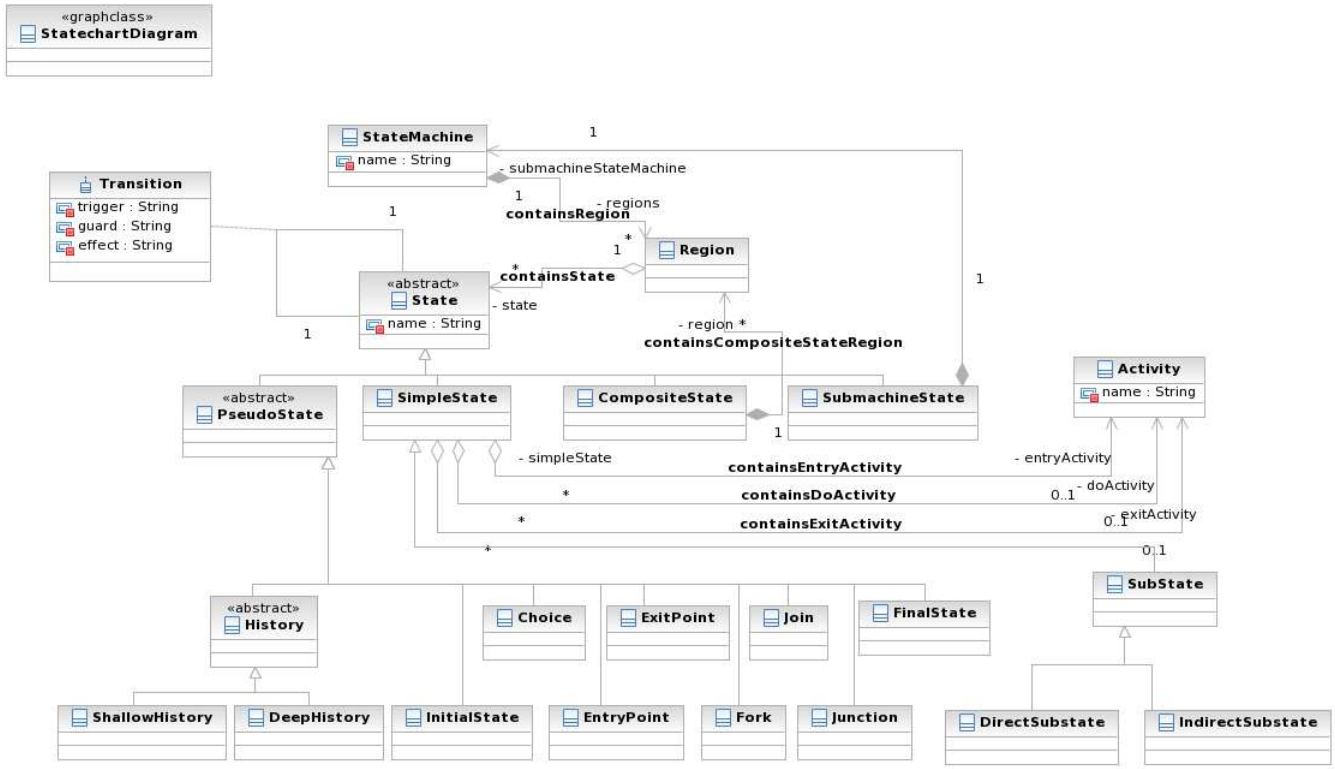


Abbildung 2.8.: Das TGraph-Schema zum Fallbeispiel. Es ermöglicht die Repräsentation von UML-Statechart Diagrammen mittels TGraphen.

```
18 VertexClass Fork: PseudoState;
19 VertexClass Join: PseudoState;
20 VertexClass Junction: PseudoState;
21 VertexClass SimpleState: State;
22 VertexClass CompositeState: State;
23 VertexClass SubmachineState: State;
24 VertexClass Region;
25 EdgeClass ContainsCompositeStateRegion from CompositeState (1,1)
       to Region (0,*) role region aggregation composite;
26 EdgeClass ContainsState from Region (1,1) to State (0,*) role
     state aggregation shared;
27 VertexClass SubState: SimpleState;
28 VertexClass DirectSubstate: SubState;
29 VertexClass IndirectSubstate: SubState;
30 EdgeClass IsPartOfSubmachineStateMachine from SubmachineState
     (1,1) to StateMachine (1,1) role submachineStateMachine
     aggregation composite;
31 EdgeClass Transition from State (1,1) to State (1,1) {trigger:
     String, guard: String, effect: String};
32 VertexClass Activity {name: String};
33 EdgeClass ContainsEntryActivity from SimpleState (0,*) role
     simpleState to Activity (0,1) role entryActivity aggregation
     shared;
34 VertexClass DeepHistory: History;
```

```
35 EdgeClass ContainsDoActivity from SimpleState (0,*) to Activity
     (0,1) role doActivity aggregation shared;
36 EdgeClass ContainsExitActivity from SimpleState (0,*) to
     Activity (0,1) role exitActivity aggregation shared;
37 VertexClass ShallowHistory: History;
38 VertexClass FinalState: PseudoState;
```

Listing 3-3: Das TGraph-Schema des Fallbeispiels aus Abbildung 2.8 im TG Datei Format

2.4. Beziehungen zwischen TGraphen

2.4.1. TGraph-Morphismen

TGraphen können durch TGraph-Morphismen miteinander in Verbindung gesetzt werden. Ein TGraph-Morphismus zwischen zwei TGraphen TG_1 und TG_2 bildet die Knoten und Kanten eines TGraphen auf die Knoten und Kanten des anderen TGraphen ab.

Definition 7 *TGraph-Morphismus*

Seien $TG_1 = (V_{1Seq}, E_{1Seq}, \Lambda_{1Seq}, type_1, value_1)$ *und* $TG_2 = (V_{2Seq}, E_{2Seq}, \Lambda_{2Seq}, type_2, value_2)$ *zwei TGraphen. Ein TGraph-Morphismus* $f : TG_1 \rightarrow TG_2$ *ist ein Tupel* $f = (f_V, f_E)$*. Die Funktion* $f_V : V_1 \rightarrow V_2$ *und* $f_E : E_1 \rightarrow E_2$ *bilden die Knoten, bzw. Kanten des TGraphen* TG_1 *auf* TG_2 *ab.*

$$\begin{array}{ccc} E_1 & \overset{\alpha_1}{\underset{\omega_1}{\rightrightarrows}} & V_1 \\ {\scriptstyle f_E}\downarrow & & \downarrow{\scriptstyle f_V} \\ E_2 & \overset{\alpha_2}{\underset{\omega_2}{\rightrightarrows}} & V_2 \end{array}$$

TGraph-Homomorphismen

Eine spezielle Form der TGraph-Morphismen bilden die TGraph-Homomorphismen. Sie sind strukturerhaltend.

Definition 8 *TGraph-Homomorphismus*

Seien $TG_1 = (V_{1Seq}, E_{1Seq}, \Lambda_{1Seq}, type_1, value_1)$ *und* $TG_2 = (V_{2Seq}, E_{2Seq}, \Lambda_{2Seq}, type_2, value_2)$ *zwei TGraphen. Ein TGraph-Homomorphismus* $f : TG_1 \rightarrow TG_2$ *ist ein TGraph-Morphismus* $f = (f_V, f_E)$*, für den gilt:* $f_V \circ \alpha_1 = \alpha_2 \circ f_E$ *und* $f_V \circ \omega_1 = \omega_2 \circ f_E$*.*

Die Struktur der TGraphen bleiben unter einem Homomorphismus erhalten.

TGraph-Isomorphismen

Ein TGraph-Morphismus ist injektiv, wenn beide Abbildungen f_V und f_E injektiv sind.
Ein TGraph-Morphismus ist surjektiv, wenn beide Abbildungen f_V und f_E surjektiv sind.
Sind beide Abbildungen injektiv und surjektiv, so ist der TGraph-Morphismus bijektiv.

Definition 9 *TGraph-Isomorphismus*

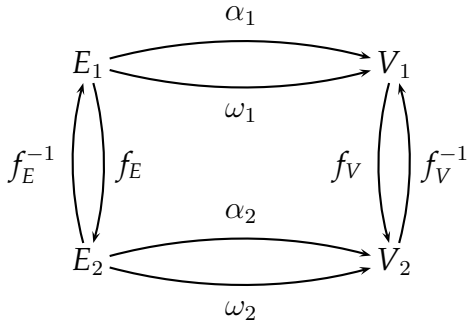
Seien $TG_1 = (V_{1Seq}, E_{1Seq}, \Lambda_{1Seq}, type_1, value_1)$ *und* $TG_2 = (V_{2Seq}, E_{2Seq}, \Lambda_{2Seq}, type_2, value_2)$ *zwei TGraphen. Ein TGraph-Isomorphismus* $f : TG_1 \rightarrow TG_2$ *ist ein TGraph-Homomorphismus* $f = (f_V, f_E)$*, so dass die Abbildungen* f_V *und* f_E *bijektiv sind.*

Ein TGraph-Isomorphismus erhält somit die Struktur zweier TGraphen und jedem Element eines TGraphen wird eindeutig ein Element in dem jeweils anderen TGraphen zugeordnet.

Zwei TGraphen TG_1 und TG_2 gelten als **isomorph**, geschrieben $TG_1 \cong TG_2$, wenn es einen TGraph-Isomorphismus zwischen TG_1 und TG_2 gibt.

Umkehrung von TGraph-Morphismen

Zu jedem injektiven, partiellen TGraph-Morphismus $f : TG_1 \rightarrowtail TG_2$ mit $f = (f_V, f_E)$ existiert die Umkehrung $f^{-1} : TG_2 \rightarrowtail TG_1$ mit $f^{-1} = (f_V^{-1}, f_E^{-1})$. Die Umkehrung eines TGraph-Morphismus besteht aus den Umkehrabbildungen $f_V^{-1} : V_2 \nrightarrow V_1$ und $f_E^{-1} : E_2 \nrightarrow E_1$.



Für TGraph-Morphismen gelten die gleichen Eigenschaften wie für Abbildungen im Allgemeinen. Zu einem injektiven TGraph-Morphismus f ist die Umkehrung

f^{-1} ebenfalls injektiv. Ist ein TGraph-Morphismus bijektiv, so ist auch dessen Umkehrung bijektiv.

2.4.2. Das Mapping von TGraphen

Für den Vergleich von zwei TGraphen wird eine eindeutige Zuordnung der Graphenelemente des einen TGraphen zu den Graphenelementen des anderen TGraphen benötigt. Ein **partieller, injektiver TGraph-Morphismus** ist eine solche Zuordnung und wird als **Mapping** bezeichnet.

Demnach handelt es sich bei jedem injektiven TGraph-Morphismus und insbesondere jedem TGraph-Isomorphismus um ein Mapping. Insbesondere gelten die Eigenschaften für injektive TGraph-Morphismen, wie die Bildung der Umkehrung, auch für TGraph-Mappings.

Ein Mapping liefert dem Vergleichsverfahren die notwendigen Informationen zur Identifizierung und Klassifizierung von Abweichungen zwischen zwei TGraphen.

Definition 10 *Mapping von TGraphen*

Ein Mapping M zweier TGraphen $TG_1 = (V_{1Seq}, E_{1Seq}, \Lambda_{1Seq}, type_1, value_1)$ *und* $TG_2 = (V_{2Seq}, E_{2Seq}, \Lambda_{2Seq}, type_2, value_2)$ *besteht aus zwei partiellen Injektionen* $m_V : V_1 \rightarrowtail V_2$ *und* $m_E : E_1 \rightarrowtail E_2$*, die einen partiellen, injektiven TGraph-Morphismus bilden.*

Jeder TGraph besteht aus Knoten und Kanten. Deshalb enthält ein Mapping zwischen zwei TGraphen zwei Abbildungen. Da ein Mapping eindeutig sein muss und nicht immer jedem Graphenelement aus einem TGraphen TG_L auch ein Graphenelement des TGraphen TG_R zugeordnet werden kann, handelt es sich hier um partielle Injektionen.

Problem: Mapping-Problem

Input: zwei TGraphen
$TG_1 = (V_{1Seq}, E_{1Seq}, \Lambda_{1Seq}, type_1, value_1)$ und
$TG_2 = (V_{2Seq}, E_{2Seq}, \Lambda_{2Seq}, type_2, value_2)$

Output: ein Mapping $M = (m_V, m_E)$ mit
$m_V : V_{1Seq} \rightarrowtail V_{2Seq}$ und
$m_E : E_{1Seq} \rightarrowtail E_{2Seq}$

Zu zwei TGraphen bestimmt der *map* Operator ein Mapping:

$$map : TGraph \times TGraph \rightarrow Mapping$$

TGraphen können beliebig komplex sein. Dadurch stellen sie ein sehr ausdrucksstarkes Mittel zur Repräsentation von Artefakten dar. Für Verfahren zur Lösung

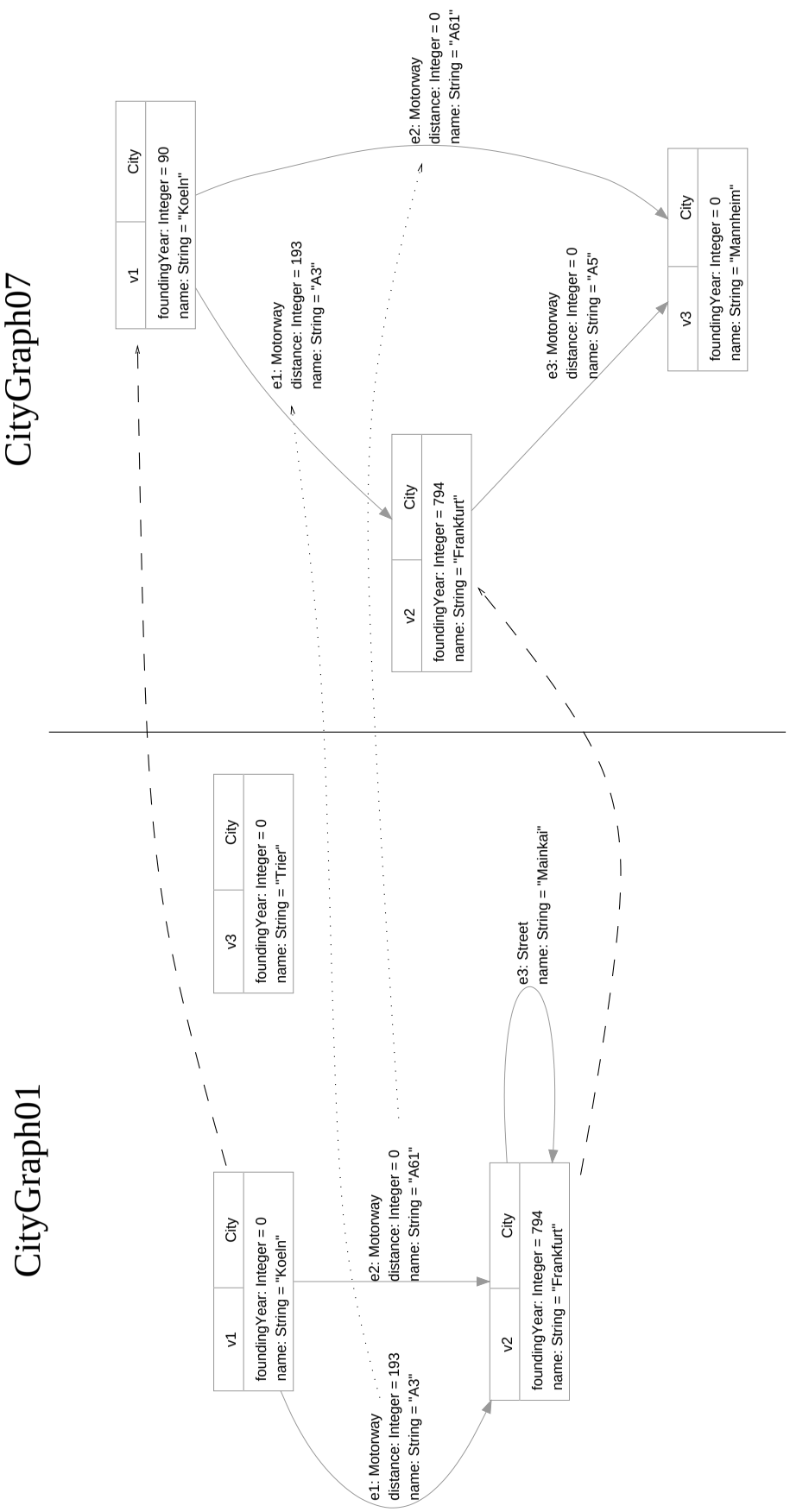


Abbildung 2.9.: Beispiel für ein Mapping zweier TGraphen

des Mapping-Problems ist ihre Komplexität nicht nur von Vorteil. Die Suche nach einem passenden Mapping-Partner für ein Element kann sehr schwierig sein. Nicht nur die Eigenschaften der Elemente, wie deren Typ und Attribute, sind von Bedeutung. Auch können die Beziehungen zwischen Elementen einen wichtigen Indikator darstellen. Letzlich muss der Entwickler eines Verfahrens zur Lösung des Mapping-Problems anhand der gestellten Anforderungen geeignete Entwurfsentscheidungen treffen.

Ein Beispiel für ein Mapping zeigt Abbildung 2.9. Hierbei werden die Graphenelemente der beiden TGraphen CityGraph01 und CityGraph07 einander zugeordnet. Das Mapping wird dargestellt durch gestrichelte und gepunktete Pfeile. Ein gestrichelter Pfeil zeigt die Zuordnung zweier Knoten. Ein gepunkteter Pfeil repräsentiert die Zuordnung zweier Kanten.

Umkehrung von TGraph-Mappings

Analog zu der Umkehrung von TGraph-Morphismen kann die Umkehrung zu Mappings definiert werden. Da es sich bei einem Mapping um einen partiellen, injektiven TGraph-Morphismus handelt, existiert zu jedem Mapping $M : TG_1 \nrightarrow TG_2$, mit $m_V : V1 \nrightarrow V2$ und $m_E : E1 \nrightarrow E2$, dessen **Umkehr-Mapping** $M^{-1} : TG_2 \nrightarrow TG_1$, mit $m_V^{-1} : V2 \nrightarrow V1$ und $m_E^{-1} : E2 \nrightarrow E1$. Für ein Umkehr-Mapping gilt:

$$\forall v1 \in V1, \forall v2 \in V2 : m_V^{-1}(v2) = v1 \Leftrightarrow m_V(v1) = v2\}$$

und

$$\forall e1 \in E1, \forall e2 \in E2 : m_E^{-1}(e2) = e1 \Leftrightarrow m_E(e1) = e2\}$$

2.4.3. Zusammenhang zwischen dem Mapping und Vergleichsverfahren

Das Mapping zwischen zwei TGraphen bildet die Grundlage zur Identifizierung und Klassifizierung von Abweichungen. **Verfahren zur Lösung des Mapping-Problems sind ein zentraler Baustein von Vergleichsverfahren.** Sie haben einen großen Einfluss auf die qualitativen und quantitativen Eigenschaften von Vergleichsverfahren.

Ein Mapping enthält implizit Informationen über Abweichungen zweier TGraphen. Ein Vergleichsverfahren verwendet ein Mapping, um diese Informationen zu sammeln und in einer geeigneten Form zu repräsentieren. Die Ausgabe eines Vergleichsverfahrens enthält explizite Informationen über Abweichungen und stellt diese in einer geeigneten Form dar.

Anhand des in Abbildung 2.9 dargestellten Mappings kann der Zusammenhang zwischen den Informationen des Mapping und den Abweichungen zweier TGraphen veranschaulicht werden. So wird beispielsweise der Knoten $v3$ des TGraphen CityGraph01 keinem Knoten des TGraphen CityGraph07 zugeordnet. Dies deutet auf eine Abweichung zwischen beiden Graphen hin. Auch zu der Kante $e3$ des TGraphen CityGraph01 gibt es keine Entsprechung im TGraphen CityGraph07. In CityGraph07 gibt es zu der Kante $e3$ und dem Knoten $v3$ keine Entsprechung in CityGraph01.

Das Mapping enthält weitere Informationen, die auf Abweichungen der beiden TGraphen hindeuten. Zum Beispiel wird dem Knoten $v1$ aus CityGraph01 der Knoten $v1$ in CityGraph07 durch das Mapping zugeordnet. Jedoch unterscheiden sich beide Knoten bezüglich ihrer Attribut-Werte.

Eine weitere Form der Abweichung ist eine Veränderung der Inzidenzen. Diese manifestieren sich anhand von Abweichungen der Startknoten oder Zielknoten, der durch das Mapping einander zugeordneten Kanten. In dem Beispiel in Abbildung 2.9 wird beispielsweise der Kante $e2$ aus CityGraph01 die Kante $e2$ aus CityGraph07 zugeordnet. Der Startknoten von $e2$ aus CityGraph01 wird durch das Mapping dem Startknoten von $e2$ aus CityGraph07 zugeordnet. Dies trifft jedoch nicht für die Zielknoten der beiden Kanten zu. Der Zielknoten von $e2$ aus CityGraph01 wird durch das Mapping nicht dem Zielknoten von $e2$ in CityGraph07 zugeordnet. Dies bedeutet, dass sich beide Kanten bezüglich ihres Zielknotens unterscheiden.

2.4.4. Das Mapping im Fallbeispiel

Die beiden TGraphen des Fallbeispiels könnten, wie in Abbildung 2.10 dargestellt, durch ein Mapping einander zugeordnet werden. Dieses Mapping wurde frei gewählt. Wie die Abbildung verdeutlicht, kann das Ergebnis bereits für TGraphen mit geringer Anzahl an Knoten und Kanten sehr unübersichtlich werden.

Bei zunehmender Anzahl der Knoten und Kanten ist eine derartige Veranschaulichung des Mappings für den Betrachter verwirrend und nicht intuitiv verständlich.

Aus diesem Grund ist das gleiche Mapping nochmals in Abbildung 2.11 dargestellt. In diesem Fall zeigt es direkt anhand der Diagramme das Ergebnis des Mappings. Diese Abbildung soll verdeutlichen, welchen inhaltlichen Wert ein Mapping für ein Vergleichsverfahren darstellt.

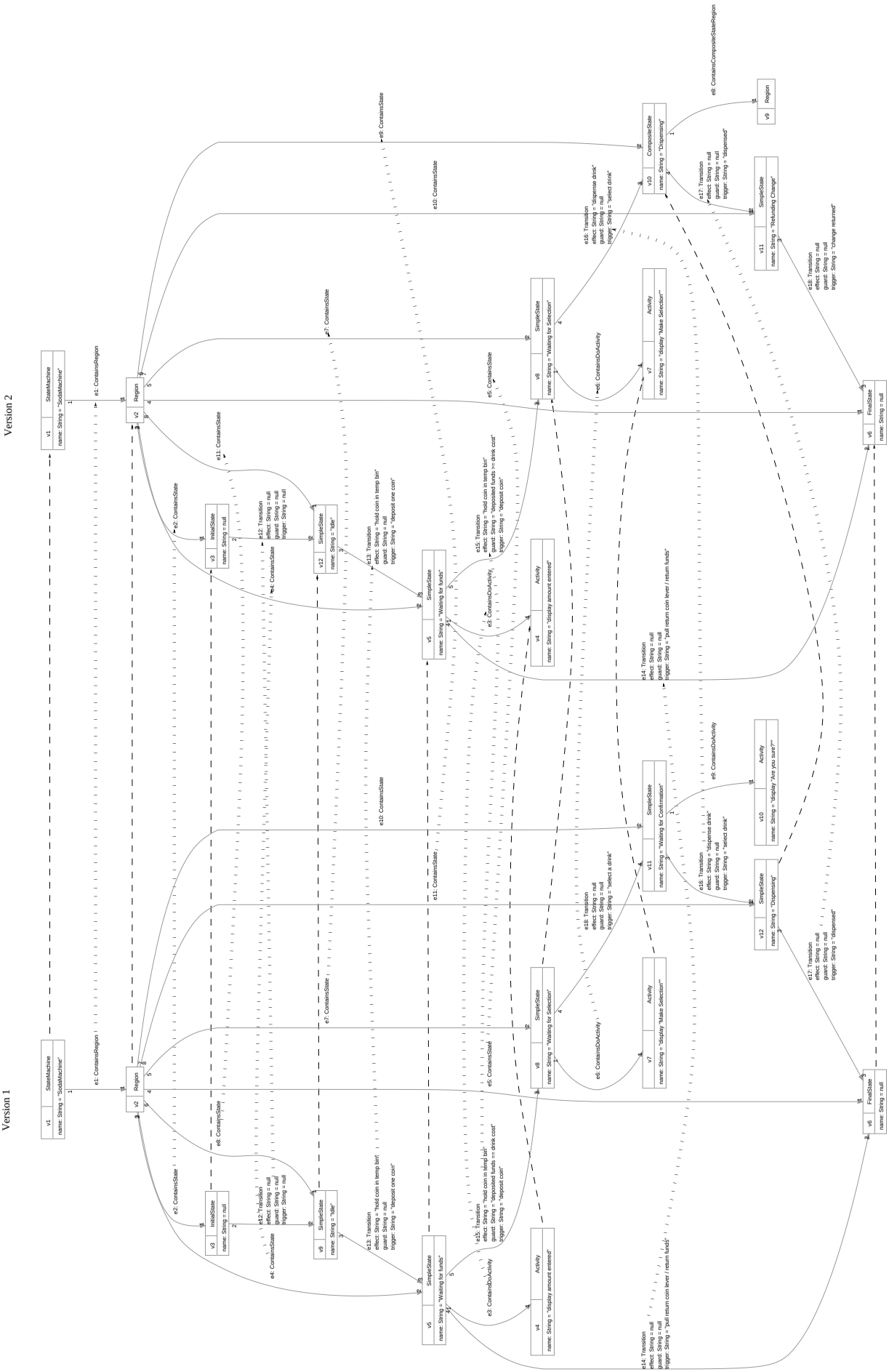


Abbildung 2.10.: Beispiel für ein Mapping zweier TGraphen

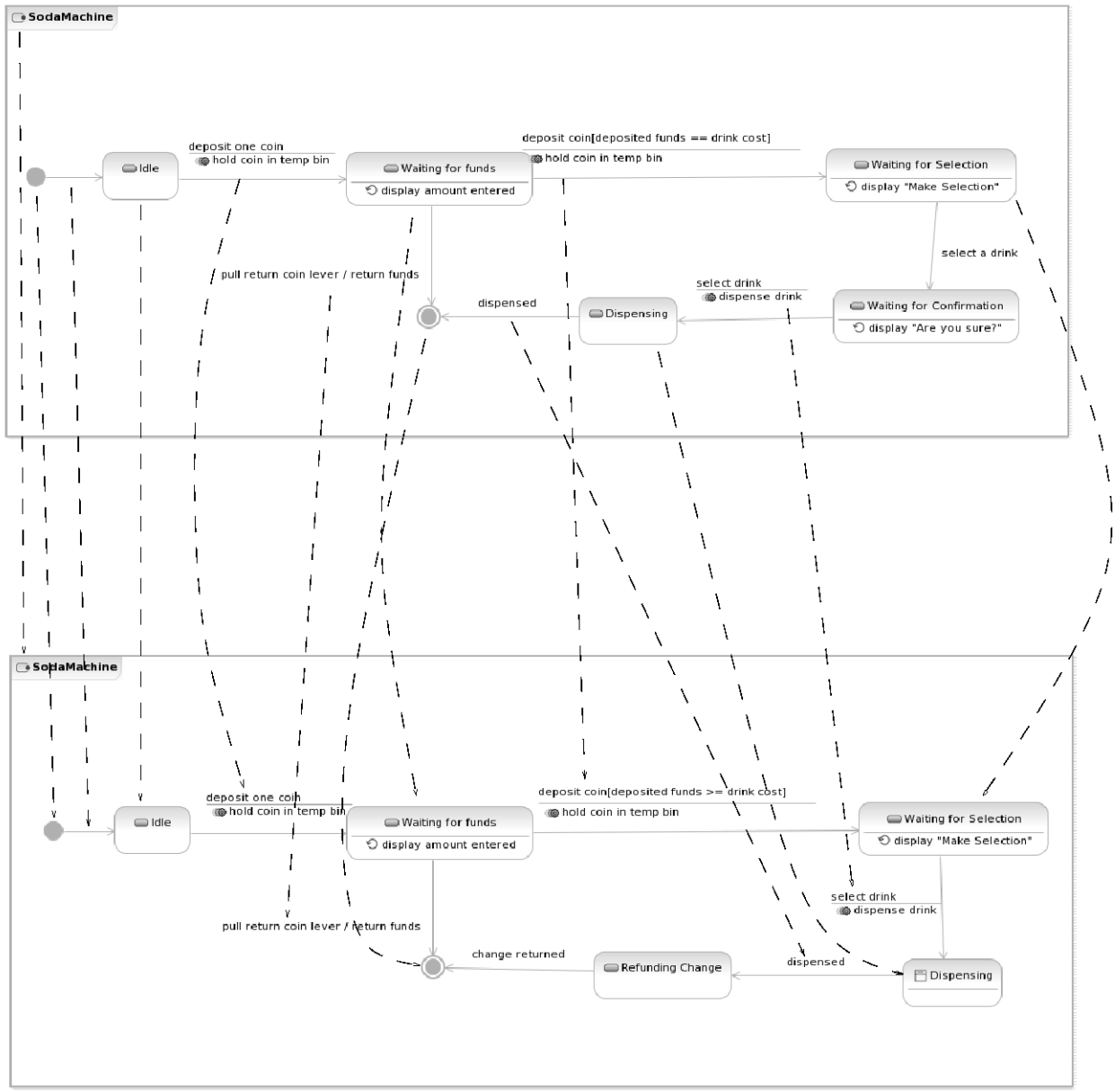


Abbildung 2.11.: Beispiel für das Mapping der TGraphen aus Abbildung 2.10, übertragen auf die repräsentierten Statechart-Diagramme des Fallbeispiels

2.5. Die Differenz von TGraphen

Um den Begriff der Differenz von TGraphen genauer zu erfassen, ist eine formale Definition notwendig. Diese soll es ermöglichen den Begriff der Differenz klar von anderen Begriffen, wie z.B. Delta, abzugrenzen. **Die Begriffe Differenz und Delta sind nicht gleichwertig.** Dennoch werden sie oftmals synonym verwendet. Eine genaue Definition und Abgrenzung beider Begriffe ist erforderlich.

Die **Differenz** ist die **Menge aller inhaltlichen Abweichungen** zweier TGraphen TG_1 und TG_2. Ein **Delta** ist eine **Sequenz von Transformationsanweisungen**, die von einem Verfahren genutzt werden können, um aus TG_1 einen TGraphen TG_2' zu erzeugen, der TG_2 entspricht.

Die Differenz zweier TGraphen wird durch ein **Vergleichsverfahren** berechnet. Es erhält als Eingabe zwei TGraphen und erzeugt als Ausgabe die Differenz.
Sei *TGraph* die Menge aller TGraphen und sei *Difference* die Menge aller TGraph-Differenzen. Für zwei TGraphen bestimmt die Abbildung

$$compare : TGraph \times TGraph \rightarrow Difference$$

die Differenz beider TGraphen.

Um die beiden TGraphen voneinander unterscheiden zu können, werden sie im folgenden als der **linksseitige TGraph** TG_L und der **rechtsseitige TGraph** TG_R bezeichnet. Hierbei ist anzumerken, dass keinem der beiden TGraphen eine besondere Rolle zukommt, beide dienen als Eingabe-Argumente eines Vergleichsverfahrens.

2.5.1. Definition

Um eine formale Definition der Differenz formulieren zu können, bedarf es einer Untersuchung der **Abweichungen zwischen TGraphen**. Zwei TGraphen können sich in vielerlei Hinsicht unterscheiden. TGraphen bestehen aus Knoten und Kanten, sie sind gerichtet, attributiert, typisiert und angeordnet. In allen diesen Eigenschaften können zwei TGraphen voneinander abweichen. Dementsprechend müssen alle diese Eigenschaften bei der Definition der Differenz berücksichtigt werden.

Element-Differenz

Ein TGraph besteht aus Elementen, den Knoten und Kanten. Zwei Elemente aus zwei TGraphen können durch ein Mapping einander zugeordnet werden. Kann einem Element eines TGraphen durch ein Mapping kein Element in dem anderen TGraphen zugeordnet werden, so handelt es sich um eine Abweichung der

beiden TGraphen. Der Vergleich von zwei TGraphen muss Informationen zu **Abweichungen der Elemente** liefern.

Definition 11 *Element-Differenz*

Seien

- $TG_L = (V_{LSeq}, E_{LSeq}, \Lambda_{LSeq}, type_L, value_L)$ *und*
- $TG_R = (V_{RSeq}, E_{RSeq}, \Lambda_{RSeq}, type_R, value_R)$ *zwei TGraphen*
- $M = (m_V, m_E)$ *ein Mapping der TGraphen* TG_L *und* TG_R

Die Element-Differenz der TGraphen TG_L *und* TG_R *bezüglich M ist ein Tupel* $(V_{LU}, V_{RU}, E_{LU}, E_{RU})$ *bestehend aus:*

- *der Menge* $V_{LU} \subseteq V_L$ *aller Knoten aus* TG_L *denen kein in* TG_R *vorkommender Knoten durch das Mapping M zugeordnet wird:*

  $$V_{LU} := \{V_L \setminus dom(m_V\}$$

- *der Menge* $V_{RU} \subseteq V_R$ *aller Knoten aus* TG_R *denen kein in* TG_L *vorkommender Knoten durch das Mapping M zugeordnet wird:*

  $$V_{RU} := \{V_R \setminus ran(m_V\}$$

- *der Menge* $E_{LU} \subseteq E_L$ *aller Kanten aus* TG_L *denen keine in* TG_R *vorkommende Kante durch das Mapping M zugeordnet wird:*

  $$E_{LU} := \{E_L \setminus dom(m_E\}$$

- *der Menge* $E_{RU} \subseteq E_R$ *aller Kanten aus* TG_R *denen keine in* TG_L *vorkommende Kante durch das Mapping M zugeordnet wird:*

  $$E_{RU} := \{E_R \setminus ran(m_E\}$$

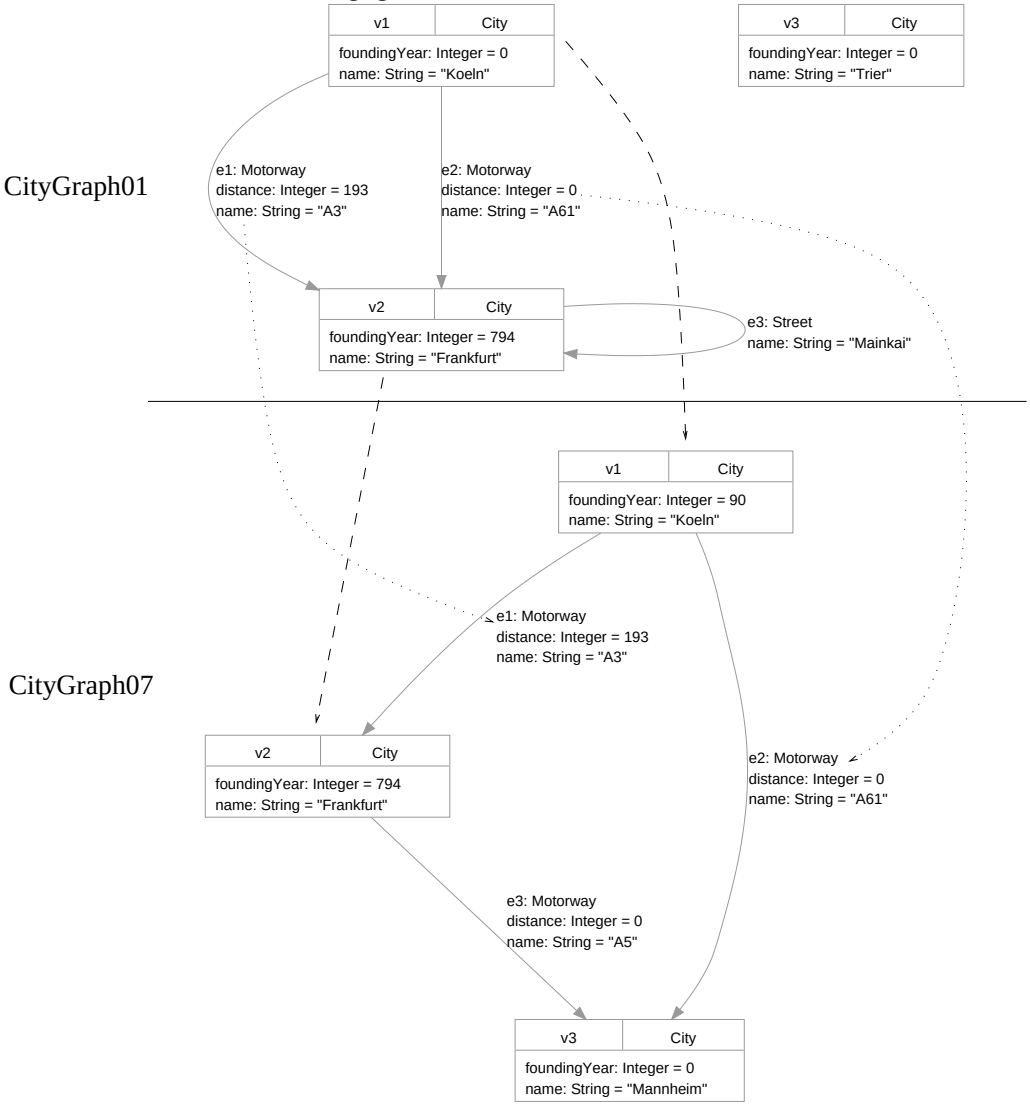
Der in der Definition verwendete Index *U* steht hierbei für 'Unmatched' und deutet darauf hin, dass alle in der jeweiligen Menge enthaltenen Elemente nicht im Mapping enthalten sind.

Die Knoten und Kanten eines TGraphen repräsentieren Informationen. Wenn zwei TGraphen in Bezug auf ihre Elemente voneinander abweichen, so ist dies ein wichtiger Hinweis auf inhaltliche Unterschiede.

Die Element-Differenz basiert direkt auf dem Mapping. Die Qualität des verwendeten Mappingverfahrens hat somit einen großen Einfluss auf die, durch ein Vergleichsverfahren berechnete, Element-Differenz.

Wie das folgende Beispiel zeigt, ist es sehr leicht die Element-Differenz für zwei TGrahen und deren Mapping zu bestimmen.

Beispiel 2 *Seien die beiden abgebildeten TGraphen und das eingezeichnete Mapping* $M = (m_V, m_E)$ *mit* $m_V = \{v1 \mapsto v1, v2 \mapsto v2\}$ *und* $m_E = \{e1 \mapsto e1, e2 \mapsto e2\}$ *gegeben:*



Für die Element-Differenz der beiden TGraphen gilt:

- $V_{LU} = \{v3\}$
- $V_{RU} = \{v3\}$
- $E_{LU} = \{e3\}$
- $E_{RU} = \{e3\}$

Inzidenz-Differenz

Da TGraphen gerichtet sind, kann jeder Kante eindeutig ein Startknoten und ein Zielknoten zugeordnet werden. Eine Kante ist positiv inzident zu ihrem Startknoten und negativ inzident zu ihrem Zielknoten. Zwischen zwei TGraphen können diese Zuordnungen abweichen. Die **Abweichungen der Inzidenzen** müssen auch durch ein Vergleichsverfahren erfasst werden.

Definition 12 *Inzidenz-Differenz*

Seien

- $TG_L = (V_{LSeq}, E_{LSeq}, \Lambda_{LSeq}, type_L, value_L)$ *und*
- $TG_R = (V_{RSeq}, E_{RSeq}, \Lambda_{RSeq}, type_R, value_R)$ *zwei TGraphen*
- $M = (m_V, m_E)$ *ein Mapping der TGraphen* TG_L *und* TG_R

Die Inzidenz-Differenz der TGraphen TG_L *und* TG_R *bezüglich M ist ein Tupel* $(\alpha_{diff}, \omega_{diff})$ *bestehend aus:*

- *der Abbildung* $\alpha_{diff} : E_L \nrightarrow E_R$ *aller durch das Mapping einander zugeordneter Kanten, deren Startknoten nicht durch das Mapping einander zugeordnet werden. Es gilt:*

$$\forall e_1 \in E_L, \forall e_2 \in E_R : e_1 \mapsto e_2 \in \alpha_{diff} \Leftrightarrow m_E(e_1) = e_2 \wedge m_V(\alpha_L(e_1)) \neq \alpha_R(e_2)$$

- *der Abbildung* $\omega_{diff} : E_L \nrightarrow E_R$ *aller durch das Mapping einander zugeordneter Kanten, deren Zielknoten nicht durch das Mapping einander zugeordnet werden. Es gilt:*

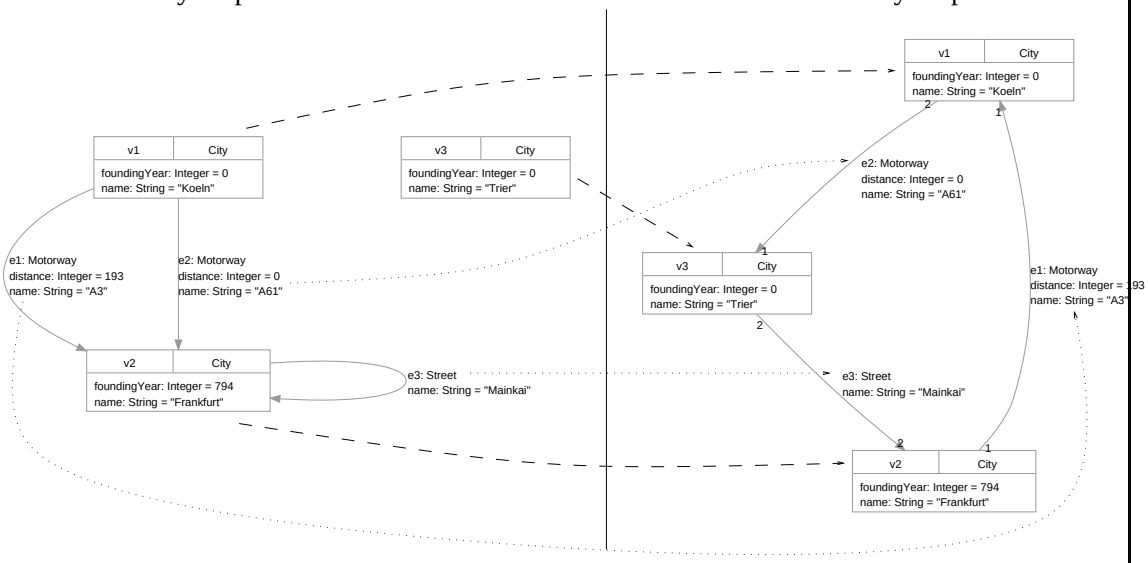
$$\forall e_1 \in E_L, \forall e_2 \in E_R : e_1 \mapsto e_2 \in \omega_{diff} \Leftrightarrow m_E(e_1) = e_2 \wedge m_V(\omega_L(e_1)) \neq \omega_R(e_2)$$

Zu den wichtigsten Informationen eines TGraphen zählen die Beziehungen zwischen Knoten. Diese werden durch gerichtete Kanten repräsentiert. Weichen diese Beziehungen zwischen zwei TGraphen voneinander ab, so wird dies durch die Inzidenz-Differenz ausgedrückt. Sie liefert dem Anwender wichtige Informationen beim Vergleich zweier TGraphen.

Wie in dem folgenden Beispiel zu sehen ist, können derartige Abweichungen auch dann auftreten, wenn es keine Abweichungen bezüglich der Elemente gibt. Zu jedem Knoten und jeder Kante der beiden TGraphen gibt es einen Mapping-Partner. Dennoch weichen beide TGraphen voneinander ab. Die Beziehungen zwischen den Elementen sind in beiden TGraphen unterschiedlich.

Beispiel 3 *Seien die beiden abgebildeten TGraphen und das Mapping* $M = (m_V, m_E)$ *mit* $m_V = \{v1 \mapsto v1, v2 \mapsto v2, v3 \mapsto v3\}$ *und* $m_E = \{e1 \mapsto e1, e2 \mapsto e2, e3 \mapsto e3\}$ *gegeben:*

CityGraph01 CityGraph02



Für die Inzidenz-Differenz der beiden TGraphen gilt:

- $\alpha_{diff} = \{e1 \mapsto e1, e3 \mapsto e3\}$
- $\omega_{diff} = \{e1 \mapsto e1, e2 \mapsto e2\}$

Typ-Differenz

TGraphen sind typisiert. Jedem Element wird ein Typ zugewiesen. Es ist nicht ausgeschlossen, dass zwei durch das Mapping einander zugeordnete Elemente in Bezug auf ihren Typ voneinander abweichen. Die Differenz beinhaltet Informationen zu **Abweichungen in der Typisierung**.

Definition 13 *Typ-Differenz*

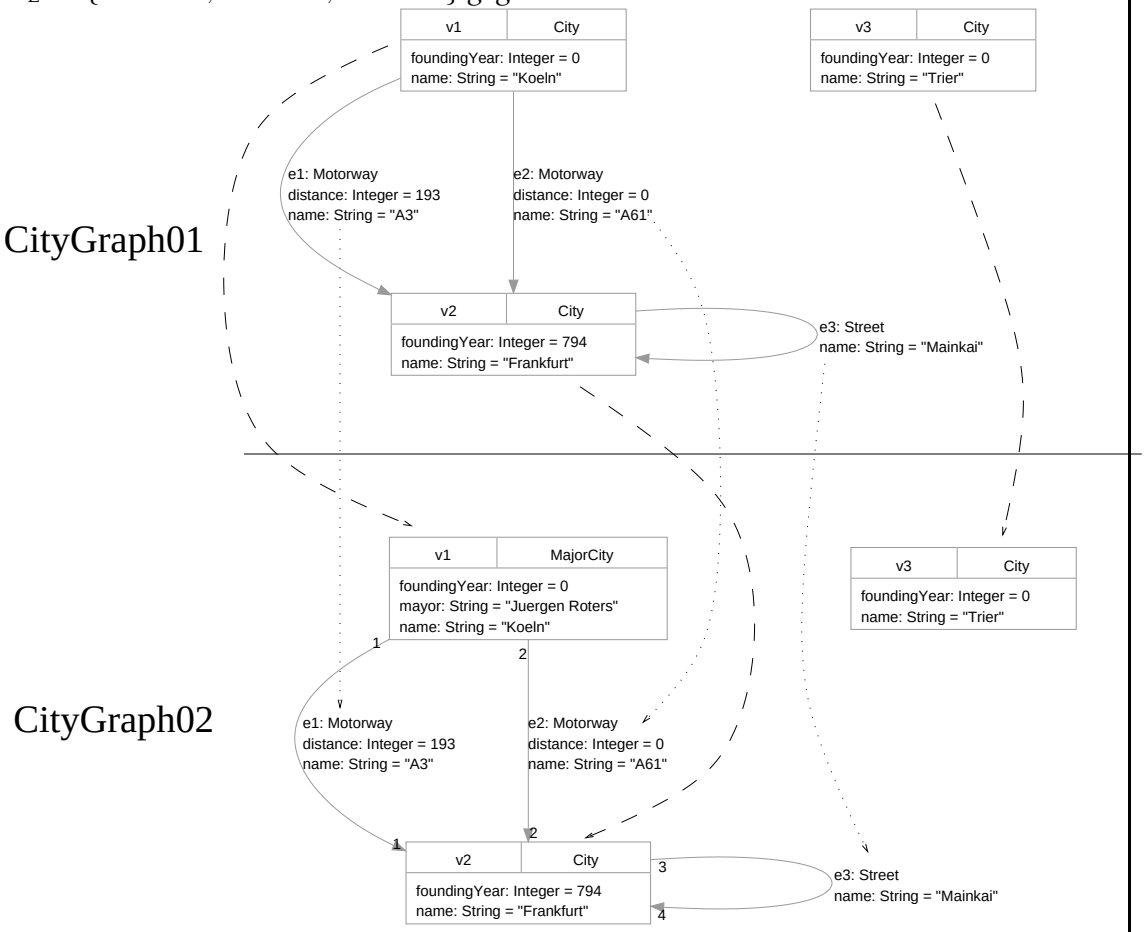
Seien

- $TG_L = (V_{LSeq}, E_{LSeq}, \Lambda_{LSeq}, type_L, value_L)$ *und*
- $TG_R = (V_{RSeq}, E_{RSeq}, \Lambda_{RSeq}, type_R, value_R)$ *zwei TGraphen*
- $M = (m_V, m_E)$ *ein Mapping der TGraphen* TG_L *und* TG_R

Die Typ-Differenz der TGraphen TG_L *und* TG_R *bezüglich M ist eine Abbildung* $type_{diff} : (V_L \cup E_L) \nrightarrow (V_R \cup E_R)$*, die jedes Graphenelement aus* TG_L *auf das durch das Mapping M zugeordnete Element aus* TG_R *abbildet, wenn sich beide Elemente bezüglich der Typisierung unterscheiden. Es gilt:*

$$\forall x \in (V_L \cup E_L), \forall y \in (V_R \cup E_R) :$$
$$x \mapsto y \in type_{diff} \Leftrightarrow M(x) = y \wedge type_L(x) \neq type_R(y)$$

Beispiel 4 *Seien die beiden abgebildeten TGraphen und das Mapping* $M = (m_V, m_E)$ *mit* $m_V = \{v1 \mapsto v1, v2 \mapsto v2, v3 \mapsto v3\}$ *und* $m_E = \{e1 \mapsto e1, e2 \mapsto e2, e3 \mapsto e3\}$ *gegeben:*



Für die Typ-Differenz der beiden TGraphen gilt:

$$type_{diff} = \{v1 \mapsto v1\}$$

Wie das Beispiel zeigt, können zwei TGraphen voneinander abweichen, obwohl es keinerlei Element-Differenz und keine Abweichungen der Inzidenzen gibt. Die beiden Knoten $v1$ aus $CityGraph01$ und $v1$ aus $CityGraph02$ werden durch das Mapping einander zugeordnet. Sie weichen in Bezug auf ihre Typisierung voneinander ab.

Zudem enthält $v1$ in $CityGraph02$ ein zusätzliches Attribut. Derartige Abweichungen der Attributierung werden durch die Attribut-Differenz erfasst. Sie sind nicht Bestandteil der Typ-Differenz.

Attribut-Differenz

Eine weitere Eigenschaft von TGraphen ist die Attributierung von Elementen. Sowohl den Knoten als auch den Kanten eines TGraphen können Attribut-Werte

Paare zugewiesen werden. **Abweichungen der Attribute** treten bei einer **Abweichung in der Attributierung** oder bei **Abweichungen der Attributwerte** auf.

Definition 14 *Attribut-Differenz*

Seien

- $TG_L = (V_{LSeq}, E_{LSeq}, \Lambda_{LSeq}, type_L, value_L)$ *und*
- $TG_R = (V_{RSeq}, E_{RSeq}, \Lambda_{RSeq}, type_R, value_R)$ *zwei TGraphen*
- $M = (m_V, m_E)$ *ein Mapping der TGraphen* TG_L *und* TG_R

Die Attribut-Differenz der TGraphen TG_L *und* TG_R *bezüglich M ist ein Tupel* $(value_{diff}, attrL_{diff}, attrR_{diff})$ *bestehend aus:*

- *einer Abbildung* $value_{diff} : (V_L \cup E_L) \times AttrId \times Value \nrightarrow (V_R \cup E_R) \times AttrId \times Value$ *die alle Abweichungen der Attributwerte erfasst. Es gilt:*

$$\begin{aligned} &\forall x \in (V_L \cup E_L), \forall y \in (V_R \cup E_R), \forall a \in AttrId : \\ &(x, a, value(x)(a)) \mapsto (y, a, value(y)(a)) \in value_{diff} \\ &\Leftrightarrow M(x) = y \wedge value(x)(a) \neq value(y)(a) \end{aligned}$$

- *einer Abbildung* $attrL_{diff} : (V_L \cup E_L) \nrightarrow AttrId \times Value$ *die alle Abweichungen der Attributierung des linksseitigen TGraphen erfasst. Es gilt:*

$$\begin{aligned} &\forall x \in (V_L \cup E_L), \forall y \in (V_R \cup E_R), \forall a \in AttrId : \\ &x \mapsto (a, value(x)(a)) \in attrL_{diff} \\ &\Leftrightarrow M(x) = y \wedge a \notin dom(value(y)) \end{aligned}$$

- *einer Abbildung* $attrR_{diff} : (V_R \cup E_R) \nrightarrow AttrId \times Value$ *die alle Abweichungen der Attributierung des rechtsseitigen TGraphen erfasst. Es gilt:*

$$\begin{aligned} &\forall x \in (V_L \cup E_L), \forall y \in (V_R \cup E_R), \forall a \in AttrId : \\ &y \mapsto (a, value(y)(a)) \in attrR_{diff} \\ &\Leftrightarrow M(x) = y \wedge a \notin dom(value(x)) \end{aligned}$$

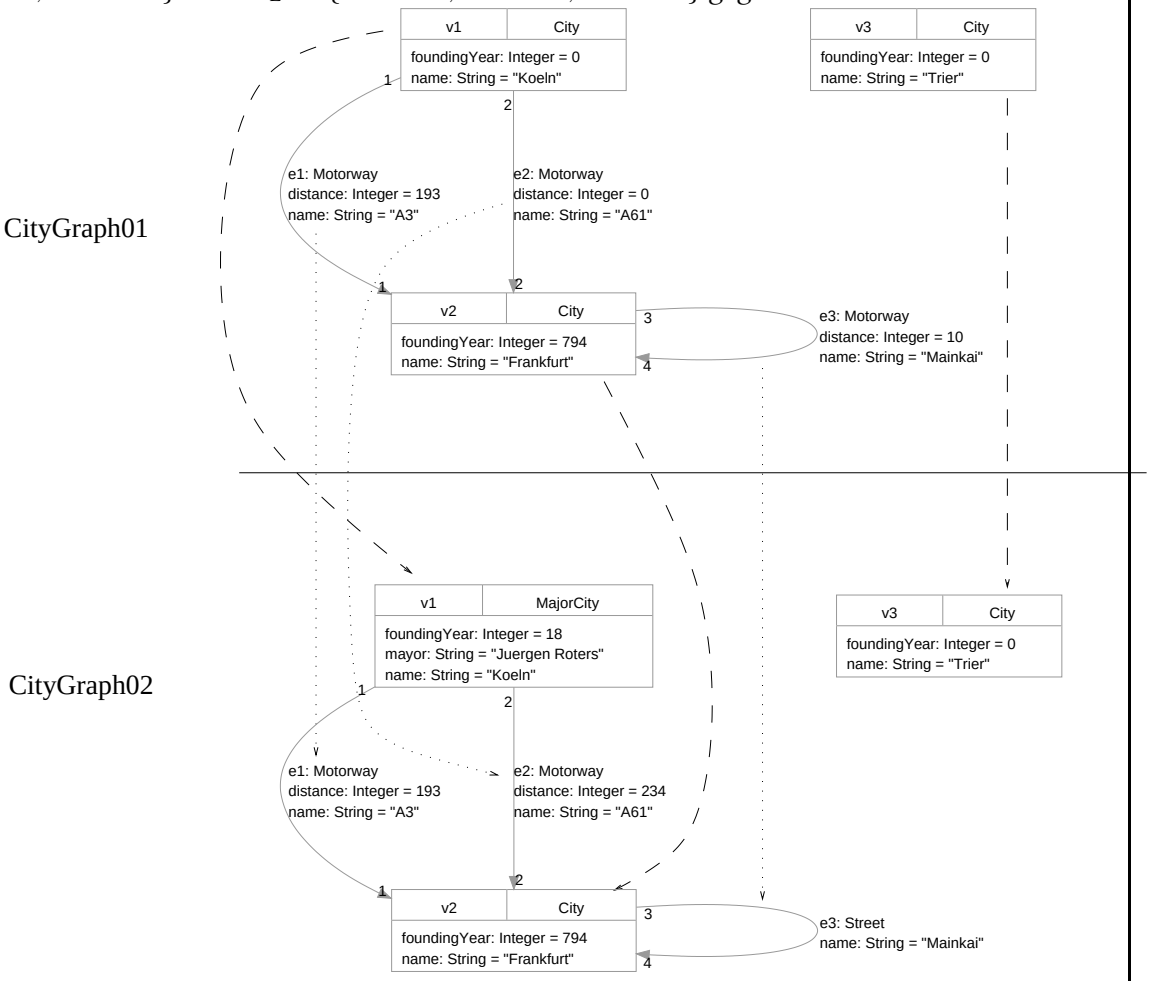
Eine **Abweichung der Attribut-Werte** liegt vor, wenn zwei durch das Mapping einander zugeordnete Elemente das gleiche Attribut mit unterschiedlichen Werten beinhalten. Derartige Abweichungen werden durch die Abbildung $value_{diff}$ erfasst.

Eine **Abweichung der Attributierung** liegt vor, wenn ein Element x ein Attribut enthält, welches nicht in dem durch das Mapping zugeordneten Element y enthalten ist. Ist x ein Element von TG_L so wird die Abweichung durch die Ab-

bildung $attrL_{diff}$ erfasst. Ist x ein Element von TG_R so wird die Abweichung durch die Abbildung $attrR_{diff}$ erfasst.

Mit Hilfe von Attribut-Werte Paaren können Knoten und Kanten detaillierte Informationen enthalten. Beim Vergleich zweier TGraphen stellt eine Abweichung eines Attributwertes eine wichtige Information für den Anwender dar. Kleine Abweichungen an Knoten oder Kanten werden durch die Attribut-Differenz explizit sichtbar.

Beispiel 5 *Seien die beiden abgebildeten TGraphen CityGraph01 und CityGraph02, und das Mapping* $M = (m_V, m_E)$ *mit* $m_V = \{v1 \mapsto v1, v2 \mapsto v2, v3 \mapsto v3\}$ *und* $m_E = \{e1 \mapsto e1, e2 \mapsto e2, e3 \mapsto e3\}$ *gegeben:*



Für die Attribut-Differenz der beiden TGraphen gilt:

$$value_{diff} = \{(v1, foundingYear, 0) \mapsto (v1, foundingYear, 18), \\ (e2, distance, 0) \mapsto (e2, distance, 234)\}$$

$$attrL_{diff} = \{e3 \mapsto (distance, 10)\}$$

$$attrR_{diff} = \{v1 \mapsto (mayor, JuergenRoters)\}$$

Anordnungs-Differenz

TGraphen sind angeordnet. Diese Eigenschaft kann in drei Teileigenschaften gegliedert werden. Die Anordnung der Knoten, die Anordnung der Kanten und die Anordnung der Inzidenzen. Die **Abweichungen der Anordnung** zweier TGraphen besteht aus den Abweichungen aller drei Teileigenschaften. Diese drei Teileigenschaften können getrennt voneinander betrachtet werden.

Für die folgende Definition sei hier angemerkt, dass für eine Sequenz S mit $x_i \in S$ das Element x an der Stelle $i \in \mathbb{N}$ angegeben wird. Eine Teilsequenz von S entsteht, indem eine beliebige Menge von Elementen weggelassen wird, ohne ihre Anordnung zu verändern. Mit $\langle x_i, \ldots, x_j \rangle \in S;\ i, j \in \mathbb{N}$ wird eine Teilsequenz von S bezeichnet.

Die globale Anordnung der Kanten, die globale Anordnung der Knoten und die Inzidenzanordnungen werden jeweils durch Sequenzen repräsentiert. Die **Abweichungen zweier Sequenzen** können unterschiedlich beschrieben werden.

Sie können **absolut** erfasst werden, indem man die Elemente hinsichtlich ihrer absoluten Position innerhalb der Sequenzen betrachtet. Wenn sich z.B. ein Element x in der Sequenz S_L an der Position $i \in \mathbb{N}$ befindet und in der Sequenz S_R an der Position $j \in \mathbb{N}$ und $i \neq j$ gilt, so würde das Tupel (x, i, j) die Abweichung der Position absolut beschreiben.

Abweichungen können aber auch **relativ** beschrieben werden. In diesem Fall werden die Elemente hinsichtlich ihrer Vorgänger betrachtet und nicht ihre absolute Anordnung innerhalb der Sequenzen. Wenn z.B. ein Element x in der Sequenz S_L an der Position $i \in \mathbb{N}$ und in S_R an der Position $j \in \mathbb{N}$ vorkommt und es gilt $S_L[i-1] \neq S_R[j-1]$, dann ist dies eine relative Abweichung der Anordnung. Diese Abweichung kann durch das Tupel $(x, S_L[i-1], S_R[j-1])$ beschrieben werden.

Die Abweichungen der Sequenzen $S_L = \langle a, b, c, d \rangle$ und $S_R = \langle b, c, a, d \rangle$ können durch die Menge $\{(a, 1, 3), (b, 2, 1), (c, 3, 2)\}$ absolut beschrieben werden. Im Gegensatz dazu genügt das Tupel $(a, \perp, c)$, um die Abweichungen relativ zu beschreiben. Alle anderen Abweichungen, z.B. das Tupel (d, c, a) werden nicht benötigt, da sie implizit in der Information der relativen Veränderung von a enthalten sind. Hierbei wird mit $\perp$ ausgedrückt, dass der Vorgänger eines Elements undefiniert ist, dies bedeutet implizit, dass sich das Element am Anfang der Sequenz befindet.

Die relative Beschreibung der Abweichungen zweier Sequenzen ist gegenüber der absoluten Beschreibung effizienter, wie das Beispiel zeigt. Dieser Vorteil soll ausgenutzt werden, um die Abweichungen der Anordnung zweier TGraphen möglichst kompakt zu beschreiben.

Definition 15 *Anordnungs-Differenz*

Seien

- $TG_L = (V_{LSeq}, E_{LSeq}, \Lambda_{LSeq}, type_L, value_L)$ *und*
- $TG_R = (V_{RSeq}, E_{RSeq}, \Lambda_{RSeq}, type_R, value_R)$ *zwei TGraphen*
- $M = (m_V, m_E)$ *ein Mapping der TGraphen* TG_L *und* TG_R

Die Anordnungs-Differenz der TGraphen TG_L *und* TG_R *ist ein Tupel* $(V_{SeqDiff}, E_{SeqDiff}, \Lambda_{SeqDiff})$ *bestehend aus:*

- *einer Abbildung* $V_{SeqDiff} : V_{LSeq} \nrightarrow V_R$*, die alle Abweichungen der globalen Anordnung der Knoten erfasst. Es gilt:*

$$V_{SeqDiff} := \{\langle v_i, .., v_j\rangle \mapsto v_k \in V_{LSeq} \times V_R \mid i, j, k \in \mathbb{N} \wedge m_V(v_i) = v_{k+1} \wedge m_V(v_{i-1}) \neq v_k\}$$

- *einer Abbildung* $E_{SeqDiff} : E_{LSeq} \nrightarrow E_R$*, die alle Abweichungen der globalen Anordnung der Kanten erfasst. Es gilt:*

$$E_{SeqDiff} := \{\langle e_i, .., e_j\rangle \mapsto e_k \in E_{LSeq} \times E_R \mid i, j, k \in \mathbb{N} \wedge m_E(e_i) = e_{k+1} \wedge m_E(e_{i-1}) \neq e_k\}$$

- *einer Abbildung* $\Lambda_{SeqDiff} : V_L \times \Lambda_{LSeq} \nrightarrow V_R \times E_R \times \{in, out\}$ *die alle Abweichungen der Inzidenz-Anordnung erfasst. Es gilt:*

$$\Lambda_{SeqDiff} := \{v_1, \langle (e_i, d_i), .., (e_j, d_j)\rangle \mapsto v_2, e_k, d_k \in V_L \times \Lambda_{LSeq} \nrightarrow V_R \times E_R \times \{in, out\} \mid i, j, k \in \mathbb{N}, d \in \{in, out\} \; m_V(v_1) = v_2 \wedge m_E(e_i) = e_{k+1} \wedge m_E(e_{i-1}) \neq e_k\}$$

Damit die Anordnungs-Differenz keine überflüssigen Elemente enthält, gelten folgende Bedingungen:

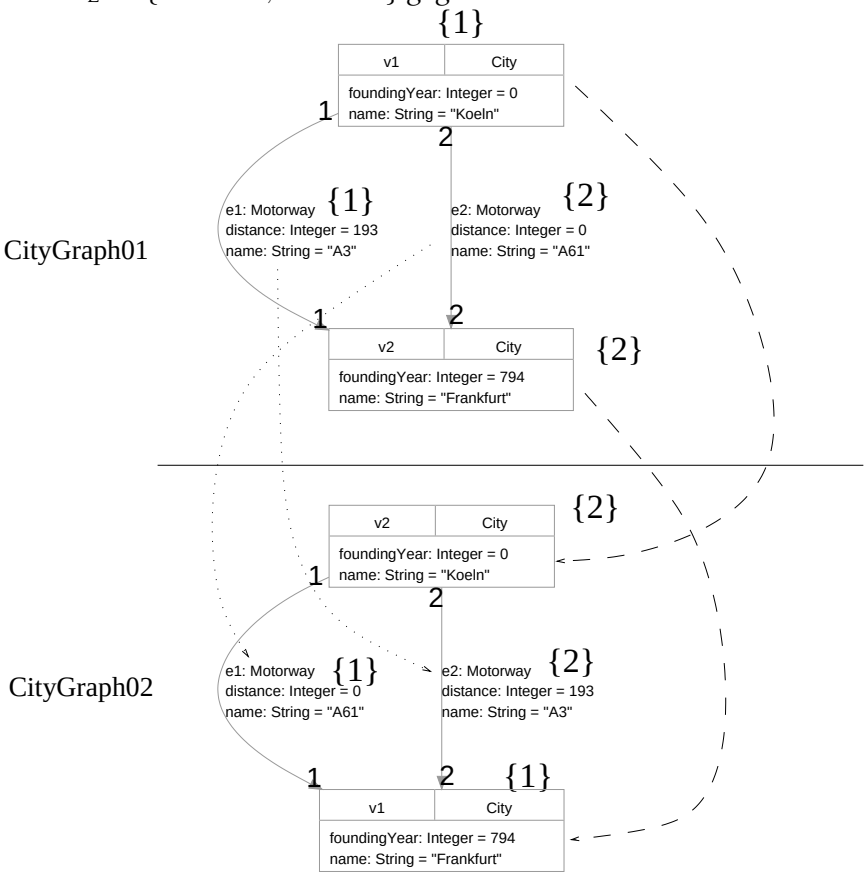
- Kommt ein Knoten v in einer Sequenz des Definitionsbereichs von $V_{SeqDiff}$ vor, so sollte sein Mapping-Partner $m_V(v)$ nicht im Wertebereich von $V_{SeqDiff}$ vorkommen. Dadurch wird verhindert, dass die Abweichung der Anordnung von v, bzw. $m_V(v)$ durch die Anordnungsdifferenz doppelt beschrieben wird.
  $\forall\, v \in x = \langle v_i, \ldots, v_j\rangle \in \mathrm{dom}(V_{SeqDiff}) : m_V(v) \notin \mathrm{ran}(V_{SeqDiff})$

- Kommt eine Kante e in einer Sequenz des Definitionsbereichs von $E_{SeqDiff}$ vor, so sollte ihr Mapping-Partner $m_E(e)$ nicht im Wertebereich von $E_{SeqDiff}$ vorkommen. Dadurch wird verhindert, dass die Abweichung der Anordnung von e, bzw. $m_E(e)$ durch die Anordnungs-Differenz doppelt beschrieben wird.
  $\forall\, e \in x = \langle e_i, \ldots, e_j\rangle \in \mathrm{dom}(E_{SeqDiff}) : m_E(e) \notin \mathrm{ran}(E_{SeqDiff})$

- Kommt eine Kante e in einer Sequenz eines Tupels $v, \langle e_i, \ldots, e_j \rangle$ des Definitionsbereichs von $\Lambda_{SeqDiff}$ vor, so sollte ihr Mapping-Partner $m_E(e)$ nicht in einem Tupel im Wertebereich von $\Lambda_{SeqDiff}$ vorkommen. Dadurch wird verhindert, dass die Abweichung der Inzidenzen eines Knotens v, bzw. $m_V(v)$ durch die Anordnungsdifferenz doppelt beschrieben wird.

In dem folgenden Beispiel sind zwei fast identische TGraphen abgebildet. Sie stimmen in Bezug auf ihre Elemente, deren Typisierung und Attributierung überein. Auch gibt es keinerlei Abweichungen hinsichtlich der Beziehungen zwischen den Elementen. Dennoch gibt es Abweichungen. Die Knoten, Kanten und auch die Inzidenzen sind in beiden TGraphen unterschiedlich angeordnet.

Beispiel 6 *Seien die beiden abgebildeten TGraphen CityGraph01 und CityGraph02, und das Mapping* $M = (m_V, m_E)$ *mit* $m_V = \{v1 \mapsto v2, v2 \mapsto v1\}$ *und* $m_E = \{e1 \mapsto e2, e2 \mapsto e1\}$ *gegeben:*



Für die Anordnungs-Differenz der beiden TGraphen gilt:

$$V_{SeqDiff} = \{\langle v1 \rangle \mapsto \langle v1 \rangle\}$$

$$E_{SeqDiff} = \{\langle e1 \rangle \mapsto \langle e1 \rangle\}$$

$$\Lambda_{SeqDiff} = \{(v1, \langle (e1, out) \rangle) \mapsto (v2, \langle (e1, out) \rangle), \\ (v2, \langle (e1, in) \rangle) \mapsto (v1, \langle (e1, in) \rangle)\}$$

Durch den Vergleich zweier TGraphen in Bezug auf ihre Eigenschaften erhält der Anwender Informationen über Abweichungen. In ihrer Gesamtheit werden alle durch ein Vergleichsverfahren berechneten Abweichungen als die Differenz bezeichnet.

Definition 16 *Differenz von TGraphen*

Seien

- $TG_L = (V_{LSeq}, E_{LSeq}, \Lambda_{LSeq}, type_L, value_L)$ *und*
- $TG_R = (V_{RSeq}, E_{RSeq}, \Lambda_{RSeq}, type_R, value_R)$ *zwei TGraphen*
- $M = (m_V, m_E)$ *ein Mapping der TGraphen* TG_L *und* TG_R

Die Differenz der TGraphen TG_L *und* TG_R *bezüglich M besteht aus*

- *der Element-Differenz* $(V_{LU}, V_{RU}, E_{LU}, E_{RU})$,
- *der Inzidenz-Differenz* $(\alpha_{diff}, \omega_{diff})$,
- *der Typ-Differenz* $type_{diff}$,
- *der Attribut-Differenz* $(value_{diff}, attrL_{diff}, attrR_{diff})$ *und*
- *der Anordnungs-Differenz* $(V_{SeqDiff}, E_{SeqDiff}, \Lambda_{SeqDiff})$ *der TGraphen* TG_L *und* TG_R.

Die Differenz zweier TGraphen beschreibt exakt alle Abweichungen. Alle Eigenschaften der TGraphen werden berücksichtigt.

2.6. Das TGraph-Delta

Das Delta zweier TGraphen TG_1 und TG_2 ermöglicht es, aus dem TGraphen TG_1 einen zu TG_2 äquivalenten TGraphen TG'_2 zu erzeugen. Der TGraph TG'_2 wird aus TG_1 und dem Delta erzeugt.

2.6.1. Äquivalenz von TGraphen

TGraphen können unter verschiedenen Blickwinkeln hinsichtlich ihrer Gleichwertigkeit untersucht werden. Diese werden im Folgenden betrachtet.

Element-Äquivalenz

TGraphen bestehen aus Knoten und Kanten. Existiert ein Mapping, welches jedem Element zweier TGraphen einen Mapping-Partner zuordnet, so gelten die beiden TGraphen bezüglich ihrer Elemente als gleichwertig.

Definition 17 *Element-Äquivalenz*

Zwei TGraphen TG_1 und TG_2 gelten als Element-äquivalent, geschrieben $TG_1 \equiv_{element} TG_2$, wenn ein totales, bijektives Mapping $M : TG_1 \rightarrowtail\!\!\!\!\!\rightarrow TG_2$ existiert.

Beispielsweise sind die TGraphen in Beispiel 3 auf Seite 45, Beispiel 4 auf Seite 47 und Beispiel 5 auf Seite 49 Element-äquivalent. In jedem der Beispiele wurde ein totales, bijektives Mapping der beiden TGraphen angegeben.

Die Element-Äquivalenz trifft nur die Aussage, dass es zu zwei TGraphen ein totales, bijektives Mapping gibt. Es trifft bewusst keine genaue Aussage über die Gleichwertigkeit der TGraphen in Bezug auf ihre Struktur, ihre Attributierung, ihre Typisierung oder Anordnung. Analog zur Definition der Differenz werden diese Eigenschaften getrennt betrachtet.

Äquivalenz-Relationen müssen die Eigenschaften der Reflexivität, Symmetrie und Transitivität erfüllen. Um zu zeigen, dass es sich bei der Element-Äquivalenz um eine Äquivalenzrelation auf TGraphen handelt, müssen die Eigenschaften für Äquivalenzrelationen nachgewiesen werden.

Die **Reflexivität** der Relation ist wie folgt nachweisbar. Zu jedem TGraphen TG gibt es ein Mapping $M : TG \rightarrow TG$, welches jeden Knoten und jede Kante auf sich selbst abbildet. Dieses totale Mapping ist bijektiv. Jeder TGraph ist somit zu sich selbst Element-äquivalent. Für die Menge aller TGraphen *TGraph* gilt:
$\forall\, TG \in TGraph : TG \equiv_{element} TG$

Die **Symmetrie** der Relation kann wie folgt nachgewiesen werden. Für zwei Element-äquivalente TGraphen TG_1, TG_2 und ein Mapping $M : TG_1 \rightarrowtail\!\!\!\!\!\rightarrow TG_2$ mit $M = (m_V, m_E)$ lässt sich ein Umkehr-Mapping $M^{-1} : TG_2 \rightarrowtail\!\!\!\!\!\rightarrow TG_1$ mit $M^{-1} = (m_V^{-1}, m_E^{-1})$ konstruieren. Da die TGraphen Element-äquivalent sind, muss das Mapping total und bijektiv sein. Das Umkehr-Mapping M^{-1} ist somit ebenfalls total und bijektiv. Es gilt:
$\forall\, TG_1, TG_2 \in TGraph : TG_1 \equiv_{element} TG_2 \Rightarrow TG_2 \equiv_{element} TG_1$.

Die **Transitivität** der Relation kann mit Hilfe der Komposition von TGraph-Morphismen nachgewiesen werden. Seien die TGraphen TG_1 und TG_2 Element-äquivalent. Seien ebenso die TGraphen TG_2 und $TG3$ Element-äquivalent. Dann gibt es ein totales, bijektives Mapping $M1 : TG_1 \rightarrowtail\!\!\!\!\!\rightarrow TG_2$ und ein totales, bijektives Mapping $M2 : TG_2 \rightarrowtail\!\!\!\!\!\rightarrow TG_3$ Die Komposition beider Mappings $M3 = M2 \circ M1 : TG_1 \rightarrowtail\!\!\!\!\!\rightarrow TG_3$ ergibt ein totales, bijektives Mapping $M3$. Die TGraphen TG_1 und TG_3 sind somit ebenfalls Element-äquivalent unter $M3$. Für alle TGraphen TG_1,

TG_2, TG_3 gilt:
$TG_1 \equiv_{element} TG_2 \wedge TG_2 \equiv_{element} TG_3 \Rightarrow TG_1 \equiv_{element} TG_3$.

Inzidenz-Äquivalenz

Durch die Elemente und deren Beziehungen untereinander wird die Struktur eines TGraphen bestimmt. Mit dem Begriff des TGraph-Isomorphismus kann eine weitere Äquivalenzrelation auf TGraphen festgelegt werden. Diese bestimmt ob zwei TGraphen hinsichtlich ihrer Elemente und ihrer Struktur als gleichwertig zu betrachten sind.

Definition 18 *Inzidenz-Äquivalenz*

Zwei TGraphen TG_1 und TG_2 gelten als Inzidenz-äquivalent, geschrieben $TG_1 \equiv_{incidence} TG_2$, wenn ein totales, bijektives und strukturerhaltendes Mapping $M : TG_1 \rightarrowtail\!\!\!\!\!\rightarrow TG_2$ existiert, so dass die beiden TGraphen Element-äquivalent und isomorph sind unter M.

Im Gegensatz zur Element-Äquivalenz fordert die Inzidenz-Äquivalenz nicht nur die Existenz eines totalen Mappings, sondern die beiden TGraphen müssen unter einem solchen Mapping auch isomorph sein.

Typ-Äquivalenz

In Beispiel 4 auf Seite 47 sind zwei TGraphen abgebildet, die unter dem dort definierten Mapping Inzidenz-äquivalent sind. Unter dem Blickwinkel der Typisierung ist ebenfalls eine Aussage über die Gleichwertigkeit zweier TGraphen möglich. Die beiden TGraphen aus Beispiel 4 können hinsichtlich der Typisierung nicht als gleichwertig bezeichnet werden.

Definition 19 *Typ-Äquivalenz*

Zwei TGraphen TG_1 und TG_2 gelten als Typ-äquivalent, geschrieben $TG_1 \equiv_{type} TG_2$, wenn ein totales, bijektives Mapping $M : TG_1 \rightarrowtail\!\!\!\!\!\rightarrow TG_2$ existiert, so dass die beiden TGraphen Element-äquivalent sind,$TG_1 \equiv_{element} TG_2$, und die folgenden Bedingungen erfüllt werden:

1. $\forall v1 \in V_1, \forall v2 \in V_2 : m_V(v1) = v2 \Rightarrow type(v1) = type(v2)$
2. $\forall e1 \in E_1, \forall e2 \in E_2 : m_E(e1) = e2 \Rightarrow type(e1) = type(e2)$

Die Typ-Äquivalenz stellt somit fest, ob zwei TGraphen in Bezug auf die Typisierung als gleichwertig zu betrachten sind. In Beispiel 6 auf Seite 52 sind zwei Typ-äquivalente TGraphen zusammen mit einem totalen Mapping abgebildet.

Attribut-Äquivalenz

Die Attributierung wird bei der Inzidenz-Äquivalenz und der Typ-Äquivalenz nicht berücksichtigt. Es ist möglich zwei TGraphen hinsichtlich der Attributierung ihrer Elemente auf Gleichwertigkeit hin zu untersuchen. Dabei wird von folgender Annahme ausgegangen. Es gilt:
$\forall x, y \in Vertex \cup Edge, \forall a \in AttrId$:
$a \notin \text{dom}(value(x)) \wedge a \notin \text{dom}(value(y)) \Rightarrow value(x)(a) = \perp = value(y)(a)$.
Der Attribut-Wert ist undefiniert, wenn ein Attribut nicht im Definitionsbereich der Attributierung eines Elementes liegt.

Definition 20 *Attribut-Äquivalenz*

Zwei TGraphen TG_1 und TG_2 gelten als Attribut-äquivalent, geschrieben $TG_1 \equiv_{attr} TG_2$, wenn ein totales, bijektives Mapping $M : TG_1 \rightarrowtail\!\!\!\!\!\rightarrow TG_2$ existiert, so dass die beiden TGraphen Element-äquivalent sind,$TG_1 \equiv_{element} TG_2$, und die folgenden Bedingungen erfüllt werden:

1. $\forall v1 \in V_1, \forall v2 \in V_2, \forall a \in AttrId$:
   $m_V(v1) = v2 \Rightarrow value(v1)(a) = value(v2)(a)$

2. $\forall e1 \in E_1, \forall e2 \in E_2, \forall a \in AttrId$:
   $m_E(e1) = e2 \Rightarrow value(e1)(a) = value(e2)(a)$

Sind zwei TGraphen Attribut-äquivalent, so bedeutet dies die Gleichheit aller durch ein Mapping M einander zugeordneten Elemente in Bezug auf ihre Attributierung. Dabei folgt implizit aus der Definition, wenn zwei TGraphen TG_1 und TG_2 Attribut-äquivalent sind, werden folgende Bedingungen erfüllt:

1. Zwei durch M einander zugeordnete Knoten enthalten die gleichen Attribute:
   $\forall v1 \in V_1, \forall v2 \in V_2, \forall a \in AttrId$:
   $m_V(v1) = v2 \Rightarrow (a \in \text{dom}(value(v1)) \Leftrightarrow a \in \text{dom}(value(v2)))$

2. Zwei durch M einander zugeordnete Kanten enthalten die gleichen Attribute:
   $\forall e1 \in E_1, \forall e2 \in E_2, \forall a \in AttrId$:
   $m_E(e1) = e2 \Rightarrow (a \in \text{dom}(value(e1)) \Leftrightarrow a \in \text{dom}(value(e2)))$

Ein Beispiel für zwei Attribut-äquivalente TGraphen ist in Beispiel 6 auf Seite 52 zu sehen.

Anordnungs-Äquivalenz

Die Anordnung ist eine weitere Eigenschaft, die bei den bisherigen Definitionen der Äquivalenz nicht berücksichtigt wurde.

Definition 21 *Anordnungs-Äquivalenz*

Zwei TGraphen TG_1 und TG_2 gelten als Anordnungs-äquivalent, geschrieben $TG_1 \equiv_{ord} TG_2$, wenn ein totales, bijektives Mapping $M : TG_1 \rightarrowtail\!\!\!\!\!\rightarrow TG_2$ existiert, so dass die beiden TGraphen Element-äquivalent sind,$TG_1 \equiv_{element} TG_2$, und die folgenden Bedingungen erfüllt werden:

1. *Die Anordnung der Knoten sind unter dem Mapping M äquivalent.*
   $\forall v1 \in V_1, \forall v2 \in V_2, \exists i \in \mathbb{N}:$
   $m_V(v1) = v2 \Rightarrow V_{1Seq}[i] = v1 \wedge V_{2Seq}[i] = v2$
2. *Die Anordnung der Kanten sind unter dem Mapping M äquivalent.*
   $\forall e1 \in E_1, \forall e2 \in E_2, \exists i \in \mathbb{N}:$
   $m_E(e1) = e2 \Rightarrow E_{1Seq}[i] = e1 \wedge E_{2Seq}[i] = e2$
3. *Die Anordnung der Inzidenzen sind unter dem Mapping M äquivalent.*
   $\forall v1 \in V_1, \forall i \in \mathbb{N}, d \in \{in, out\}:$
   $\Lambda_{1Seq}(v1)[i] = (e1, d) \Rightarrow \Lambda_{2Seq}(m_V(v1))[i] = (m_E(e1), d)$

Zusammenfassung

Zwei TGraphen können unter verschiedenen Blickwinkeln auf ihre Gleichwertigkeit hin betrachtet werden. Hierzu wurden die Begriffe der Inzidenz-Äquivalenz, der Typ-Äquivalenz, der Attribut-Äquivalenz und der Anordnungs-Äquivalenz definiert. Welche dieser Aspekte für die Erzeugung von TGraph-Deltas letztlich berücksichtigt werden, ist abhängig von dem jeweiligen Anwendungsszenario und den Anforderungen. So ist es z.B. möglich, dass für ein bestimmtes Anwendungsgebiet die Anordnungs-Äquivalenz für TGraphen nicht gefordert wird.

Die TGraph-Äquivalenz

Definition 22 *TGraph-Äquivalenz*

*Sei $Equivalence = \{Element, Incidence, Type, Attribute, Order\}$ die Menge der definierten Äquivalenzen. Eine **TGraph-Äquivalenz** besteht aus einer Teilmenge von Equivalence.*

Sind zwei TGraphen z.B. Element-äquivalent und Typ-Äquivalent, so gelten sie unter dem Blickwinkel der TGraph-Äquivalenz $\{Element, Type\} \subset Equivalence$ als gleichwertig. Betrachtet man jedoch die selben TGraphen unter dem Aspekt der TGraph-Äquivalenz $\{Element, Incidence, Type\}$, so sind diese TGraphen nicht gleichwertig.

2.6.2. Änderungsoperationen auf TGraphen

Bereits zu Beginn von Abschnitt 2.6 wurde festgestellt, dass es durch ein Delta zweier TGraphen TG_1 und TG_2 möglich ist, aus dem TGraphen TG_1 einen zu TG_2 äquivalenten TGraphen TG_2' zu erzeugen. Um TG_2' zu erzeugen, muss der TGraph TG_1 schrittweise verändert werden. Dazu werden Änderungsoperationen benötigt. Eine **Änderungsoperation** verändert einen TGraphen in spezifizierter Art und Weise.

Grundlegende TGraph-Änderungsoperationen

Im Folgenden werden grundlegende Änderungsoperationen auf TGraphen beschrieben. Es ist denkbar und nicht ausgeschlossen, dass für bestimmte Anwendungsfälle weitere Änderungsoperationen definiert werden. Dies erscheint sinnvoll, um beispielsweise Deltas kompakter darzustellen oder um Änderungsinformationen auf einer höheren konzeptuellen Ebene im Delta zu repräsentieren.

Zur Vereinfachung der Spezifikation wird von der Annahme ausgegangen, dass Änderungsoperationen nur mit Argumenten aufgerufen werden, so dass das Ergebnis der Operation immer ein Schema-konformer TGraph ist. Bei der Spezifikation der Änderungsoperationen werden deshalb keine Bedingungen bzgl. der **Schema-Konformität** gestellt. Beim späteren Entwurf und der Implementierung müssen jedoch diese Annahmen berücksichtigt werden. Die JGRaLab-API verhindert das Erzeugen von TGraphen, die nicht Schema-konform sind. In einem solchen Fall wird eine Ausnahme erzeugt. Beim Entwurf und der Implementierung ist dies zu berücksichtigen.

Im Folgenden werden einige grundlegende Änderungsoperationen spezifiziert, die auf einen gegebenen TGraphen TG ausgeführt werden können.

Hinzufügen eines Knotens

Das Hinzufügen eines Knotens zu einem TGraphen TG ist eine der grundlegenden Änderungsoperationen auf TGraphen. Ihre Signatur kann wie folgt angegeben werden:
$AddVertex : TypeId \rightarrow Vertex$
Diese Operation ist wie folgt spezifiziert:

AddVertex

$\Delta(TG)$
$type? : TypeId$
$v! : Vertex$

$\exists\, v! \in Vertex : type(v!) = type?$
$TG' = (V_{Seq} \frown \langle v! \rangle, E_{Seq}, \Lambda_{Seq}, type', value)$

Durch erfolgreiche Ausführung der Operation wird ein Knoten $v!$, des durch den Parameter *type* spezifizierten Typs, erzeugt. Der Knoten wird am Ende der Knotensequenz angehängt. Dies wird in der Nachbedingung durch die Konkatenation von V_{Seq} und $\langle v? \rangle$ ausgedrückt.

Löschen eines Knotens

Das Löschen eines Knotens aus einem TGraphen erzeugt einen neuen TGraphen. Die Signatur dieser Änderungsoperation kann wie folgt angegeben werden:
$DeleteVertex : Vertex$
Die Operation ist wie folgt spezifiziert.

DeleteVertex

$\Delta(TG)$
$v? : Vertex$

$v? \in V \wedge \Lambda_{Seq}(v?) = \langle\rangle$
$TG' = (V_{Seq} \upharpoonright (V \setminus v?), E_{Seq}, \Lambda_{Seq}, type, value)$

Diese Änderungsoperation ist nur anwendbar, wenn der zu entfernende Knoten im TGraphen enthalten ist und der Knoten zu keiner Kante des TGraphen inzident ist.

Das Ergebnis der Operation ist ein TGraph, der den Knoten nicht enthält. Dabei bleibt die Anordnung der übrigen Knoten erhalten. Dies wird mit Hilfe des Filter-Operators $\upharpoonright$ ausgedrückt. V_{Seq} ist eine Sequenz über V. Der Filter-Operator erzeugt aus der Sequenz V_{Seq} und einer Teilmenge von V, hier $V \setminus v$, eine Teilsequenz von V_{Seq}. Diese besteht nur aus den Elementen der Teilmenge $V \setminus v$. Die Anordnung der Elemente bleibt erhalten.

Hinzufügen einer Kante

Analog zu den Knoten gibt es auch für Kanten Änderungsoperationen. Die Änderungsoperation zum Hinzufügen einer Kante lässt sich wie folgt angeben:
$AddEdge : TypeId \rightarrow Edge$
Sie wird wie folgt spezifiziert:

AddEdge

$\Delta(TG)$
$type? : TypeId$
$v_{out}?, v_{in}? : Vertex$
$e! : Edge$

$v_{out}? \in V \wedge v_{in}? =\in V$
$\exists\, e! \in Edge : type(e!) = type?$
$TG' = (V_{Seq}, E_{Seq} \frown e?, \Lambda'_{Seq}, type, value)$
$\forall\, x \in (V \setminus \{v_{out}?, v_{in}?\}) : \Lambda'_{Seq}(x) = \Lambda_{Seq}(x)$
$\Lambda'_{Seq}(v_{out}?) = \Lambda_{Seq}(v_{out}?) \frown (e!, out)$
$\Lambda'_{Seq}(v_{in}?) = \Lambda_{Seq}(v_{in}?) \frown (e!, in)$

Durch Ausführung einer solchen Operation wird einem TGraphen TG eine Kante e hinzugefügt. Die Vorbedingungen zur Ausführung der Änderungsoperation ist, dass der Startknoten v, als auch der Zielknoten w der Kante im TGraphen enthalten sind.

Das Ergebnis der Operation ist ein veränderter TGraph TG. Die hinzugefügte Kante e befindet sich am Ende der Kantensequenz von TG. An die Inzidenzsequenz des Startknotens v wird das Tupel (e, out) angehängt. An die Inzidenzsequenz des Zielknotens w wird das Tupel (e, in) angehängt.

Löschen einer Kante

Neben dem Hinzufügen einer Kante beinhaltet ein grundlegender Operationssatz auf TGraphen das Löschen einer Kante.
DeleteEdge : *Edge*
Diese Änderungsoperation ist wie folgt spezifiziert:

DeleteEdge

$\Delta(TG)$
$e? : Edge$

$e? \in E$
$TG'! = (V_{Seq}, E_{Seq} \upharpoonright (E \setminus e?), \Lambda'_{Seq}, type, value)$
$\forall\, v \in (V \setminus \{\alpha(e?), \omega(e?)\}) : \Lambda'_{Seq}(v) = \Lambda_{Seq}(v)$
$\Lambda'_{Seq}(\alpha(e?)) = \Lambda_{Seq}(\alpha(e?)) \upharpoonright (\mathrm{ran}(\Lambda_{Seq}(\alpha(e?))) \setminus (e?, out))$
$\Lambda'_{Seq}(\omega(e?)) = \Lambda_{Seq}(\omega(e?)) \upharpoonright (\mathrm{ran}(\Lambda_{Seq}(\omega(e?))) \setminus (e?, in))$

Die einzige Vorbedingung der Operation ist, dass die zu löschende Kante im TGraphen enthalten ist.

Das Ergebnis ist ein veränderter TGraph TG, der die zu löschende Kante e nicht mehr enthält. Die Kante e wird aus der Kantensequenz entfernt, dabei bleibt die Anordnung der übrigen Kanten erhalten. Zudem wird als Nachbedingungen gefordert, dass die Tupel (e, in) und (e, out) aus den Inzidenzsequenzen des Start-

knotens, bzw. des Zielknotens von e entfernt werden. Dabei darf die Anordnung der übrigen Tupel in den Inzidenzsequenzen nicht verändert werden.

Veränderung des Startknotens einer Kante

Neben dem Hinzufügen und Löschen von Elementen sind auch Änderungsoperationen zum Verändern vorhandener Elemente und deren Beziehungen wichtig. Hierzu zählt die Veränderung des Startknotens einer Kante.
Ihre Signatur ist wie folgt definiert:
$UpdateAlpha : Edge \times Vertex$
Die Änderungsoperation ist wie folgt spezifiziert:

UpdateAlpha

$\Delta(TG)$
$e? : Edge$
$v? : Vertex$

$e? \in E \wedge v? \in V \wedge (e?, out) \notin \Lambda_{Seq}(v?)$
$TG' = (V_{Seq}, E_{Seq}, \Lambda'_{Seq}, type, value)$
$\forall w \in (V \setminus \{\alpha(e?), v?\}) : \Lambda'_{Seq}(w) = \Lambda_{Seq}(w)$
$\Lambda'_{Seq}(\alpha(e?)) = \Lambda_{Seq}(\alpha(e?)) \upharpoonright (\text{ran}(\Lambda_{Seq}(\alpha(e?))) \setminus (e?, out))$
$\Lambda'_{Seq}(v?) = \Lambda_{Seq}(v?) \frown (e?, out)$

Durch die Ausführung dieser Operation wird der Startknoten einer Kante verändert. Die Vorbedingungen besagen, dass die Kante e und der neue Startknoten v im TGraphen TG enthalten sein müssen. Außerdem darf der neue Startknoten nicht mit dem alten Startknoten identisch sein.

Das Ergebnis ist ein TGraph, der aus denselben Knoten und Kanten besteht. Die Inzidenzsequenz des alten Startknotens und des neuen Startknotens werden durch die Änderungsoperation verändert. Das Tupel (e, out) wird aus der Inzidenzsequenz des alten Startknotens entfernt, ohne die Anordnung der übrigen Tupel zu verändern. Das Tupel (e, out) wird an die Inzidenzsequenz des neuen Startknotens v angehängt.

Veränderung des Zielknotens einer Kante

Die *UpdateOmega* Änderungsoperation verändert den Zielknoten einer Kante.
$UpdateAlpha : Edge \times Vertex$
Diese Operation ist wie folgt spezifiziert:

UpdateOmega

$\Delta(TG)$
$e? : Edge$
$v? : Vertex$

$e? \in E \wedge v? \in V \wedge (e?, in) \notin \Lambda_{Seq}(v?)$
$TG' = (V_{Seq}, E_{Seq}, \Lambda'_{Seq}, type, value)$
$\forall w \in (V \setminus \{\omega(e?), v?\}) : \Lambda'_{Seq}(w) = \Lambda_{Seq}(w)$
$\Lambda'_{Seq}(\omega(e?)) = \Lambda_{Seq}(\omega(e?)) \upharpoonright (\text{ran}(\Lambda_{Seq}(\omega(e?))) \setminus (e?, in))$
$\Lambda'_{Seq}(v?) = \Lambda_{Seq}(v?) \frown (e?, in)$

Um diese Änderungsoperation ausführen zu können muss die Kante e und ihr neuer Zielknoten v im TGraphen TG enthalten sein. Zudem darf der neue Zielknoten nicht mit dem alten Zielknoten der Kante übereinstimmen.

Das Ergebnis der Operation ist ein veränderter TGraph TG mit identischen Elementen. Lediglich die Inzidenzsequenzen des alten Zielknoten und des neuen Zielknoten werden verändert. Das Tupel (e, in) wird aus der Inzidenzsequenz des alten Zielknotens entfernt, ohne die Anordnung der übrigen Elemente zu verändern. An die Inzidenzsequenz des neuen Zielknoten wird das Tupel (e, in) angehängt.

Veränderung des Typs eines Elements

Die *UpdateType*-Operation verändert den Typ eines Elements.
$UpdateType : (Edge \cup Vertex) \times TypeId$
Diese Änderungsoperation wird wie folgt spezifiziert:

UpdateType

$\Delta(TG)$
$x? : (Edge \cup Vertex)$
$t? : TypeId$

$x? \in TG$
$TG'! = (V_{Seq}, E_{Seq}, \Lambda_{Seq}, type', value)$
$\forall y \in ((V \cup E) \setminus \{x?\}) : type'(y) = type(y)$
$type'(x?) = t?$

Die einzige Vorbedingung der Operation ist, dass das zu verändernde Element x in dem TGraphen TG enthalten ist.

Das Ergebnis der Operation ist ein veränderter TGraph, wobei nur der Typ des Elements x verändert wurde.

Veränderung der Attributierung eines Elements

Um den Wert eines Attributes zu verändern wird die *UpdateAttributeValue* Operation eingeführt.
$UpdateAttributeValue : (Edge \cup Vertex) \times AttrId \times Value$
Sie ist wie folgt spezifiziert:

UpdateAttributeValue

$\Delta(TG)$
$x? : (Edge \cup Vertex)$
$a? : AttrId$
$value? : Value$

$x? \in TG \land a? \in \mathrm{dom}(value(x?))$
$TG' = (V_{Seq}, E_{Seq}, \Lambda_{Seq}, type, value')$
$\forall y \in ((V \cup E) \setminus x?\}) : value'(y) = value(y)$
$\forall b \in (AttrId \setminus \{a?\}) : value'(x?)(b) = value(x?)(b)$
$value'(x?)(a?) = value?$

Das Element x, dessen Attribut a verändert werden soll, muss in dem TGraphen TG vorhanden sein. Zudem muss das Element x das Attribut a enthalten.

Das Ergebnis ist ein veränderter TGraph. Nur der Attributwert für das Attribut a des Elementes x wurde verändert.

Veränderung der Anordnung der Elemente

Die Veränderung der Anordnung der Elemente in den Knoten- bzw. Kanten-Sequenzen wird durch vier Änderungsoperationen ermöglicht. Die im folgenden spezifizierte *PutVertexAfter*-Operation ermöglicht es, einen Knoten in der Knotensequenz zu verschieben, indem er hinter einem anderen Knoten der Knotensequenz angeordnet wird.
$PutVertexAfter : Vertex \times Vertex$

PutVertexAfter

$\Delta(TG)$
$v1? : Vertex$
$v2? : Vertex$

$v1? \in V \land v2 \in V$
$TG' = (V'_{Seq}, E_{Seq}, \Lambda_{Seq}, type, value)$
$\mathbf{let}\, r = V_{Seq} \upharpoonright (V \setminus v1?)$
$(\mathbf{let}\, s \frown t = r) \land last\ s = v2?$
$V'_{Seq} = s \frown v1? \frown t$

Die Vorbedingung wird erfüllt, wenn der zu verschiebende Knoten $v1$ und sein neuer Vorgänger $v2$ in dem TGraphen TG enthalten sind.

Die Nachbedingung der Operation wird erfüllt, wenn ein TGraph TG' entsteht, der dem TGraphen TG, mit Ausnahme der Anordnung der Knoten, gleicht. Der Knoten $v1$ befindet sich in TG' in der Knotensequenz hinter dem Knoten $v2$. Die Anordnung der übrigen Knoten wird dadurch implizit verändert. Der ursprüngliche Vorgänger und der ursprüngliche Nachfolger von $v1$ in der Knotensequenz V_{Seq} in TG sind in der Knotensequenz V'_{Seq} in TG' direkte Nachbarn. Der Nachfolger von $v2$ in der Knotensequenz V_{Seq} ist in der Knotensequenz V'_{Seq} der Nachfolger von $v1$.

Analog zu der *PutVertexAfter*-Operation wird die *PutEdgeAfter*-Operation spezifiziert.
$PutEdgeAfter : Edge \times Edge$
Diese Änderungsoperation verschiebt eine Kante in der Kantensequenz eines TGraphen.

PutEdgeAfter

$\Delta(TG)$
$e1? : Edge$
$e2? : Edge$

$e1? \in E \land e2? \in E$
$TG' = (V_{Seq}, E'_{Seq}, \Lambda_{Seq}, type, value)$
$\textbf{let}\, r = E_{Seq} \upharpoonright (E \setminus e1?)$
$(\textbf{let}\, s \frown t = r) \land last\ s = e2?$
$E'_{Seq} = s \frown e1? \frown t$

Die Vorbedingungen und Nachbedingungen der Operation sind analog zu den Bedingungen der *PutVertexAfter*-Operation.

Bei den bisherigen Operationen *PutVertexAfter* und *PutEdgeAfter* wurden die Elemente unter Angabe ihres neuen Vorgängers verschoben. Die folgende beiden Operationen verschieben einen Knoten, bzw. eine Kante, innerhalb der Knoten- bzw. Kanten-Sequenz eines TGraphen vor ein anderes Element der Sequenz.
Zuerst wird die Operation zur Verschiebung eines Knotens in der Knotensequenz von TG spezifiziert. Ihre Signatur kann wie folgt angegeben werden:
$PutVertexAfter : Vertex \times Vertex$
Diese Operation ist wie folgt spezifiziert:

PutVertexBefore

$\Delta(TG)$
$v1? : Vertex$
$v2? : Vertex$

$v1? \in V \land v2? \in V$
$TG' = (V'_{Seq}, E_{Seq}, \Lambda_{Seq}, type, value)$
$\textbf{let}\, r = V_{Seq} \upharpoonright (V \setminus v1?)$
$(\textbf{let}\, s \frown t = r) \land head\ t = v2?$
$V'_{Seq} = s \frown v1? \frown t$

Der Knoten $v1$ der verschoben werden soll und der Zielknoten $v2$ vor den $v1$ verschoben werden soll, müssen im TGraphen TG bereits enthalten sein.

Das Ergebnis der Operation ist ein veränderter TGraph. Für dessen Knotensequenz V'_{Seq} gilt, dass der Knoten $v1$ nach Ausführung der Operation sich direkt vor $v2$ innerhalb der Sequenz befindet.

Analog dazu wird im Folgenden die *PutEdgeBefore*-Änderungsoperation spezifiziert.
$PutEdgeAfter : Edge \times Edge$
Sie verändert die Anordnung in der Inzidenzsequenz eines Knotens im TGraphen TG.

PutEdgeBefore

$\Delta(TG)$
$e1? : Edge$
$e2? : Edge$

$e1? \in E \land e2? \in E$
$TG' = (V_{Seq}, E'_{Seq}, \Lambda_{Seq}, type, value)$
$\textbf{let}\, r = E_{Seq} \upharpoonright (E \setminus e1?)$
$(\textbf{let}\, s \frown t = r) \land head\ t = e2?$
$E'_{Seq} = s \frown e1? \frown t$

Die zu verschiebende Kante $e1$ und die Kante $e2$, vor der sie in der Kantensequenz platziert werden soll, müssen im TGraphen TG enthalten sein.

Durch Ausführung der Operation wird der TGraph TG verändert. Die Kante $e1$ befindet sich in der veränderten Kantensequenz E'_{Seq} vor der Kante $e2$.

Veränderung der Inzidenz-Anordnung

Sowohl Kanten und Knoten, als auch die Inzidenzen der Knoten sind angeordnet. Zu jedem Knoten gibt es eine Inzidenzsequenz, welche die eingehenden und ausgehenden Kanten anordnet.

Mit den Änderungsoperationen *PutIncidenceAfter* und *PutIncidenceBefore* wird die Anordnung der zu einem Knoten inzidenten Kanten verändert.
$PutIncidenceAfter : Vertex \times Edge \times \{in, out\} \times Edge \times \{in, out\}$ Die Operation ist wie folgt spezifiziert:

PutIncidenceAfter

$\Delta(TG)$
$v? : Vertex$
$e1?, e2? : Edge$
$c, d \in \{in, out\}$

$v? \in V \wedge e1?, e2? \in E$
$(e1?, c) \in \Lambda_{Seq}(v?)$
$(e2?, d) \in \Lambda_{Seq}(v?)$
$TG' = (V_{Seq}, E_{Seq}, \Lambda'_{Seq}, type, value)$
$\forall\, w \in (V \setminus \{v?\}) : \Lambda'_{Seq}(w) = \Lambda_{Seq}(w)$
$\textbf{let}\, r = \Lambda_{Seq}(v?) \upharpoonright (\text{ran}(\Lambda_{Seq}(v?)) \setminus (e1?, c))$
$(\textbf{let}\, s \frown t = r) \wedge tail\ s = (e2?, d)$
$\Lambda'_{Seq} = s \frown (e1?, c) \frown t$

Der Knoten v, die Kante $e1$ und die Kante $e2$ müssen in der Knoten, bzw. Kantensequenz von TG vorkommen. Des Weiteren müssen beide Kanten zu dem Knoten v inzident sein. Durch den Parameter c wird festgelegt, ob es sich bei $e1$ um eine eingehende oder ausgehende Kante bezüglich v handelt. Durch den Parameter d wird festgelegt, ob es sich bei $e2$ um eine eingehende oder ausgehende Kante bezüglich v handelt.

Die Operation verändert nur die Inzidenzsequenz des Knotens v in dem TGraphen TG. Das Tupel $(e1, c)$, wird in der Inzidenzsequenz $\Lambda'_{Seq}(v)$ hinter dem Tupel $(e2, d)$, angeordnet.

Analog zur Änderungsoperation *PutIncidenceAfter* wird die Operation *PutIncidenceBefore* spezifiziert. Ihre Signaturen gleichen sich.
$PutIncidenceBefore : Vertex \times Edge \times \{in, out\} \times Edge \times \{in, out\}$
Die beiden Änderungsoperationen unterscheiden sich jedoch hinsichtlich ihrer Spezifikation.

PutIncidenceBefore

$\Delta(TG)$
$v? : Vertex$
$e1?, e2? : Edge$
$c, d \in \{in, out\}$

$v? \in V \wedge e1?, e2? \in E$
$(e1?, c) \in \Lambda_{Seq}(v?)$
$(e2?, d) \in \Lambda_{Seq}(v?)$
$TG' = (V_{Seq}, E_{Seq}, \Lambda'_{Seq}, type, value)$
$\forall\, w \in (V \setminus \{v?\}) : \Lambda'_{Seq}(w) = \Lambda_{Seq}(w)$
$\textbf{let}\, r = \Lambda_{Seq}(v?) \upharpoonright (\text{ran}(\Lambda_{Seq}(v?)) \setminus (e1?, c))$
$\textbf{let}(s \frown t = r) \wedge head\ t = (e2?, d)$
$\Lambda'_{Seq}(v?) = s \frown (e1?, c) \frown t$

Die Vorbedingungen der beiden Änderungsoperationen sind identisch. Lediglich bezüglich der Nachbedingungen unterscheiden sie sich. Die Änderungsoperati-

on *PutIncidenceBefore* ordnet ein Tupel $(e1, c)$ in der Inzidenzsequenz $\Lambda'_{Seq}(v)$ hinter dem Tupel $(e2, d)$ an.

2.6.3. Definition des TGraph-Delta

Die Begriffe der Äquivalenz von TGraphen und der Änderungsoperation auf TGraphen bilden die Grundlage zur Definition des TGraph-Delta.

Eine **Operationsanweisung** besteht aus einem Operationsbezeichner, z.B. *AddVertex*, und den benötigten Parameterwerten. Eine Anweisung spezifiziert die Ausführung einer Änderungsoperation. Wird eine Operationsanweisung op_i auf einen TGraphen TG ausgeführt, so entsteht ein veränderter TGraph TG'. Dies wird wie folgt notiert: $TG \stackrel{op_i}{\rightarrow} TG'$.

Ein Delta zweier TGraphen TG_1 und TG_2 ist eine **Sequenz von Operationsanweisungen**. Jede Operationsanweisung wird durch eine Änderungsoperation ausgeführt. Die erste Änderungsoperation im Delta verändert den TGraphen TG_1 und erzeugt einen veränderten TGraphen TG'_1. Dieser wird als Argument für die nächste Änderungsoperation verwendet, die durch die nächste Operationsanweisung im Delta festgelegt wird. Jede Änderungsoperation liefert als Ergebnis einen TGraphen, auf den die folgende Operation angewandt wird, bis alle Operationsanweisungen ausgeführt wurden. Wurden alle Operationen erfolgreich ausgeführt, so ist das Ergebnis der zu TG_2 äquivalente TGraph TG'_2. Die Ausführung der Änderungsoperationen eines Deltas auf einen TGraphen wird als **Patch** bezeichnet.

Definition 23 *TGraph-Delta*

Seien TG_1 und TG_2 zwei TGraphen. Sei $\Delta = \langle op_1, op_2, \ldots, op_n \rangle$ eine Sequenz von Änderungsoperationsanweisungen, so dass gilt: $TG_1 \stackrel{op_1}{\rightarrow} \ldots \stackrel{op_n}{\rightarrow} TG'_2$. Die Sequenz Δ bezeichnet man als TGraph-Delta von TG_1 und TG_2, wenn $TG'_2 \equiv TG_2$ gilt.

In der Definition wird die Äquivalenzrelation nicht festgelegt. Zwei TGraphen können unter verschiedenen Blinkwinkeln als äquivalent betrachtet werden.

Ein TGraph-Delta wird durch den **diff** Operator berechnet. Sei *TGraph* die Menge aller TGraphen und *Delta* die Menge aller TGraph-Deltas. Der diff Operator erzeugt zu zwei gegebenen TGraphen ein TGraph-Delta. Neben den beiden TGraphen besteht dabei die Eingabe auch aus einem Element der *TGraphEquivalence*. Dieses bestimmt unter welchem Blickwinkel der Gleichwertigkeit das Delta zu erzeugen ist. Wird zum Beispiel festgelegt, dass lediglich die Element-Äquivalenz zu berücksichtigen ist, so wird das generierte Delta lediglich Operationsanweisungen zum Hinzufügen und Entfernen von Elementen beinhalten.

$$diff : TGraph \times TGraph \times TGraphEquivalence \rightarrow Delta$$

Dies bedeutet, dass zu zwei TGraphen unterschiedliche Deltas berechnet werden können, je nachdem auf welche TGrap-Äquivalenz die Berechnung des Deltas abzielt.

Die Konkatenation von TGraph-Deltas

Sei $\Delta_1 = \langle op_1, op_2, \ldots, op_n \rangle$ ein TGraph-Delta der TGraphen TG_1 und TG_2. Sei $\Delta_2 = \langle op_j, op_{j+1}, \ldots, op_m \rangle$ ein TGraph-Delta der TGraphen TG_3 und TG_4. Die **Konkatenation** beider Deltas ergibt die Sequenz $\Delta_1 \circ \Delta_2 = \langle op_i, op_2, \ldots, op_n, op_j, op_{j+1}, \ldots, op_m \rangle$. Die Sequenz $\Delta_1 \circ \Delta_2$ ist ein TGraph-Delta der TGraphen TG_1 und TG_4, wenn gilt: $TG_2 \equiv TG_3$. Wenn die beiden TGraphen TG_2 und TG_3 nicht äquivalent sind, so ist nicht feststellbar, ob die Ausführung der Änderungsoperationen einen zu TG_4 äquivalenten TGraphen erzeugt.

2.6.4. Patch

Wird ein Delta auf einen TGraphen angewendet, so spricht man von einem **Patch**. Einen TGraphen zu patchen bedeutet die **schrittweise Durchführung der Änderungsanweisungen** eines gegebenen Deltas auf den TGraphen. Das Ergebnis ist ein veränderter TGraph.

Ein Patch ist eine Operation, die als Eingabe einen TGraphen und ein Delta erhält. Durch Abarbeitung der Änderungsoperationen wird ein TGraph erzeugt. Sei *TGraph* die Menge aller TGraphen und *Delta* die Menge aller TGraph-Deltas. Der **patch** Operator ist wie folgt definiert:

$$patch : TGraph \times Delta \rightarrow TGraph$$

Sei ein Delta Δ zweier TGraphen TG_1 und TG_2 gegeben. Wenn TG_1 mit Δ erfolgreich gepatcht wird, so ist das Ergebnis ein TGraph TG_2', der äquivalent ist zu TG_2.

Ein Patch mit Δ kann auch auf einem zu TG_1 verschiedenen TGraphen TG_3 ausgeführt werden. Es ist jedoch nicht garantiert, dass dieser Patch durchgeführt werden kann. So ist z.B. die Ausführung einer Änderungsoperation zum löschen eines Elementes nur dann anwendbar, wenn dieses Element auch in TG_3 existiert.

Wird ein Delta Δ zweier TGraphen TG_1 und TG_2 auf einen zu TG_1 nicht äquivalenten TGraphen TG_3 angewendet, so kann nicht garantiert werden, dass jede Operationsanweisung in *Delta* ausgeführt werden kann.

2.7. TGraph-Merging

Der **Prozess des Zusammenführens mehrerer TGraphen** zu einem neuen TGraphen wird als **TGraph-Merge** bezeichnet. Das Ergebnis eines Merge ist ein TGraph. Dieser enthält Merkmale all derjenigen TGraphen, aus denen er erzeugt wurde.

Der Prozess des Mergings erfolgt im Idealfall automatisiert. Dabei können jedoch **Konflikte** auftreten. Deshalb ist beim Merging eine Interaktion mit einem menschlichen Akteur oft unumgänglich.

2.7.1. Konflikte

Seien zwei TGraphen $TG1$ und $TG2$ gegeben, die durch das TGraph-Merging zu einem neuen TGraphen zusammengeführt werden sollen. Sei $M : TG1 \nrightarrow TG2$ ein Mapping der beiden TGraphen und $\mathit{Difference}(TG1, TG2)$ ihre Differenz bezüglich M. Als **Konflikt** werden sämtliche in $\mathit{Difference}(TG1, TG2)$ erfassten Abweichungen der beiden TGraphen bezeichnet.

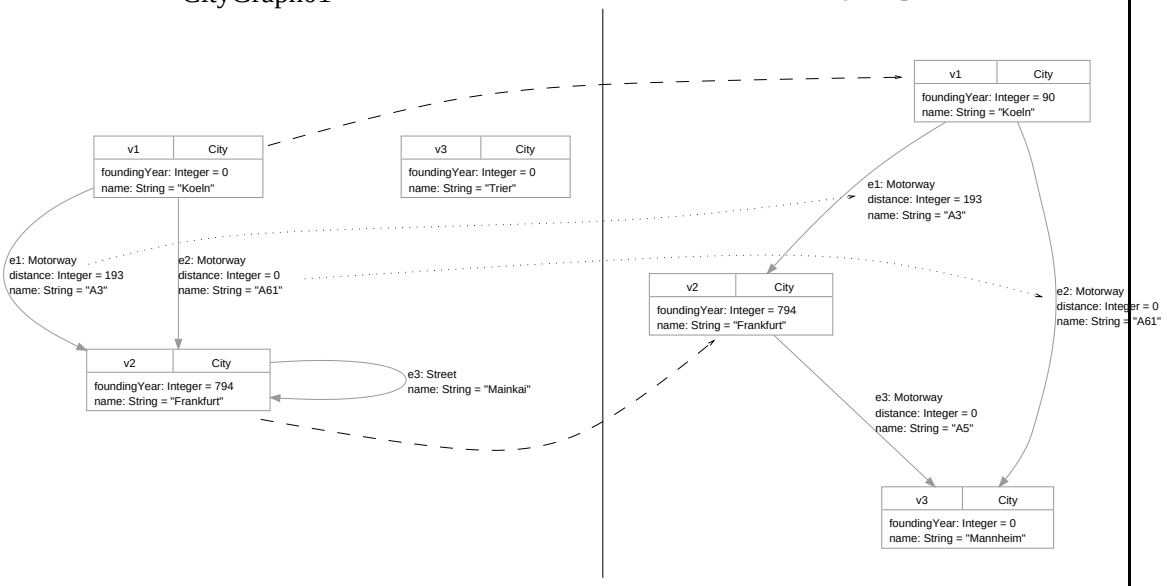
Neben der manuellen **Auflösung der Konflikte** durch einen Anwender, ist auch eine automatisierte Auflösung der Konflikte durch bestimmte Regeln denkbar. Beispielsweise könnte eine Regel festlegen, dass der zusammengeführte TGraph keine Elemente aus $TG1$ oder $TG2$ beinhaltet, die durch die Element-Differenz erfasst wurden. Durch Verwendung derartiger Regeln könnte ein Anwender bspw. versuchen mit möglichst geringem Aufwand aus zwei TGraphen einen neuen TGraphen zu erzeugen, der nur gemeinsame Eigenschaften der beiden TGraphen beinhaltet.

Eine andere sinnvolle Regel wäre bspw., dass Konflikte, die aufgrund der Anordnungs-Differenz zweier TGraphen entstehen, beim Merge nicht berücksichtigt werden. Eine solche Regel vereinfacht den Prozess des Mergings. Dies erscheint sinnvoll, wenn beispielsweise die Anordnung in dem neu generierten TGraphen für den Anwender keine wichtige Rolle spielt.

Beispiel 7 *Seien die beiden abgebildeten TGraphen und das eingezeichnete Mapping* $M = (m_V, m_E)$ *mit* $m_V = \{v1 \mapsto v1, v2 \mapsto v2\}$ *und* $m_E = \{e1 \mapsto e1, e2 \mapsto e2\}$ *gegeben.*

CityGraph01

CityGraph07



Beide TGraphen weichen bzgl. der Elemente, der Inzidenzen und der Attributierung voneinander ab. Die Anordnung wird in diesem Beispiel vernachlässigt. Wenn beide TGraphen durch einen TGraph-Merge zusammengeführt werden sollen, so müssen die durch die Differenz der TGraphen identifizierten Konflikte aufgelöst werden. So ist beispielsweise zu entscheiden, ob der erzeugte TGraph den Knoten $v3$ *aus* $CityGraph01$ *enthalten soll. Aufgrund der Inzidenz-Differenz der beiden TGraphen ist z.B. auch zu entscheiden, welcher Zielknoten der Kante* $e2$ *aus* $CityGraph01$ *zuzuordnen ist.*

Je nach Anwendungsszenario können derartige Regeln den Prozess des TGraph-Mergings vereinfachen. Dennoch ist eine manuelle Konfliktauflösung in manchen Anwendungsfällen unumgänglich.

In der vorliegenden Arbeit werden zwei Formen des TGraph-Merging betrachtet. Der Two-Way Merge und der Three-Way Merge.

2.7.2. Two-Way Merge

Beim **Two-Way Merge** wird aus zwei TGraphen $TG1$ und $TG2$ ein neuer TGraph $TG3$ erzeugt.

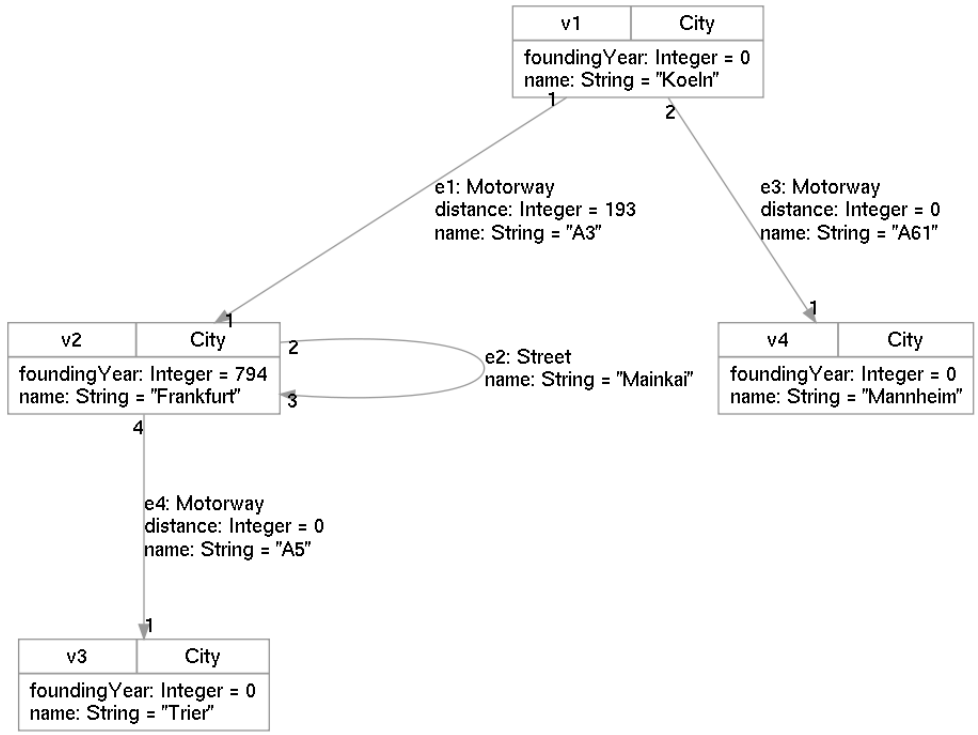
$$twoWayMerge : TGraph \times TGraph \rightarrow TGraph$$

Das Problem des Two-Way Merge ist die Generierung eines konsistenten TGraphen durch Aufdeckung und Auflösung der Konflik-

te, die bei der Zusammenführung der beiden TGraphen auftreten.

Problem:	Two-Way Merge
Input:	zwei TGraphen TG_1 und TG_2, die zum gleichen TGraph-Schema *TGS* konform sind
Output:	ein zu *TGS* konformer TGraph *TG'*, der aus den TGraphen TG_1 und TG_2 besteht

Beispiel 8 *Die Zusammenführung der beiden TGraphen aus Beispiel 7 auf der vorherigen Seite wird als Two-Way Merge bezeichnet. Die auftretenden Konflikte werden durch die Anwendung von Regeln und der Interaktion mit einem Anwender aufgelöst. Wurden alle Konflikte aufgelöst, so entsteht ein neuer TGraph :*



2.7.3. Three-Way Merge

Eine weitere Form der Zusammenführung zweier TGraphen ist der **Three-Way Merge**. Dabei sollen zwei TGraphen $TG1$ und $TG2$, die beide aus einem TGraphen $TG0$ entstanden sind, zusammengeführt werden. $TG0$ wird in diesem Zusammenhang als der gemeinsame **Basis-TGraph** von $TG1$ und $TG2$ bezeichnet.

Wie auch beim Two-Way Merge ist das Ergebnis des Three-Way Merge ein neuer TGraph. Dieser soll alle Änderungen, die in $TG1$ und $TG2$ an dem Basis-TGraphen $TG0$ durchgeführt wurden, beinhalten.

$$threeWayMerge : TGraph \times TGraph \times TGraph \rightarrow TGraph$$

Problem:	Three-Way Merge
Input:	zwei TGraphen $TG1$ und $TG2$, die aus einem gemeinsamen Basis-TGraphen $TG0$ erzeugt wurden und zu einem gleichen TGraph-Schema TGS konform sind
Output:	ein zu TGS konformer TGraph TG', der aus $TG0$ und den daran ausgeführten Änderungen in den TGraphen $TG1$ und $TG2$ besteht

KAPITEL 3

ANALYSE UND DEFINITION DER AUFGABENSTELLUNG

3.1. Aufgabenstellung

Im Folgenden wird die Zielsetzung der Diplomarbeit formuliert. Darüber hinausgehend wird mit der Produktidee und dem Produktkonzept eine Vision formuliert, die absichtlich über das Ziel der Diplomarbeit hinausgeht. Diese Produktvision soll es ermöglichen, zukünftige Verbesserungen und funktionale Weiterentwicklungen zu antizipieren, so dass das hier entwickelte Verfahren in einer zukünftigen Arbeit ohne tiefgreifende Änderungen verwendet werden kann.

3.1.1. Zielsetzung

Das Ziel der Arbeit liegt darin, eine Vorgehensweise zu entwickeln, die eine anwendbare Berechnung von Deltas für TGraphen ermöglicht. Unter der Anwendbarkeit versteht sich hierbei die Erfüllung verschiedener Anforderungen an die Benutzbarkeit, das Laufzeitverhalten und die Interoperabilität. Das Verfahren soll die Grundlage für ein Werkzeug sein, das Vergleiche unterschiedlicher TGraphen ermöglicht. Die Integration eines derartigen Vergleichswerkzeugs in bestehende Versionskontrollsysteme, wie zum Beispiel svn, cvs oder git, ist hierbei zu berücksichtigen.

3.1.2. Produktidee

Die TGraph-Diffutils sind eine Sammlung von Werkzeugen zur Unterstützung der Verwaltung von TGraphen. Jedes Werkzeug ist auf die Erfüllung einer bestimmten Aufgabe ausgerichtet.

Die TGraph-Diffutils enthalten Werkzeuge zur Erfüllung folgender Aufgaben:

- Vergleich von TGraphen
- Generierung von TGraph-Deltas
- Patchen von TGraphen
- Zusammenführung von TGraphen

Die TGraph-Diffutils Werkzeuge können in andere Anwendungen integriert werden. Sie stellen somit wichtige Bausteine zur Entwicklung TGraph-basierter Anwendungen dar.

Die TGraph-Diffutils Werkzeug-Sammlung ist vergleichbar mit den GNU Diffutils[1]. Dabei handelt es sich um Werkzeuge, die zur Verwaltung von textuellen Artefakten eingesetzt werden. Unter anderem ist das Unix Programm diff, welches für den Vergleich textueller Artefakte eingesetzt wird, in den GNU Diffutils enthalten.

3.1.3. Produktkonzept

Das Konzept der TGraph-Diffutils sieht es vor, dass basierend auf der JGralab API ein System entwickelt wird, das unterschiedliche Werkzeuge zur Verwaltung von TGraphen bereitstellt.

Der Fokus dieser Diplomarbeit liegt auf der Entwicklung eines Werkzeugs. Dessen Aufgabe ist es zu zwei gegebenen TGraphen ein Delta zu berechnen. Das Werkzeug soll über folgende Merkmale verfügen:

- Eine effiziente Berechnung von Deltas.
- Eine nahtlose Integration in die JGralab-Toolwelt.
- Eine unkomplizierte Erweiterbarkeit, vor allem in Bezug auf das verwendete Mapping Verfahren.
- Eine einfache Integration in bestehende VCS Systeme.

Ein Delta soll über folgende Merkmale verfügen:

- Eine möglichst kompakte Speicherung der Änderungsoperationen.
- Eine textuelle Repräsentation, die sowohl maschinen-, als auch menschenlesbar ist.

[1] `http://www.gnu.org/software/diffutils/diffutils.html` Stand: 1. Juli 2010

Ein Delta muss für ein Patch-Verfahren anwendbar sein. Zugleich muss ein Mensch in der Lage sein, die Änderungen zweier TGraphen anhand eines Deltas nachzuvollziehen.

3.2. Produktfunktionen

Neben der Entwicklung des Werkzeugs zur Delta-Generierung, ist auch die zukünftige Entwicklung weiterer Werkzeuge zu berücksichtigen. Es ist wichtig, die geforderten Produktfunktionen zu identifizieren. Dafür wurde das Anwendungsfalldiagramm in Abbildung 3.1 entworfen. Es zeigt die Anwendungsfälle und Akteure. Daraus können Produktfunktionen und Anforderungen abgeleitet werden.

Wie die Abbildung zeigt, gibt es verschiedene Anwendungsfälle, die bei der Entwicklung berücksichtigt werden müssen. Es werden hier zwei Akteure unterschieden. Das **Versionskontrollsystem** nutzt die TGraph-Diffutils zur Verwaltung von TGraphen. Der **Entwickler** nutzt die einzelnen Werkzeuge um mit TGraphen zu arbeiten oder als Bausteine für die Entwicklung eigener Anwendungen.

Für eine ausführlichere Beschreibung der einzelnen Anwendungsfälle sei hier auf Anhang A.2 verwiesen. Diese Anwendungsfälle dienen als Ausgangspunkt für die Formulierung der Anforderungen an die Produktfunktionen, die in dem folgenden Abschnitt erläutert werden.

3.3. Anforderungen

Die Anforderungen werden im Folgenden beschrieben und einer von vier möglichen Kategorien zugeordnet:

Pflicht. Alle Anforderungen dieser Kategorie müssen in der Diplomarbeit umgesetzt werden.

Absicht. Diese Anforderungen sind für die Zielerfüllung der Diplomarbeit nicht notwendig. Bei ausreichenden zeitlichen Ressourcen könnten sie jedoch umgesetzt werden.

Wunsch. Anforderungen, die in diese Kategorie eingeordnet werden, stellen Funktionalitäten und Eigenschaften des Systems dar, die nicht im Verlauf dieser Diplomarbeit umgesetzt werden.

Vorschlag. Anforderungen dieser Kategorie werden zur Umsetzung vorgeschlagen. Es muss jedoch noch entschieden werden, ob sie für die Entwicklung der TGraph-Diffutils von Bedeutung sind.

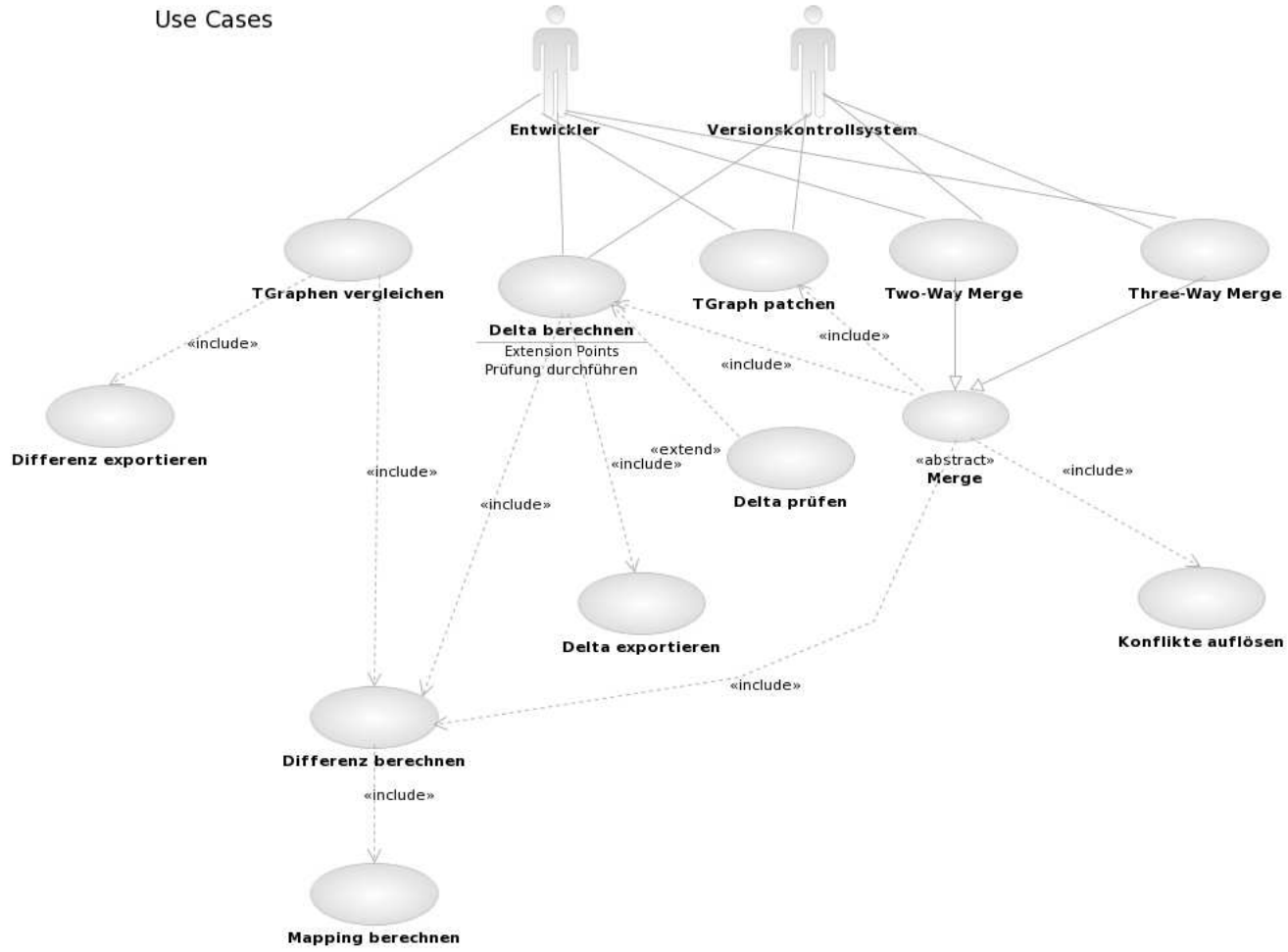


Abbildung 3.1.: Die Anwendungsfälle im Überblick

3.3.1. Anforderungen an Produktfunktionen

Die Anforderungen an die Funktionen des Systems werden in diesem Abschnitt beschrieben. Das System stellt einem Akteur bestimmte Funktionen zur Verfügung. Diese leiten sich in erster Linie aus den beschriebenen Anwendungsfällen des Systems her. Jede Produktfunktion stellt für einen Akteur einen bestimmten Mehrwert dar.

Anforderung 1 *Delta berechnen*

Beschreibung: Das System berechnet ein Delta zu zwei gegebenen TGraphen $TG1$ und $TG2$.

Kategorie: Pflicht

Anforderung 2 *Delta exportieren*

Beschreibung: Ein berechnetes Delta kann durch das System exportiert werden.
Kategorie: Pflicht

Anforderung 3 *TGraph patchen*

Beschreibung: Das System kann einen TGraphen mit einem Delta patchen.
Kategorie: Absicht

Anforderung 4 *TGraphen vergleichen*

Beschreibung: Das System kann zu zwei gegebenen TGraphen die Differenz berechnen und in einem geeigneten Format ausgeben.

Kategorie: Pflicht

Anforderung 5 *Two-Way Merge*

Beschreibung: Das System kann zwei TGraphen zu einem neuen TGraphen zusammenführen. Konflikte werden durch Interaktion mit dem Anwender aufgelöst.

Kategorie: Wunsch

Anforderung 6 *Three-Way Merge*

Beschreibung: Das System kann zwei TGraphen, die auf einem gemeinsamen Basis-TGraphen basieren, zu einem neuen TGraphen zusammenführen. Konflikte werden dabei durch das System automatisch aufgelöst.

Kategorie: Wunsch

3.3.2. Anforderungen an die Daten

Dieser Abschnitt befasst sich mit den Anforderungen an die Daten. Von besonderer Bedeutung für die Entwicklung der TGraph-Diffutils sind die Anforderungen an die Datenstrukturen, die für die Repräsentation des TGraph-Delta und der TGraph-Differenz verwendet werden.

Repräsentation des TGraph-Delta

Ein TGraph-Delta wird innerhalb der TGraphDiffutils in einer bestimmten Datenstruktur vorliegen. Für einen Akteur muss das System diese Datenstruktur in eine geeignete Repräsentationsform umwandeln. Ein Beispiel für eine mögliche Delta-Repräsentation wäre ein formal definiertes ASCII Datei Format oder eine mit XML-Schema definierte XML Sprache.

Anforderung 7 *Standard Format*

Beschreibung: Es wird ein formal spezifiziertes Standard Format zur Delta-Repräsentation benötigt.

Kategorie: Pflicht

Anforderung 8 *Verständlichkeit*

Beschreibung: Die Repräsentation des Delta muss in einem intuitiv verständlichen Format erfolgen. Für einen Anwender muss es möglich sein, die gespeicherten Änderungsoperationen im Delta und deren Wirkung zu interpretieren, ohne dass er diese ausführen und den Vor- und Nach-Zustand des TGraphen vergleichen muss.

Kategorie: Pflicht

Anforderung 9 *Speicherplatz-Effizienz*

Beschreibung: Insbesondere für ein VCS ist die kompakte Speicherung von Versionen im Repository durch Deltas wichtig. Ein Delta muss möglichst kompakt gespeichert werden können.

Kategorie: Absicht

Anforderung 10 *Schema-Unabhängigkeit*

Beschreibung: Die Repräsentation ist unabhängig von dem verwendeten TGraph-Schema der verglichenen TGraphen.

Kategorie: Pflicht

Anforderung 11 *TGraph-Unabhängigkeit*

Beschreibung: Das Delta kann unabhängig von dessen Entstehungs-Kontext auf beliebige TGraphen angewandt werden.

Kategorie: Vorschlag

Anforderung 12 *Erweiterung der Änderungsoperationen*

Beschreibung: Damit Deltas noch kompakter werden, können Änderungsoperationen erweitert werden. Änderungsoperationen beziehen sich dann nicht mehr auf einzelne Elemente, sondern auf Mengen von Elementen, die bestimmte Eigenschaften aufweisen. Dadurch werden weniger Änderungsoperationen benötigt. Die Deltas werden kompakter.

Kategorie: Vorschlag

Repräsentation der TGraph-Differenz

Anforderung 13 *Verständlichkeit*

Beschreibung: Für einen Anwender ermöglicht die Repräsentation der Differenz einen schnellen und leicht verständlichen Überblick zu Abweichungen zweier TGraphen.

Kategorie: Absicht

3.3.3. Anforderungen an die Anpassbarkeit und Erweiterbarkeit

Anforderung 14 *Einbringung neuer Mapping-Verfahren*

Beschreibung: Ein Entwickler kann ein neues Mapping-Verfahren in das System einbringen.

Kategorie: Pflicht

Diese Anforderung ist besonders wichtig, da es nicht Ziel der Diplomarbeit ist ein besonders effizientes Mapping-Verfahren zu entwicklen. Durch diese Anforderung soll sichergestellt werden, dass ein Entwickler ein neues Mapping-Verfahren ohne hohen Aufwand in die TGraph-Diffutils integrieren kann.

Anforderung 15 *Einbringung neuer Änderungsanweisungen*

Beschreibung: Ein Entwickler kann neue Änderungsanweisungen einbringen.

Kategorie: Pflicht

Anforderung 16 *TGraph-Äquivalenz für die Delta-Berechnung*

Beschreibung: Der Anwender kann selbst bestimmen, welche Äquivalenz von TGraphen durch das berechnete Delta erreicht werden soll. Ist z.B. die Anordnungs-Äquivalenz für den Einsatz des Deltas unwichtig, so kann der Anwender diese abwählen. Dadurch kann die Performanz des Verfahrens erhöht werden.

Kategorie: Pflicht

3.3.4. Anforderungen an die Kompatibilität und Interoperabilität

Anforderung 17 *Integration in VCS*

Beschreibung: Die Werkzeuge der TGraph-Diffutils können in aktuelle VCS, wie svn und git, integriert werden.

Kategorie: Pflicht

Anforderung 18 *Integration in JGraLab*

Beschreibung: Die Werkzeuge der TGraph-Diffutils fügen sich nahtlos in die JGraLab-Toolwelt ein.

Kategorie: Pflicht

Anforderung 19 *Aufwärtskompatibilität von TGraph-Deltas zu JGralab*

Beschreibung: Die TGraph-Diffutils sind in der Lage, Deltas, die mit älteren Versionen der JGralab API erstellt wurden, auch mit neueren Versionen der JGralab API zu verwenden.

Kategorie: Vorschlag

Anforderung 20 *Abwärtskompatibilität von TGraph-Deltas zu JGralab*

Beschreibung: Die TGraph-Diffutils sind in der Lage, Deltas, die mit neueren Versionen der JGralab API erstellt wurden, auch mit älteren Versionen der JGralab API zu verwenden.

Kategorie: Vorschlag

3.3.5. Anforderungen an die Benutzerschnittstelle

Anforderung 21 *Aufruf der Werkzeuge*

Beschreibung: Jedes Werkzeug der TGraph-Diffutils kann einzeln über ein eigenes Kommando aufgerufen werden.

Kategorie: Pflicht

Diese Anforderung ist für Anwender wichtig, die z.B. von der Kommandozeile aus auf die Funktionen der TGraph-Diffutils zugreifen möchten.

Anforderung 22 *Bereitstellung einer API*

Beschreibung: Die TGraphDiffutils stellen eine API zur Verfügung, die es Entwicklern ermöglicht, die Funktionen der Diffutils in eigenen Programmen zu verwenden.

Kategorie: Pflicht

Anforderung 23 *Anzeige des Fortschritts*

Beschreibung: Der Anwender kann sich den Fortschritt der aktuellen Berechnung anzeigen lassen.

Kategorie: Pflicht

Anforderung 24 *Visualisierung der Differenz von TGraphen*

Beschreibung: Die TGraph-Diffutils ermöglichen eine visuelle Darstellung der Differenz. Diese ermöglicht es einem Anwender einen Überblick über die Abweichungen zweier TGraphen zu erhalten.

Kategorie: Wunsch

KAPITEL 4

ENTWURF UND SPEZIFIKATION

4.1. Die Architektur der TGraph-Diffutils

4.1.1. Der Aufbau von JGraLab und die Einordnung der TGraph-Diffutils

Für den Entwurf der TGraph-Diffutils spielt das Umfeld eine große Rolle. Die TGraph-Diffutils basieren auf der JGraLab-API und erweitern diese um zusätzliche Dienstleistungen. Die nahtlose Integration der TGraph-Diffutils in die JGraLab-Toolwelt ist eine der wichtigsten Anforderungen, siehe Anforderung 18 auf Seite 80. Aus diesem Grund ist es wichtig, die Einordnung der TGraph-Diffutils in der JGraLab-Toolwelt zu berücksichtigen.

Der Überblick

Abbildung 4.1 liefert einen Überblick über die JGraLab-Toolwelt. Der verwendete Schichtenstil zeigt die einzelnen Komponenten geschichtet aus der Aufruf-Sicht. Jede Komponente einer Schicht verwendet ausschließlich Komponenten der gleichen Schicht oder der darunter liegenden Schichten. Durch diese Verwendung entstehen Abhängigkeiten zwischen den Komponenten, die in der Abbildung nicht explizit gezeigt werden.

Die unterste Schicht, genannt **Infrastructure**, besteht aus dem JGraLab-Core, der API für TGraphen und der Graph Repository Query Language 2, einer Anfragesprache für TGraphen. Der JGraLab-Core ist die Komponente, die von jeder anderen Komponente für die Verarbeitung von TGraphen direkt oder indirekt verwendet wird. Diese Komponente stellt eine API zur Verfügung, die eine direkte Verarbeitung und Manipulation von TGraphen ermöglicht.

Auf dieser Schicht bauen die **Utilities** auf, die JGraLab um zusätzliche Dienste erweitern. Diese Komponenten sind generisch, d.h. sie können jeden TGraphen verarbeiten, egal zu welchem Schema er konform ist. Hierzu zählt z.B. TG2Dot,

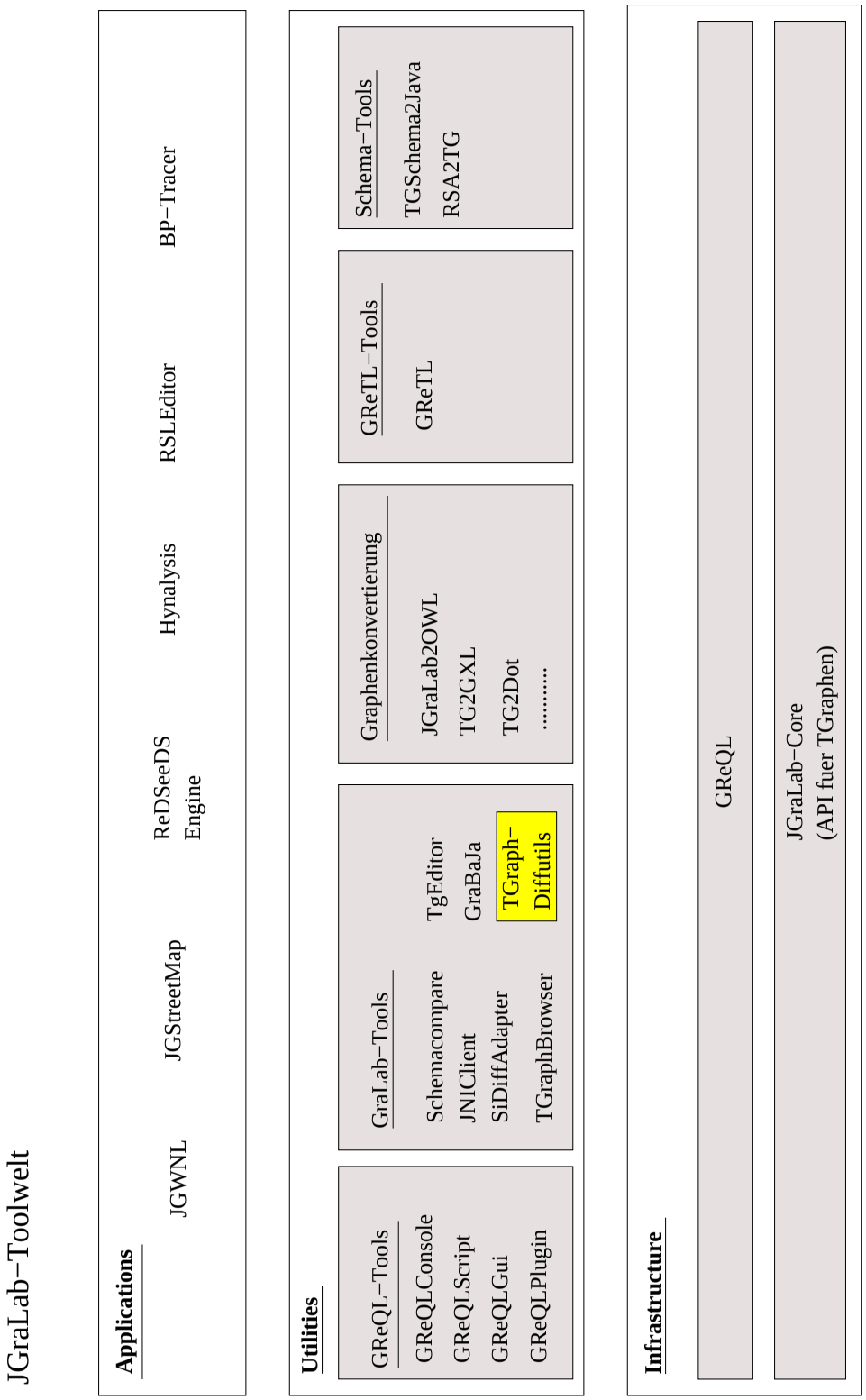


Abbildung 4.1.: Ein grober Überblick über die JGraLab-Toolwelt.

welches es ermöglicht, TGraphen im DOT Datei Format zu speichern und mittels der GraphViz[1]-Werkzeuge zu visualisieren. Ebenfalls auf dieser Schicht befindet sich GReTL, eine Transformationssprache, die zur Umsetzung von Modelltrans-

[1] `http://www.graphviz.org/`, Stand:1. Juli 2010

formationen dient oder zum Beispiel der JniServer, der den Zugriff auf TGraphen aus C++-Programmen heraus ermöglicht.

Die **TGraph-Diffutils** werden den Utilities zugeordnet. Sie erweitern JGraLab um Werkzeuge zum Vergleich und zur Zusammenführung von TGraphen.

Auf der Schicht der **Applications** befinden sich Anwendungen, die auf dem Core und den Utilities aufbauen und für einen konkreten Anwendungszweck entwickelt wurden. Sie sind im Allgemeinen nicht generisch und werden entwickelt, um mit TGraphen eines bestimmten Schemas zu arbeiten. Ein Beispiel für eine solche Anwendung ist jgStreetMap, die mit Graphen bzgl. eines vom OpenStreetMap[2]-Projekt adaptierten Schemas arbeiten.

4.1.2. Der Entwurf der Bausteine

Um die Bausteine genauer beschreiben zu können, werden sie klassifiziert. Die hier verwendete Klassifizierung orientiert sich an [Eva03]. Um in den Diagrammen die Klassifizierung der Bausteine explizit zu zeigen, werden entsprechende Stereotypen verwendet.

Application. Ein derartig gekennzeichneter Baustein stellt ein ausführbares Programm dar. Die von dem Programm bereitgestellte Funktion wird von den TGraph-Diffutils zur Verfügung gestellt, das Programm selbst enthält keine Anwendungslogik. Es nimmt die Optionen und Argumente des Anwenders entgegen und startet die erforderlichen Schritte für die Berechnung.

Facade. Ein solcher Baustein stellt eine alternative Schnittstelle für ein Subsystem dar. Er vereinfacht und vereinheitlicht den Zugriff auf das Subsytem. Klienten, die auf Funktionen des Subsystems zugreifen, werden somit besser vom Subsystem entkoppelt. Das Subsystem kann dadurch Veränderungen unterzogen werden, ohne dass Änderungen an den Klienten vorgenommen werden müssen.

Value Object. Ein Baustein der einen deskriptiven Aspekt der Domäne ohne konzeptuelle Identität repräsentiert, wird als Value Object klassifiziert. Ein Value Object erhält somit keine eigene Identität, sondern wird durch seine Attribute definiert. Es sollte als unveränderlich behandelt werden.

Entity. Ein Baustein, der über eine eigene Identität verfügt, wird als Entity klassifiziert. Entities können voneinander unterschieden werden, auch wenn sie die gleichen Attribute und Attribut-Werte beinhalten. Ein Entity unterliegt einem Lebenszyklus. Es kann unterschiedliche Zustände und Ausprägungen annehmen, ist meist veränderlich und kann durch seine Identität von anderen Entities unterschieden werden.

[2] `http://www.openstreetmap.de/`, Stand:1. Juli 2010

Aggregate Root. Eine besondere Form eines Entity Bausteins stellt ein Aggregate Root dar. Er zählt zu einer Bausteingruppe, die als **Aggregate** bezeichnet wird.
Es kann sinnvoll sein, mehrere Bausteine zu einem größeren Baustein zusammenzusetzen und als Einheit zu behandeln. Ein solcher zusammengesetzter Baustein wird als **Aggregate** bezeichnet. Jeder derartige Baustein verfügt über eine Aggregate Root. Ein Zugriff von Bausteinen außerhalb des Aggregates auf die Bausteine innerhalb des Aggregates ist nur über die Aggregate Root möglich. Die Bausteine werden somit abgekapselt. Veränderungen am Aggregate können besser kontrolliert werden, woduch die Konsistenz des zusammengesetzten Bausteins erhalten bleibt.
Die Aggregate Root kontrolliert den Zugriff auf die Bausteine innerhalb des Aggregates und ist für die Einhaltung von Invarianten zuständig. Er spezifiziert somit die Schnittstelle des Aggregates und stellt somit eine **Facade** dar. Dieser Baustein kann bei Bedarf temporäre Referenzen auf Bausteine innerhalb des Aggregates aushändigen.

Service. Eine Dienstleistung, die von dem System angeboten wird, keinem anderen Baustein direkt zugeordnet werden kann und ein wichtiges Konzept der Domäne darstellt, wird in Form eines Service Bausteins erfasst. Ein Service wird in Form eines Interface angeboten. Er ist zustandslos.

4.1.3. Die Paketstruktur der TGraph-Diffutils

Das Paketdiagramm in Abbildung 4.2 zeigt die hierarchische Strukturierung der TGraph-Diffutils und ihre Einordnung in die JGraLab Paketstruktur. Das Paket `de.uni_koblenz.jgralab.utilities.tgraphdiffutils` ist das Wurzelpaket der TGraph-Diffutils. Es enthält drei Pakete und mehrere Klassen.

Die Klasse `TGDiffutils` übernimmt eine besondere Aufgabe. Sie soll den einfachen Zugriff auf die einzelnen Werkzeuge der TGraph-Diffutils ermöglichen und kontrollieren. Hierzu stellt sie statische Operationen bereit. Sie enthält nicht die eigentliche Anwendungslogik, sondern delegiert die Aufgaben an die entsprechenden Objekte. Diese Klasse stellt Entwicklern die Funktionen der TGraph-Diffutils zur Verfügung und ist somit ein zentraler Bestandteil der API. Da sie Klienten eine Schnittstelle für das gesamte System bereitstellt und somit die Klienten von den TGraph-Difftuils entkoppelt, handelt es sich hierbei um eine **Facade**.

Bei den Klassen `TGDiff`, `TGCompare`, `TGPatch` und `TGMerge` handelt es sich um ausführbare Programme. Sie stellen die Funktionen der TGraph-Diffutils Endanwendern zur Verfügung, die nicht über die API auf die Funktionen zugreifen wollen, sondern die Werkzeuge von der Kommandozeile aus nutzen möchten. Jede dieser Klassen stellt somit eine **Application** dar,

Die Pakete `de.uni_koblenz.jgralab.utilities.tgraphdiffutils.core` und `de.uni_koblenz.jgralab.utilities.tgraphdiffutils.service` enthalten die essentiellen Bausteine der TGraph-Diffutils und die Anwendungs-

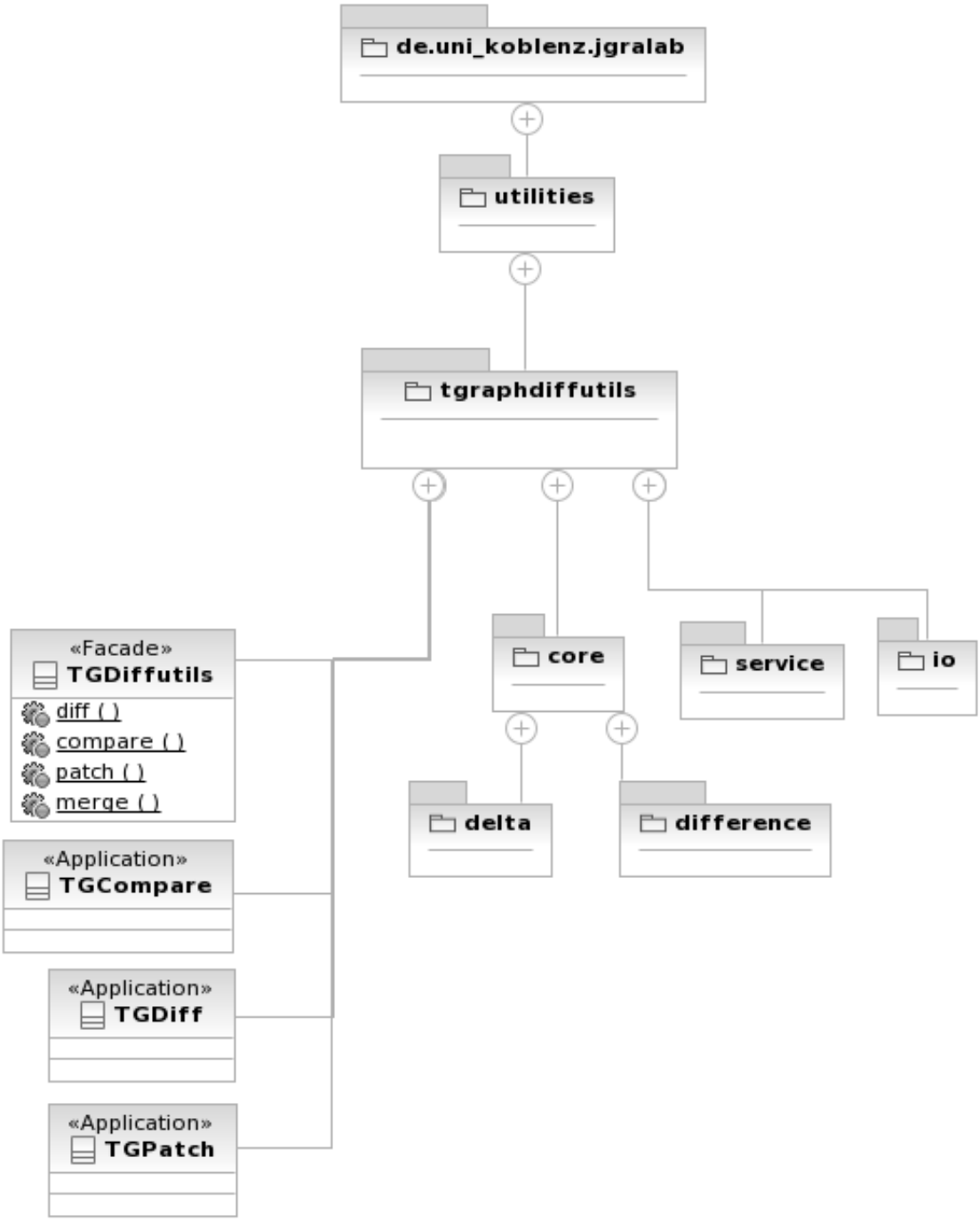


Abbildung 4.2.: Die Paketstruktur der TGraph-Diffutils.

logik. Sie setzen die geforderten Produktfunktionen um. Diese Bausteine und ihre Beziehungen werden in dem folgenden Abschnitt erläutert.

Das Paket `de.uni_koblenz.jgralab.utilities.tgraphdiffutils.io` enthält Klassen, die von den TGraph-Diffutils für die Eingabe und Ausgabe verwendet werden.

4.2. Der TGraph-Diffutils Core

Die TGraph-Diffutils erweitern JGraLab um Dienstleistungen. Für die Bereitstellung dieser Leistungen werden bestimmte Endprodukte und Zwischenprodukte benötigt. Der **TGraph-Diffutils Core** beinhaltet diejenigen Bausteine, die ein Endprodukt oder ein Zwischenprodukt darstellen, sowie weitere Bausteine zur Erzeugung und Verarbeitung dieser Produkte.

In Kapitel 2 wurden die Begriffe und ihre Zusammenhänge erläutert. Aus der Terminologie und den Anforderungen an die Produktfunktionen, siehe Kapitel 3, können folgende Bausteine identifiziert werden:

Mapping. Der Mapping-Baustein entspricht dem in Definition 10 definierten **Mapping von TGraphen**. Seine Spezifikation muss somit den dort definierten Eigenschaften entsprechen. Zwei Mapping-Objekte werden als identisch betrachtet, wenn ihre Attribute gleich sind. Ein Mapping wird somit durch seine Attribute definiert. Es sollte, nachdem es erzeugt wurde, als unveränderlich behandelt werden. Dieser Baustein kann als **Value Object** klassifiziert werden. Der Mapping-Baustein wird durch die Klasse `Mapping` realisiert, deren Spezifikation befindet sich im Anhang A.1.1 auf Seite 151.

Differenz. Der Differenz-Baustein repräsentiert die in Definition 16 formulierte **Differenz von TGraphen**. Dieser Baustein besteht aus mehreren kleineren Bausteinen, die jeweils einen Aspekt der Differenz repräsentieren. Dieser Baustein wird als **Aggregate** klassifiziert.

Delta. Der Delta-Baustein entspricht dem in Definition 23 definierten TGraph-Delta. Er besteht aus einer Sequenz von Änderungsoperationsanweisungen und kontrolliert den Zugriff auf die Änderungsoperationsanweisungen. Dieser Baustein wird als **Aggregate** klassifiziert.

Diese drei Bausteine stellen den wichtigsten Bestandteil des TGraph-Diffutils Core dar. Der Differenz- und der Delta-Baustein bestehen aus kleineren Bausteinen. Sie werden in den folgenden beiden Abschnitten detailliert erläutert und ihre Struktur wird dargestellt.

4.2.1. Der Differenz-Baustein

Der Entwurf des Differenz-Bausteins orientiert sich an der Definition 16 auf Seite 53 der TGraph-Differenz. Er wird als Aggregate modelliert. Eine formale Spezifikation befindet sich in Anhang A.1.2.

Eine TGraph-Differenz zweier TGraphen wird durch ein `Difference`-Objekt repräsentiert. Dieses übernimmt die Rolle des AggregateRoot. Jeglicher Zugriff auf Objekte innerhalb des Aggregates wird durch dieses Objekt kontrolliert. Bezie-

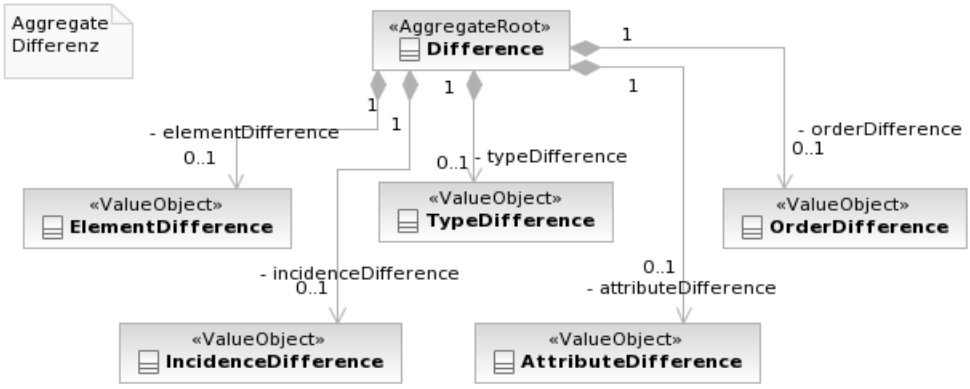


Abbildung 4.3.: Der Differenz-Baustein im Detail.

hungen zu anderen Objekten, wie z.B. den beiden TGraphen, werden durch das `Difference`-Objekt verwaltet. Es besteht aus

- einem `ElementDifference`-Objekt,
- einem `IncidenceDifference`-Objekt,
- einem `TypeDifference`-Objekt,
- einem `AttributeDifference`-Objekt und
- einem `OrderDifference`-Objekt.

Diese Objekte sind optional. Es kann vorkommen, dass z.B. die Anordnungs-Differenz nicht berechnet wird, um die Performanz des Verfahrens zu steigern. In diesem Fall ist es für die berechnete TGraph-Differenz nicht notwendig, ein `OrderDifference`-Objekt zu instanziieren und die Anordnungsdifferenz zu berechnen.

4.2.2. Der Delta-Baustein

Bei dem Delta-Baustein handelt es sich um ein Aggregate, siehe Abbildung 4.4. Sein Entwurf orientiert sich an der Definition 23 auf Seite 67 des TGraph-Deltas. Die Aggregate Root dieses Bausteins ist die Klasse `Delta`. Ein `Delta`-Objekt besteht aus einer angeordneten Liste von `Command`-Objekten.

Eine Klasse, die das `Command` Interface implementiert repräsentiert eine Klasse von Änderungoperationssanweisungen. Jedes `Command`-Objekt stellt eine Änderunsoperationsanweisung dar. Dazu zählt jede Subklasse der abstrakten Klasse `ChangeCommand`, da auch diese das Interface `Command` implementieren müssen. Diese abstrakte Klasse definiert gemeinsame Methoden und Attribute, wodurch redundanter Code in den `Command`-Klassen zusammengefasst werden kann.

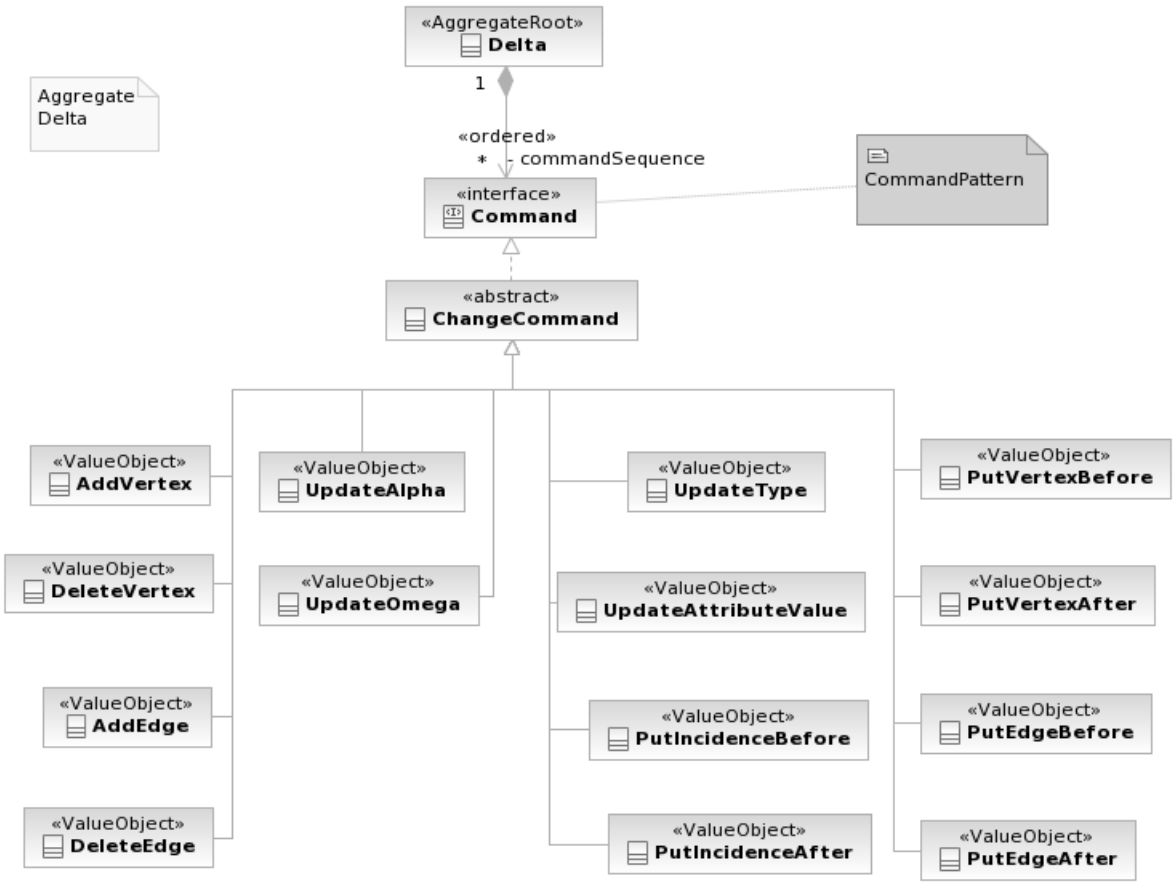


Abbildung 4.4.: Der Delta-Baustein im Detail.

Die in Abschnitt 2.6.2 auf Seite 58 definierten grundlegenden Änderungsoperationen werden hier durch entsprechende Bausteine modelliert. Dieser Satz an Änderungsoperationen kann beliebig erweitert werden, wie in Anforderung 12 gefordert.

Der Zugriff auf `Command`-Objekte wird durch die AggregateRoot `Delta` gekapselt. Ein `Delta`-Objekt kontrolliert somit den Zugriff auf die angeordneten Änderungsanweisungen. Die Überprüfung von Invarianten und die Ausnahmebehandlung kann dadurch an zentraler Stelle erfolgen. Die Ausführung der Änderungsoperationen erfolgt kontrolliert. Anweisungen, die vor oder nach der Ausführung einer Operation eines `Command`-Objektes erfolgen müssen, können leicht an zentraler Stelle verändert werden. Denkbar wäre z.B. eine Operation, welche jeweils nach Ausführung einer Änderungsoperationsanweisung aufgerufen wird und das Ergebnis der Ausführung in ein Fehlerprotokoll einträgt.

Eine formale Spezifikation des Delta-Bausteins befindet sich in Anhang A.1.3.

4.3. Die Services der TGraph-Diffutils

> A service is a software component of distinctive functional meaning that typically encapsulates a high-level business concept.[KBS04]

Die in Kapitel 2 eingeführten Operatoren, *diff*, *compare*, *patch*, *map* und *merge*, sowie die in Kapitel 3 erörterten Anwendungsfälle und Produktfunktionen, bilden die Grundlage für den Entwurf der **Services**.

Jeder **Service** stellt eine Dienstleistung bereit. Alle von den TGraph-Diffutils bereitgestellten Dienstleistungen werden in dem Paket `de.uni_koblenz.jgralab.utilities.tgraphdiffutils.service` erfasst.

Jeder in dieser Arbeit entworfene Service besteht aus einem Interface und seiner Implementation. Das **Interface** spezifiziert die Schnittstelle, die von einem Klienten genutzt werden muss, um die durch den Service bereitgestellte Dienstleistung in Anspruch zu nehmen.

Die **Implementation** eines Service darf den Klienten nur die, durch das Interface spezifizierten Operationen bereitstellen. Alle anderen Operationen und Attribute, die intern zur Realisierung der Dienstleistung benötigt werden, werden durch den Service gekapselt und sind für einen Klienten nicht sichtbar.

Im Folgenden werden die Services und ihr Entwurf beschrieben. Diese Beschreibung dient als Grundlage für ihre Realisierung.

Der Compare Service

Der Compare Service entspricht dem in Kapitel 2 definierten *compare*-Operator. Dessen Signatur $compare : TGraph \times TGraph \rightarrow Difference$ bildet die Grundlage zur Spezifikation des Interfaces für den Compare Service. Dieser Service setzt die im Anwendungsfall 5 auf Seite 167 und in der Anforderung 4 auf Seite 77 beschriebene Dienstleistung um. Das gleichnamige Interface `CompareService` spezifiziert seine Schnittstelle.

```
1 public interface CompareService {
2
3   public Difference compare(Graph leftTGraph, Graph rightTGraph,
        TGraphProperty... tGraphProperties);
4
5 }
```

Listing 3-1: Das Interface `CompareService`

Die Struktur dieses Service ist in Abbildung 4.5 zu sehen. Die Implementation der bereitgestellten Dienstleistung erfolgt in der Klasse `CompareServiceImpl`, die das `CompareService` Interface realisiert.

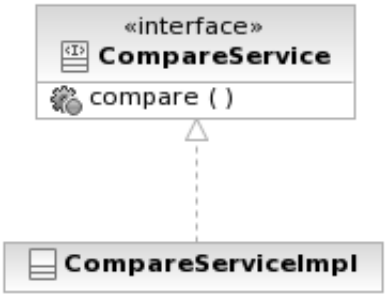


Abbildung 4.5.: Die Struktur des CompareService

Der Diff Service

Der `Diff` Service entspricht dem in Kapitel 2 definierten *diff*-Operator. Anhand seiner Signatur $diff : TGraph \times TGraph \times TGraphEquivalence \rightarrow Delta$ wurde das Interface in Listing 3-2 entworfen. Dieser Service setzt die in Anwendungsfall 1 auf Seite 163 und die in Anforderung 1 auf Seite 76 beschriebene Dienstleistung um.

```
1 public interface DiffService {
2
3   public Delta diff(Graph tgL, Graph tgR, TGraphProperty...
       tgEquivalence);
4
5 }
```

Listing 3-2: Das Interface `DiffService`

Die Struktur dieses Service ist in Abbildung 4.6 zu sehen. Die Implementation der bereitgestellten Dienstleistung erfolgt in der Klasse `DiffServiceImpl`, die das `DiffService` Interface realisiert.

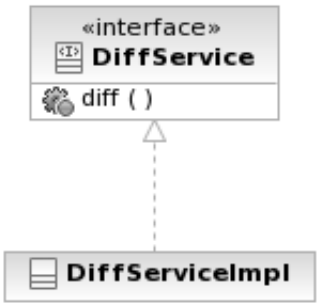


Abbildung 4.6.: Die Struktur des Diff Service

Der Patch Service

Der `Patch` Service entspricht dem in Kapitel 2 definierten *patch*-Operator. Anhand seiner Signatur $patch : TGraph \times Delta \rightarrow TGraph$ wurde das Interface `PatchService` in Listing 3-3 entworfen. Der *Patch* Service setzt somit die im Anwendungsfall 3 auf Seite 165 und in der Anforderung 3 auf Seite 77 beschriebene Dienstleistung um.

```
1 public interface PatchService {
2
3   public Graph patch(Graph tGraphToBePatched, Delta delta);
4
5   public Graph patch(Graph tGraphToBePatched, String
       deltaFileName);
6
7 }
```

Listing 3-3: Das Interface `PatchService`

Die Struktur dieses Service ist in Abbildung 4.7 zu sehen. Die Implementation der bereitgestellten Dienstleistung erfolgt in der Klasse `PatchServiceImpl`, die das `PatchService` Interface realisiert.

Abbildung 4.7.: Die Struktur des Patch Service

Der Map Service

Der `Map` Service entspricht dem in Kapitel 2 definierten *map*-Operator. Er erzeugt zu zwei gegebenen TGraphen ein Mapping. Dessen Signatur $map : TGraph \times TGraph \rightarrow Mapping$ entspricht das Interface des Map Service in Listing 3-5.

```
1 public interface MapService {
2
3   public String getId();
4
5   public MatchingStrategy getMatchingStrategy();
```

```
6
7   public Mapping map(Graph tgL, Graph tgR);
8
9   public void setMatchingStrategy(MatchingStrategy strategy);
10
11 }
```

Listing 3-4: Das Interface `MapService`

Dieser Service ist aus zwei Gründen von besonderer Bedeutung. Er wird von dem `Compare` Service zur Berechnung der Differenz benötigt. Die Qualität des `Compare` Service ist somit direkt abhängig von der Umsetzung des `Map` Service.

Der andere Grund liegt in der Variabilität seiner Umsetzung. Es gibt unterschiedliche Möglichkeiten ein Mapping zu berechnen. Jede Form der Berechnung kann ihre eigenen Vor- und Nachteile haben. Insbesondere das Laufzeitverhalten und die Genauigkeit der Berechnung können stark variieren. In dieser Diplomarbeit liegt der Fokus nicht auf der Entwicklung eines Verfahrens zur Mapping Berechnung. Wie in Anforderung 14 auf Seite 79 formuliert, soll es Entwicklern ermöglicht werden, die TGraph-Diffutils um neue Mapping-Verfahren zu erweitern.

Abbildung 4.8 veranschaulicht die Struktur des `Map` Service. Durch die Verwendung des Strategy Pattern[GHJV95] wird es ermöglicht, unterschiedliche Varianten zu implementieren. Diese haben unterschiedliche Vor- und Nachteile. Ein Akteur ist dann in der Lage eine Variante auszuwählen, die für sein Anwendungsszenario am besten geeignet ist. In Abbildung 4.8 ist der Feinentwurf des `Map` Service dargestellt.

Das `MapService` Interface wird durch die Klasse `MapServiceImpl` realisiert. Diese kontrolliert den Zugriff auf die `MatchingStrategy`. Die `MatchingStrategy` deklariert die Schnittstelle, die von allen angebotenen Algorithmen unterstützt wird.

Die Klasse `MapServiceImpl` implementiert die Operation `setMatchingStrategy(MatchingStrategy strategy)` des `MapService` Interface. Mit Hilfe dieser Operation kann ein Klient den `Map` Service konfigurieren.

Der TGraph-Merge Service

Weitere wichtige Dienstleistungen der TGraph-Diffutils betreffen die Zusammenführung von TGraphen, das TGraph-Merging. Die Umsetzung dieser Dienstleistung ist nicht Bestandteil der vorliegenden Diplomarbeit. Die Schnittstelle dieses Service wird durch das Interface `MergeService` spezifiziert.

```
1 public interface MergeService {
2
```

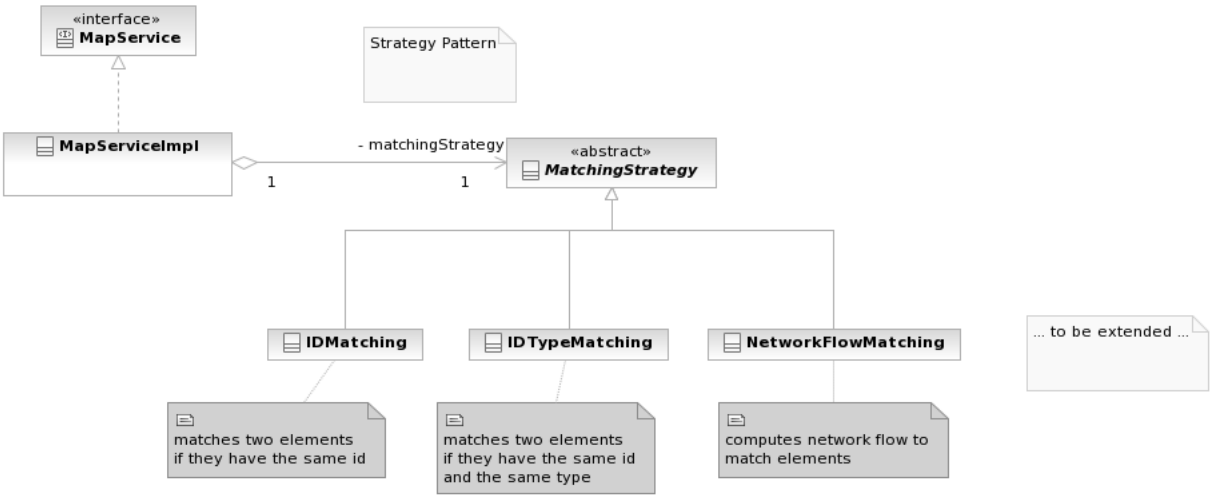


Abbildung 4.8.: Der Feinentwurf des `Map` Service.

```
3   public Graph merge(Graph tGraphLeft, Graph tGraphRight, Graph
       commonBaseTGraph, TGraphProperty... tGraphProperties);
4
5   public Graph merge(Graph tGraphLeft, Graph tGraphRight,
       TGraphProperty... tGraphProperties);
6
7 }
```

Listing 3-5: Das Interface `MergeService`

In den Zeilen 3 und 4 von Listing 3-5 werden zwei `merge` Operationen spezifiziert.

Die Spezifikation der Operation in Zeile 4 orientiert sich an dem *twoWayMerge*-Operator. Eine Implementierung dieser Operation löst somit das Problem des **Two-Way Merge**, siehe Abschnitt 2.7.2 auf Seite 70.

Diese Operation führt den linken TGraphen, `tGraphLeft`, und den rechten TGraphen, `tGraphRight`, zusammen und gibt als Ergebnis einen neuen TGraphen zurück. Die Zusammenführung soll derart erfolgen, dass alle gemeinsamen Eigenschaften der beiden TGraphen in den neuen TGraphen übernommen werden. Jegliche Abweichungen der beiden TGraphen, die durch die Differenz repräsentiert werden, stellen einen Konflikt dar. Die Implementierung der Operation muss einen geeigneten Mechanismus bereitstellen, damit diese Konflikte aufgelöst werden können. Denkbar sind hierbei die Interaktion mit einem menschlichen Anwender oder die Verwendung eines Regelwerks.

Die Operation in Zeile 4 erhält zudem als Argument ein `vararg` vom Typ `TGraphProperty`. Mit diesem Argument kann eine beliebige Menge an TGraph-Eigenschaften angegeben werden. Somit kann der Klient bestimmen, welche Abweichungen der beiden TGraphen als Konflikte zu interpretieren sind. Es kann zum Beispiel vorkommen, dass ein

Anwender zwei TGraphen zusammenführen möchte, deren Anordnung jedoch für seinen Anwendungsfall keine Rolle spielt. In diesem Fall soll der Aufruf `merge(tgl, tgr, TGraphProperty.ELEMENT, TGraphProperty.Incidence, TGraphProperty.Attribute, TGraphProperty.Type` dazu führen, dass die Anordnungs-Differenz der beiden TGraphen bei der Zusammenführung nicht berücksichtigt wird. Wird kein `TGraphProperty`-Literal als Argument übergeben, so sollte die Implementierung der Operation alle TGraph-Eigenschaften berücksichtigen.

Die Spezifikation der Operation in Zeile 3 orientiert sich an dem *threeWayMerge*-Operator. Eine Implementierung dieser Operation löst somit das Problem des **Three-Way Merge**, siehe Abschnitt 2.7.3 auf Seite 71.

4.4. Die Repräsentation des TGraph-Delta

Ein TGraph-Delta wird durch die Verwendung der TGraph-Diffutils generiert. Die TGraph-Diffutils ermöglichen es, einem Akteur zu zwei gegebenen TGraphen ein TGraph-Delta zu erzeugen, es persistent zu speichern, aus einer Datenquelle wieder zu laden und als Patch auszuführen.

In diesem Abschnitt wird die Repräsentation des TGraph-Deltas anhand von Beispielen erläutert. Eine formale Spezifikation des TGraph-Delta Formats befindet sich in Anhang A.3. Dieses Format wird für die persistente Speicherung von TGraph-Deltas verwendet.

4.4.1. Das TGraph-Delta Format

Um ein TGraph-Delta persistent zu speichern und zu einem späteren Zeitpunkt wieder zu laden und zu verwenden, muss eine geeignete Form der Repräsentation spezifiziert werden. Diese externe Repräsentation kann zum Austausch von TGraph-Deltas verwendet werden. Sie soll die Anforderungen an die Delta-Repräsentation erfüllen.

Alle benötigten Informationen eines TGraph-Deltas müssen in einer eindeutigen, standardisierten Form repräsentiert werden, siehe Anforderung 7. Das System muss in der Lage sein, ein persistent gespeichertes Delta zu einem beliebigen Zeitpunkt zu laden und ausführen zu können.

Zudem soll die externe Repräsentation für einen Anwender möglichst verständlich sein. Ein menschlicher Akteur soll in der Lage sein, die durch das TGraph-Delta beschriebenen Änderungen an einem TGraphen nachvollziehen zu können. Siehe Anforderung 8.

Eine weitere wichtige Anforderung stellt die Speicherplatz-Effizienz dar. Ein TGraph-Delta soll möglichst kompakt an einem persistenten Speicherplatz abgelegt werden können, Anforderung 9.

Das folgende Listing zeigt ein Beispiel für das TGraph-Delta Format.

```
1  // JGraLab - The Java graph laboratory
2  //   Version : Dimetrodon
3  //   Revision: 2726
4  //   Build ID: 69
5
6  // TGraphDiffutils
7  // Version: 1.0.3
8
9  Graph "StateChart01" Version:50 StatechartDiagram
10 AddVertex v0t structure.Region ;
11 AddVertex v1t structure.CompositeState ;
12 UpdateAttributeValue v1t name "Dispensing" ;
13 AddVertex v3t structure.SimpleState ;
14 UpdateAttributeValue v3t name "Refunding Change" ;
15 UpdateOmega e10 v3t ;
16 UpdateAlpha e16 v8 ;
17 UpdateOmega e16 v1t ;
18 UpdateOmega e11 v9 ;
19 UpdateAlpha e17 v1t ;
20 UpdateOmega e17 v3t ;
21 UpdateOmega e8 v1t ;
22 AddEdge e12t structure.ContainsCompositeStateRegion from
     v1t to v0t ;
23 AddEdge e13t structure.Transition from v5 to v8 ;
24 UpdateAttributeValue e13t effect "hold coin in temp bin" ;
25 UpdateAttributeValue e13t guard "deposited funds >= drink
     cost" ;
26 UpdateAttributeValue e13t trigger "deposit coin" ;
27 AddEdge e17t structure.Transition from v3t to v6 ;
28 UpdateAttributeValue e17t trigger "change returned" ;
29 PutVertexAfter v0t v8 ;
30 PutVertexAfter v1t v0t ;
31 PutVertexAfter v3t v1t ;
32 PutEdgeAfter e17t e17 ;
33 PutEdgeAfter e13t e14 ;
34 PutEdgeAfter e12t e7 ;
35 DeleteEdge e9 ;
36 DeleteEdge e15 ;
37 DeleteEdge e18 ;
38 DeleteVertex v10 ;
39 DeleteVertex v11 ;
40 DeleteVertex v12 ;
```

Listing 4-1: Ein Beispiel für das TGraph-Delta Format

In den ersten sieben Zeilen enthält das TGraph-Delta Format Informationen über die verwendete JGraLab- und TGraph-Diffutils-Version. Diese Informationen werden in Form von Kommentaren dargestellt. In der 9. Zeile stehen obligatorische Informationen zu dem linken TGraphen. Neben dem Bezeichner des TGraphen, wird seine Versionsnummer und der Bezeichner des Schemas aufgeführt. Diese Informationen können dafür genutzt werden, um festzustellen für welchen TGraphen das Delta konstruiert wurde.

Nach der 9. Zeile folgen die Änderungsanweisungen des Deltas. Jede Anweisung beginnt mit einem Bezeichner, der die Anweisung eindeutig identifiziert, und endet mit einem Semikolon. Die Bezeichner der Änderungsanweisungen wurden so gewählt, dass sie eindeutig und verständlich sind.

Jede Anweisung bezieht sich auf mindestens ein Element des TGraphen. Die Darstellungsweise orientiert sich hierbei an dem TG2-Dateiformat. Kanten werden mit einem vorangestellten `e` und ihrer Identifkationsnummer dargestellt. So steht zum Beispiel der Bezeichner `e12` für die Kante mit der Identifkationsnummer 12 in dem TGraphen. Knoten werden mit einem vorangestellten `v`, gefolgt von ihrer Identifkationsnummer repräsentiert. So steht beispielsweise `v3` für den Knoten mit der Identifkationsnummer 3.

Bei der Repräsentation der Elemente weist das TGraph-Delta Format eine Besonderheit gegenüber dem TG2-Dateiformat auf. Es gibt temporäre Bezeichner für Graphenelemente. Diese temporären Bezeichner sind notwendig, da ein TGraph-Delta neue Elemente zu einem TGraphen hinzufügen kann. Den neuen Elementen wird erst zum Zeitpunkt ihrer Generierung eine Identifkationsnummer durch JGraLab zugewiesen. Das Delta kann den Wert ihrer Identifkationsnummer nicht vorhersehen oder beeinflussen. Aus diesem Grund werden Elementen, die durch das Patchen des TGraphen mit dem Delta erst entstehen, temporäre Bezeichner vergeben.

Ein temporärer Bezeichner für eine Kante beginnt mit einem `e`, gefolgt von einer temporären Identifkationsnummer und dem Suffix `t`. So steht zum Beispiel in Zeile 22 von Listing 4-1 eine `AddEdge` Anweisung zum Hinzufügen einer Kante. Dieser neuen Kante wird der temporäre Bezeichner `e12t` zugewiesen. Analog hierzu wird der Bezeichner für neue Knoten mit dem Präfix `v`, gefolgt von der Identifkationsnummer und dem Suffix `t` gebildet.

Die temporären Bezeichner für neue Graphenelemente sind wichtig, damit sich innerhalb des Deltas weitere Änderungsanweisungen auf die neuen Elemente beziehen können. So nimmt z.B. in Zeile 17 die Anweisung Bezug auf den neuen Knoten mit dem temporären Bezeichner `v1t`. Diese Anweisung besagt, dass die Kante mit dem Bezeichner `e16` nach der Ausführung der Anweisung auf den Knoten mit dem Bezeichner `v1t` zeigen soll. Für den TGraphen hat dies eine Veränderung der Inzidenzen zur Folge. Ohne temporäre Bezeichner könnten derartige Veränderungen nicht ausgedrückt werden.

Um die Repräsentation des TGraph-Deltas kompakt zu halten, beinhaltet es nur die Informationen, die für die Ausführung der Anweisungen benötigt werden. So

enthält beispielsweise die Anweisung `DeleteVertex v11;` in Zeile 39 lediglich die Information über den Bezeichner des zu löschenden Knotens. Seine Attribute, Inzidenzen und der Typ sind nicht im Delta enthalten.

4.5. Die Repräsentation der TGraph-Differenz

Die TGraph-Diffutils ermöglichen den Vergleich zweier TGraphen. Das Ergebnis dieser Berechnung ist die TGraph-Differenz. Diese liegt intern als Datenstruktur vor. Ein Anwender kann sich die TGraph-Differenz in einem ASCII-Format ausgeben lassen.

Diese textuelle Repräsentation soll Anwendern einen einfachen Überblick über die Abweichungen zweier TGraphen ermöglichen. Im Gegensatz zu dem TGraph-Delta Dateiformat, kommt es bei der TGraph-Differenz nicht auf eine kompakte Speicherung der Daten an. Die TGraph-Differenz soll dem Anwender ausreichende Informationen liefern. Eine formale Spezifikation des TGraph-Differenz Formats befindet sich in Anhang A.4.

In dem folgenden Listing wird ein Beispiel für das TGraph-Differenz Format aufgezeigt. Es wurde ein Beispiel gewählt, bei dem sich zwei TGraphen hinsichtlich ihrer Elemente, ihrer Inzidenzen, ihrer Attribute, ihrer Typisierung und ihrer Anordnung unterscheiden.

```
1  // JGraLab - The Java graph laboratory
2  //   Version : Dimetrodon
3  //   Revision: 2726
4  //   Build ID: 69
5
6  // TGraphDiffutils
7  // Version: 1.0.3
8
9  //ElementDifference:
10 //Unmatched Edges of the left TGraph :
11 +e9: structure.ContainsDoActivity
12 //Unmatched Edges of the right TGraph :
13 +e8: structure.ContainsCompositeStateRegion
14
15
16 //IncidenceDifference:
17 //Abbreviations: a(e) := alpha(e) o(e) := omega(e)  m(v) := match(v)
18 //AlphaVertex deviations:
19 eL          | a(eL)      | m(a(eL)    | eR          | a(eR)      | m(a(eR))
20 ------------+------------+------------+-------------+------------+------------
21 e6          | v8         | v11        | e6          | v8         | v11
22 //OmegaVertex deviations:
23 eL          | o(eL)      | m(o(eL)    | eR          | o(eR)      | m(o(eR))
24 ------------+------------+------------+-------------+------------+------------
25 e10         | v11        | v8         | e11         | v12        | v9
26 e7          | v8         | v11        | e7          | v8         | v11
27 e8          | v9         | v12        | e10         | v11        | v8
28 e15         | v8         | v11        | e15         | v8         | v11
29 e17         | v6         | v6         | e17         | v11        | v8
30 e18         | v11        | v8         | e18         | v6         | v6
31
32
33 //AttributeDifference:
34 left El.   | right El.  | Attr.   | left Value                      | right Value
35 -----------+------------+---------+---------------------------------+----------
```

```
v11        | v8         | name    | "Waiting for Confirmation"      | "Waiting for
    Selection"
-----------+------------+---------+---------------------------------+----------
v8         | v11        | name    | "Waiting for Selection"         | "Refunding Change"
-----------+------------+---------+---------------------------------+----------
e15        | e15        | guard   | "deposited funds == drink cost" | "deposited funds
    >= drink cost"
-----------+------------+---------+---------------------------------+----------
e18        | e18        | trigger | "select a drink"                | "change returned"
-----------+------------+---------+---------------------------------+----------


//TypeDifference:
Left Graph                      | Right Graph
--------------------------------+--------------------------------
v10: structure.Activity         | v9: structure.Region
v12: structure.SimpleState      | v10: structure.CompositeState


//OrderDifference:
v10 put after v8
v11 put after v7
v8 put after v10
v12 put after v9
e12 put after e11
e8 put after e9
e11 put after e7
```

Listing 5-1: Ein Beispiel für das TGraph-Differenz Format

Analog zu der Repräsentation des TGraph-Delta, beginnt das TGraph-Differenz Format in den ersten sieben Zeilen mit Informationen zu der verwendeten JGraLab- und TGraph-Diffutils-Version. Die Element-Differenz, die Inzidenz-Differenz, die Attribut-Differenz, die Typ-Differenz und die Anordnungs-Differenz werden jeweils in einem eigenen Bereich dargestellt. Gibt es keine Abweichung hinsichtlich einer bestimmten Eigenschaft, so wird der entsprechende Block nicht dargestellt.

Bei dem Entwurf des Formats stand die Verständlichkeit im Vordergrund. Dies wurde in Anforderung 13 festgehalten. Eine spätere Version der TGraph-Diffutils könnte um eine bessere Form der grafischen Darstellung der TGraph-Differenz erweitert werden, da die Ausgabe auf der Konsole die Darstellung in ihren Möglichkeiten eingrenzt.

KAPITEL 5

IMPLEMENTIERUNG

Der Entwurf und die Spezifikation dienen als Ausgangspunkt für die Implementierung. Das folgende Kapitel soll dem Leser einen Einblick in die Implementierung der TGraph-Diffutils ermöglichen.

Der Source Code der TGraph-Diffutils umfasst 68 Klassen mit insgesamt 8603 Zeilen Code.[1] Kommentare und Leerzeilen wurden dabei nicht mitgezählt. Ebenso wurde der Test Code nicht berücksichtigt.

Viele der Bausteine wurden bereits hinreichend beschrieben. Dieses Kapitel soll Entwicklern einen tieferen Einblick in die Funktionsweise der TGraph-Diffutils ermöglichen. Wichtige Klassen und Methoden im Source Code werden erläutert. Zusammenhänge werden aufgezeigt und Entscheidungen bezüglich der Umsetzung begründet.

5.1. Das Mapping von TGraphen

Das Paket `de.uni_koblenz.jgralab.utilities.tgraphdiffutils.core.mapping` enthält die Klassen, die für die Repräsentation und Berechnung des Mappings genutzt werden.

5.1.1. Die Datenstruktur

Das **Mapping von TGraphen** wird innerhalb der TGraph-Diffutils durch die Klasse `Mapping` repräsentiert. Die formale Spezifikation in Anhang A.1.1 auf Seite 151 bildet die Grundlage für die Implementierung dieser Klasse.

[1] Stand: 1. Juli 2010Version 1.0.3

Eine Instanz der Klasse `Mapping` kann durch den Konstruktor, der als Argument zwei Instanzen vom Typ `Graph` erwartet, erzeugt werden. Das folgende Listing zeigt die Signatur des Konstruktors.

```
1 public Mapping(Graph leftGraph, Graph rightGraph)
```

Im Entwurf wurde das `Mapping` als ein **Value Object** klassifiziert. Es wird somit durch seine Attribute definiert. Zwei `Mappings` sind gleichwertig, wenn ihre linken und rechten TGraphen, als auch die partiellen Injektionen gleichwertig sind. Bei der Implementierung der Operationen `hashCode()` und `equals(Object)` wurde auf die Einhaltung des von Java vorgeschriebenen Vertrags geachtet [jav06].

Eine weitere Konsequenz des Entwurfs ist, dass ein Mapping nach seiner Erzeugung unveränderlich sein muss. Die Operationen `addEdgeMatch(Edge, Edge)` und `addVertexMatch(Vertex, Vertex)` ermöglichen jedoch eine Veränderung des Mappings. Diese Operationen werden für die Erzeugung eines Mappings benötigt und sollten nur durch eine `MatchingStrategy` aufgerufen werden. Nach der Berechnung eines `Mappings` durch eine `MatchingStrategy` dürfen keine Veränderungen an dem `Mapping` mehr durchgeführt werden. Diese Bedingung wird nicht explizit geprüft.

Bei der Erzeugung eines `Mappings` müssen bestimmte Invarianten eingehalten werden, wie in der Spezifikation angegeben. Dadurch soll sichergestellt werden, dass die beiden Attribute `mE` und `mV` partielle Injektionen sind, wie in Definition 10 auf Seite 34 gefordert. Zudem muss das erste Argument der Operationen `addEdgeMatch(Edge, Edge)` und `addVertexMatch(Vertex, Vertex)` immer ein Element des linken TGraphen sein und das zweite Argument immer ein Element des rechten TGraphen. Ein Entwickler muss bei der Entwicklung einer `MatchingStrategy` darauf achten, dass diese Operationen eine `IllegalArgumentException` auslösen, wenn eine der Bedingungen nicht eingehalten wird. Diese Vorgehensweise ist notwendig, da fehlerhafte `Mapping` Instanzen unvorhergesehene Fehler in anderen Teilen des Programms auslösen können.

Für den Zugriff auf den Inhalt eines `Mappings`, kann eine der Operationssignaturen aus dem folgenden Listing verwendet werden.

```
1 public Edge getEdgeMatch(Edge edge)
2 public Vertex getVertexMatch(Vertex vertex)
3 public SortedSet<Edge> getMatchedEdgesInLeftGraph()
4 public SortedSet<Edge> getMatchedEdgesInRightGraph()
5 public SortedSet<Vertex> getMatchedVerticesInLeftGraph()
6 public SortedSet<Vertex> getMatchedVerticesInRightGraph()
7 public Graph getLeftGraph()
8 public Graph getRightGraph()
```

Listing 1-1: Signaturen: Zugriff auf den Inhalt des `Mapping`

Die Operationen `getLeftGraph()` und `getRightGraph()` liefern den linken und den rechten `Graph` des `Mappings`.

Die Operation `getEdgeMatch(Edge)` liefert zu einer `Edge e1` eine `Edge e2` in dem anderen `Graph`, die ihr durch das `Mapping` zugeordnet wird. Gibt es zu `e1` keine Übereinstimmung, so liefert die Operation **`null`** zurück.

Analog dazu liefert die Operation `getVertexMatch(Vertex)` zu einem `Vertex v1` einen `Vertex v2` in dem anderen `Graph`, der `v1` durch das `Mapping` zugeordnet wird. Gibt es zu `v1` keine Übereinstimmung, so liefert die Operation **`null`** zurück.

Durch den Aufruf der Operation `getMatchedEdgesInLeftGraph()` wird ein `SortedSet` derjenigen `Edges` des linken `Graph` zurückgegeben, die durch die Berechnung des `Mappings` einer `Edge` des rechten `Graph` zugeordnet werden konnten. Gleiches gilt für die Operation `getMatchedEdgesInRightGraph()`, welche diejeniegen `Edges` des rechten `Graph` als `SortedSet` zurückliefert, denen eine `Edge` des linken `Graph` durch das `Mapping` zugeordnet wird. Die Sortierung der `SortedSets` beruht auf der Implementierung der Operation `compareTo(AttributedElement)` in der Klasse `EdgeBaseImpl`.

Analog dazu liefern die Operationen `getMatchedVerticesInLeftGraph()` und `getMatchedVerticesInRightGraph()` einen `SortedSet` der durch das `Mapping` zugeordneten `Vertices` des linken, bzw. des rechten `Graph`.

Die `toString()` Operation liefert eine verständliche Repräsentation des `Mappings`, wie in dem folgenden Beispiel zu sehen ist. Die Informationen werden in Form einer ASCII-Tabelle ausgegeben.

```
1  Matched Vertices:
2  Left Graph             | Right Graph
3  -----------------------+-----------------------
4  v3: structure.City     | v2: structure.City
5  v1: structure.City     | v1: structure.City
6
7  Matched Edges:
8  Left Graph              | Right Graph
9  ------------------------+------------------------
10 +e1: structure.Motorway | +e2: structure.Motorway
11 +e4: structure.Street   | +e1: structure.Motorway
```

Listing 1-2: Textuelle Repräsentation des `Mappings`

5.1.2. Die Berechnung des Mappings

In Abschnitt 4.3 wurde der Entwurf des **Map Service** erläutert. Dieser Service wird durch das Interface `MapService` deklariert. Dieses befindet sich in dem Paket `de.uni_koblenz.jgralab.utilities.tgraphdiffutils.service` und

repräsentiert die öffentliche Schnittstelle des **Map Service**. Sie wird durch die Klasse `MapServiceImpl` realisiert. Diese Klasse befindet sich ebenso wie das Interface `MatchingStrategy` in dem Paket `de.uni_koblenz.jgralab.utilities.tgraphdiffutils.core.mapping`. Ebenso muss sich jede Implementierung der `MatchingStrategy` in diesem Paket befinden. Dies ist notwendig, um den Zugriff auf das `Mapping` einzuschränken. Das Mapping stellt Operationen zur Veränderung bereit, diese sollen aber ausschließlich bei dessen Berechnung durch eine `MatchingStrategy` verwendet werden. Für einen Klienten muss ein `Mapping` unveränderlich sein.

Die Klasse `MapServiceImpl` implementiert das `MapService` Interface und setzt das **Strategy Pattern** um. Die Berechnung des Mappings erfolgt durch eine `MatchingStrategy`.

5.1.3. Beispiele für die Implementierung der MatchingStrategy

Im Fokus dieser Diplomarbeit stand nicht die Entwicklung einer `MatchingStrategy`. Für den Test der TGraph-Diffutils und erste Anwendungen wurden dennoch drei unterschiedliche `MatchingStrategies` implementiert. Es wurden mehrere `MatchingStrategies` entwickelt, um die Austauschbarkeit der `MatchingStrategy` und deren Auswahl durch einen Anwender zu testen. Außerdem ermöglichen es die verschiedenen Implementierungen die Auswirkungen der `MatchingStrategy` auf das gesamte System zu untersuchen. Dadurch können wichtige Erkenntnisse gewonnen werden, die für die Weiterentwicklung der TGraph-Diffutils wichtig sind.

Die IDMatchingStrategy

Die `IDMatchingStrategy` wird hier aufgeführt, um exemplarisch die Implementierung einer `MatchingStrategy` zu veranschaulichen. Sie ist im Gegensatz zu dem `SimpleFloMatcher` mit 612 Zeilen und der `TypeAttrMatchingStrategy` mit 284 Zeilen sehr überschaubar.

```
1  public final class IDMatchingStrategy implements
       MatchingStrategy {
2
3    private static final String STRATEGY_ID = "IDMatching";
4
5    @Override
6    public String getIdentifier() {
7      return STRATEGY_ID;
8    }
9
10   @Override
11   public Mapping match(Graph leftTGraph, Graph rightTGraph) {
```

```
12      Mapping mapping = new Mapping(leftTGraph, rightTGraph);
13      ProgressFunction progressFunction = TGDiffUtils.
           getProgressFunction();
14      StatusFunction statusFunction = TGDiffUtils.
           getStatusFunction();
15      progressFunction.init(leftTGraph.getVCount() + leftTGraph.
           getECount());
16      statusFunction.writeStatus("Computing the mapping : ");
17      for (Vertex v : leftTGraph.vertices()) {
18        progressFunction.progress(1);
19        if (rightTGraph.getVertex(v.getId()) != null) {
20          mapping.addVertexMatch(v, rightTGraph.getVertex(v.getId
               ()));
21        }
22      }
23      for (Edge e : leftTGraph.edges()) {
24        progressFunction.progress(1);
25        if (rightTGraph.getEdge(e.getId()) != null) {
26          mapping.addEdgeMatch(e, rightTGraph.getEdge(e.getId()));
27        }
28      }
29      progressFunction.finished();
30      return mapping;
31    }
32
33  }
```

Listing 1-3: Die Implementierung der `IDMatchingStrategy`

In Zeile 12 wird das `Mapping` initialisiert. Die Anweisungen zur Berechnung des `Mappings` befinden sich in den Zeilen 17 bis 28. Alle Elemente des linken `Graph` werden durchlaufen. Dabei wird für jedes Element geprüft, ob es in dem rechten `Graph` ein Element mit der selben Identifikationsnummer gibt. Wird ein Paar mit gleichen Identifikationsnummern gefunden, so wird es dem `Mapping` hinzugefügt.

Da Elemente nur anhand ihrer Identifikationsnummer einander zugeordnet werden, kann das durch die `IDMatchingStrategy` berechnete `Mapping` sehr ungenau sein.

Die TypeAttrMatchingStrategy

Ein weiteres Verfahren zur Berechnung des `Mappings` wird durch die `TypeAttrMatchingStrategy` realisiert. Im Vergleich zu der `IDMatchingStrategy` soll diese genauere `Mappings` berechnen.

Diese `MatchingStrategy` arbeitet nach einem einfachen Prinzip. Für jeden `Vertex` und jede `Edge` des linken `Graph` sucht das Verfahren in dem rechten `Graph` nach einer Übereinstimmung. Zwei Elemente stimmen bei diesem Verfah-

ren überein, wenn sie den gleichen Typ und die gleichen Attributwerte aufweisen.

Diese Vorgehensweise ist in Listing 1-4 in den Zeilen 21 bis 37 dargestellt. Da jeder `Vertex` des linken `Graphen` mit jedem `Vertex` des rechten `Graphen` verglichen wird, ist der gesamte Aufwand des Algorithmus quadratisch in Abhängigkeit von der Größe der `Graphen`.

Um die Laufzeit der `TypeAttrMatchingStrategy` zu verbessern wird eine Heuristik verwendet. In den meisten Anwendungsfällen werden zwei unterschiedliche Versionen des gleichen `Graphen` miteinander verglichen. In diesem Fall ist es nicht ungewöhnlich, dass einige Elemente der beiden Versionen identisch sind und die gleiche Identifikationsnummer aufweisen. Dies wird in den Zeilen 1 bis 19 des Algorithmus genutzt, um mit linearem Aufwand einen Großteil der Elemente frühzeitig zuzuordnen. Bereits zugeordnete Elemente müssen anschließend nicht mehr berücksichtigt werden.

Tests haben gezeigt, dass diese Heuristik in vielen Fällen das Laufzeitverhalten der `TypeAttrMatchingStrategy` wesentlich verbessert.

```
1  for (Vertex vLeft : leftTGraph.vertices()) {
2    Vertex vRight = rightTGraph.getVertex(vLeft.getId());
3    if (vRight != null && isMatch(vLeft, vRight)) {
4      unmatchedVerticesRight.remove(vRight);
5      m.addVertexMatch(vLeft, vRight);
6    } else {
7      unmatchedVerticesLeft.add(vLeft);
8    }
9  }
10
11 for (Edge eLeft : leftTGraph.edges()) {
12   Edge eRight = rightTGraph.getEdge(eLeft.getId());
13   if (eRight != null && isMatch(eLeft, eRight)) {
14     unmatchedEdgesRight.remove(eRight);
15     m.addEdgeMatch(eLeft, eRight);
16   } else {
17     unmatchedEdgesLeft.add(eLeft);
18   }
19 }
20
21 for (Vertex vl : unmatchedVerticesLeft) {
22   for (Vertex vr : rightTGraph.vertices(vl.getClass())) {
23     if (unmatchedVerticesRight.contains(vr) && isMatch(vl, vr))
          {
24       m.addVertexMatch(vl, vr);
25       unmatchedVerticesRight.remove(vr);
26     }
27   }
28 }
29
30 for (Vertex vl : unmatchedVerticesLeft) {
31   for (Vertex vr : rightTGraph.vertices(vl.getClass())) {
```

```
32      if (unmatchedVerticesRight.contains(vr) && isMatch(vl, vr))
           {
33        m.addVertexMatch(vl, vr);
34        unmatchedVerticesRight.remove(vr);
35      }
36    }
37  }
```

Listing 1-4: Auszug aus der Implementierung der `TypeAttrMatchingStrategy`

Der SimpleFlowMatcher

Der `SimpleFlowMatcher` basiert auf der Idee, das Problem der Berechnung des `Mappings` mit Hilfe eines Verfahrens zur Lösung des **Netzwerkflussproblems** zu lösen.

Problem:	Netzwerkflussproblem
Input:	Ein gerichteter Graph *G*, dessen Kanten gewichtet sind. Ein Source-Knoten s und ein Sink-Knoten t.
Output:	Ein maximaler Fluss von s nach t.

Dazu werden der linke und der rechte `Graph` in einem `TGNet Graph` zusammengefasst. Das folgende Listing zeigt das verwendete TGraph-Schema.

```
1  TGraph 2;
2
3  Schema de.uni_koblenz.jgralab.utilities.tgraphdiffutils.service.
      map.simplenetworkflow.NetworkFlow;
4
5  GraphClass TGNet;
6
7  Package ;
8  Package structure;
9  EdgeClass Pipe from Reference (0,*) to Reference (0,*) {weight:
      Double};
10 EdgeClass VertexPipe from VertexReference (0,*) to
      VertexReference (0,*) {weight: Double};
11 EdgeClass EdgePipe from EdgeReference (0,*) to EdgeReference
      (0,*) {weight: Double};
12 VertexClass Reference;
13 VertexClass VertexReference: Reference {id: Integer};
14 VertexClass EdgeReference: Reference {id: Integer};
```

Listing 1-5: Das TGraph-Schema zur Repräsentation des Netzwerkflusses

Bei der Erzeugung dieses `TGNet Graph` wird zu jeder `Edge e` der beiden `Graphen` ein `Vertex` vom Typ `EdgeReference` erzeugt. Jeder `EdgeReference Vertex` erhält als Attribut die Identifikationsnummer der `Edge e`, die er repräsentiert.

Ebenso wird jeder `Vertex v` der beiden `Graphen` in dem `TGNet Graph` durch einen `Vertex` vom Typ `VertexReference` repräsentiert. Auch dieser enthält als einziges Attribute die Identifikationsnummer des `Vertex v`.

Zudem wird ein einziger `Vertex` erzeugt, der als Source-`Vertex` verwendet wird. Für jeden `VertexReference vLeft`, der einen `Vertex` des linken `Graphen` repräsentiert, wird eine `Pipe` ausgehend von dem Source-`Vertex` zu `vLeft` erzeugt. Es wird ein einziger Sink-`Vertex` erzeugt. Für jeden `VertexReference vRight`, der einen `Vertex` des rechten `Graphen` repräsentiert, wird eine `Pipe` ausgehend von `vRight` zu dem Sink-`Vertex` erzeugt.

Anschließend wird ausgehend von jedem `VertexReference`, der einen `Vertex` des linken `Graphen` repräsentiert, eine `VertexPipe` zu jedem `VertexReference`, der einen `Vertex` des rechten `Graphen` repräsentiert, erzeugt. Jede `VertexPipe` wird dabei gewichtet, indem dem Attribut `weight` ein Wert zwischen 0 und 1 zugewiesen wird, der die Ähnlichkeit der beiden `Vertices` repräsentiert.

Das gleiche Prozedere wird für alle `Edges` des linken und des rechten `Graph` durchlaufen. Dabei wird für jede `Edge` eine `EdgeReference` erzeugt und diese durch gewichtete `EdgePipes` verbunden.

Das Ergebnis ist ein `TGNet Graph`, wie er exemplarisch in Abbildung 5.1 zu sehen ist. In der Abbildung steht der `Vertex v1` für den Source-`Vertex` und der `Vertex v2` für den Sink-`Vertex`.

Die Berechnung des maximalen Netzflusses erfolgt anhand der Gewichtungen der Kanten. Je höher die Gewichtung desto höher der mögliche Durchfluss.

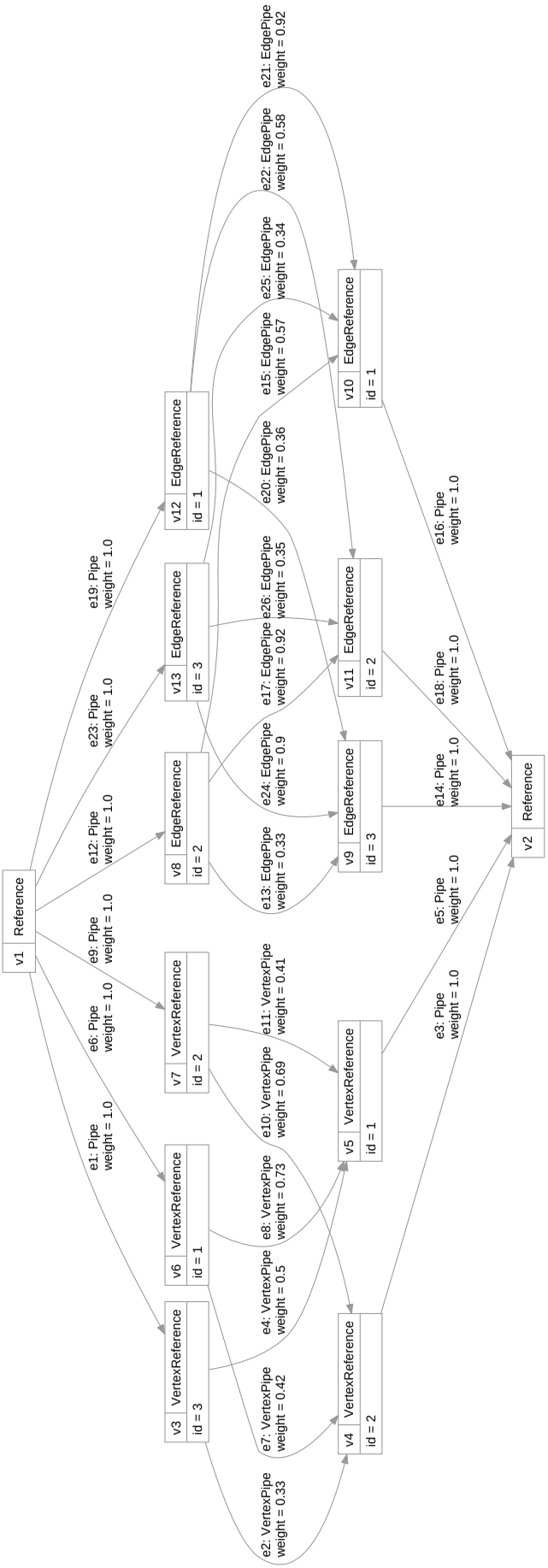


Abbildung 5.1.: Ein Beispiel für einen `TGNet Graph`

5.1.4. Die Erweiterung und Anpassung

Um die TGraph-Diffutils um ein neues Verfahren zur Berechnung des `Mappings` zu erweitern, muss ein Entwickler eine Klasse bereitstellen, die das `MatchingStrategy` Interface implementiert. Diese Klasse muss sich in dem Paket `de.uni_koblenz.jgralab.utilities.tgraphdiffutils.core.mapping` befinden. Die TGraph-Diffutils stellen jede `MatchingStrategy`, die sich in diesem Paket befindet, dem Anwender zur Auswahl bereit.

Jedes `MatchingStrategy` muss durch die Operation `getIdentifier()` einen eindeutigen Bezeichner zurückliefern. Dieser Bezeichner wird von den TGraph-Diffutils genutzt, um die registrierten `MatchingStrategies` zu verwalten. Einem Anwender, der die Werkzeuge von der Konsole aus nutzen möchte, wird die Auswahl der `MatchingStrategy` durch den eindeutigen Bezeichner ermöglicht.

Das `ServiceRepository`

Die unterschiedlichen Realisierungen der `MatchingStrategy` werden, wie auch alle anderen Services, durch die Klasse `ServiceRepository` verwaltet und bereitgestellt.

Die Klasse wurde als **Enum Singleton** implementiert. Ein Entwickler kann über das Element `ServiceRepository.INSTANCE` auf die einzige Instanz der Klasse zugreifen.

> ...a single-element enum type is the best way to implement a singleton. [Blo08]

Bei der Instanziierung des `ServiceRepository` werden die zur Verfügung stehenden `MatchingStrategies` registriert. Dies geschieht durch die Abarbeitung der Operationen `registerMatchingStrategiesInJar(String)` und `registerMatchingStrategiesInPackage(String)`. Diese Operationen können nur von dem `ServiceRepository` aufgerufen werden. Es besteht die Möglichkeit über den Aufruf der Operation `registerMatchingStrategy(Class<? extends MatchingStrategy>)` eine `MatchingStrategy` manuell zu registrieren.

5.2. Die TGraph-Differenz

Im Folgenden wird die Implementierung der TGraph-Differenz erläutert. Das Paket `de.uni_koblenz.jgralab.utilities.tgraphdiffutils.core.difference` beinhaltet die Klassen, die zur Repräsentation und Berechnung der TGraph-Differenz benötigt werden.

5.2.1. Die Datenstruktur

Der Entwurf der Datenstruktur zur Repräsentation der TGraph-Differenz wurde in Abschnitt 4.2.1 erläutert. Eine formale Spezifikation befindet sich in Anhang A.1.2 auf Seite 152.

Die `Difference` kontrolliert den Zugriff auf die `AttributeDifference`, `ElementDifference`, `IncidenceDifference`, `OrderDifference` und `TypeDifference`. Für jeden Teilbaustein beinhaltet die `Difference` eine Operation, die bei Aufruf dessen Berechnung auslöst. Die Signaturen dieser Operationen sind in Listing 2-1 dargestellt. Sie werden zur Berechnung der `Difference` benötigt.

5.2.2. Die Berechnung

```
1 protected Difference computeAttributeDifference()
2 protected Difference computeElementDifference()
3 protected Difference computeIncidenceDifference()
4 protected Difference computeOrderDifference()
5 protected Difference computeTypeDifference()
6 protected Difference close()
```

Listing 2-1: Signaturen: Berechnung der `Difference`

Für die Berechnung der `Difference` wird der **Compare Service** verwendet. Dessen Schnittstelle wird durch das Interface `CompareService` deklariert. Die Klasse `CompareServiceImpl` implementiert das Interface. Ist die Berechnung der `Difference` abgeschlossen, so kann durch den Aufruf der `close()`-Operation die `Difference` in einen unveränderlichen Zustand überführt werden. Durch diesen Aufruf wird garantiert, dass die `Difference` nach ihrer Berechnung nicht mehr verändert werden kann. Der folgende Aufruf berechnet zum Beispiel die `Difference` der `Graphen sm1` und `sm2` unter dem `Mapping m`. In dem Beispiel wird nur die `ElementDifference` und `AttributeDifference` berechnet.

```
1 Difference difference = new Difference(sm1, sm2, m).
     computeElementDifference().computeAttributeDifference().close
     ();
```

Wie das Beispiel zeigt, können die Operationen zur Berechnung der `Difference` verkettet werden. Erfolgt nach Abschluss der Berechnung durch den Aufruf der `close()`-Operation erneut ein Aufruf einer `compute`-Operation, so wird eine `IllegalStateException` ausgelöst.

Für den Zugriff auf die Informationen der `Difference` können die Operationen in dem folgenden Listing verwendet werden.

```
1 public AttributeDifference getAttributeDifference()
```

```
2 public ElementDifference getElementDifference()
3 public IncidenceDifference getIncidenceDifference()
4 public OrderDifference getOrderDifference()
5 public TypeDifference getTypeDifference()
6 public boolean isEmpty()
7 public Graph getLeftTGraph()
8 public Graph getRightTGraph()
9 public Mapping getMapping()
10 public boolean isImmutable()
```

Listing 2-2: Signaturen: Zugriff auf die Informationen der `Difference`

5.3. Das TGraph-Delta

Der Entwurf in Abschnitt 4.2.2 ist der Ausgangspunkt für die Realisierung des TGraph-Delta. Das TGraph-Delta wurde als **Aggregate** und **Value Object** klassifiziert. Für die Implementierung hat dies folgende Konsequenzen.

Die Änderungsanweisungen eines TGraph-Delta werden durch die Aggregate Root `Delta` kontrolliert und verwaltet. Bei der Implementierung eines `Command` ist deshalb darauf zu achten, dass der direkte Zugriff durch Klienten auf `Commands` nicht möglich ist. Das Hinzufügen und die Ausführung von `Commands` wird durch das `Delta` kontrolliert.

Da es sich um ein **Value Object** handelt wird ein `Delta` durch seine Attribute definiert. Zwei `Deltas` müssen demnach gleichwertig sein, wenn ihre Attribute gleichwertig sind. Außerdem muss ein `Delta` nach seiner Erzeugung unveränderlich sein.

Die Umsetzung dieser Eigenschaften wird im Folgenden erläutert.

5.3.1. Die Datenstruktur

Die Spezifikation des `Delta` befindet sich in Anhang A.3. Sie bildet die Grundlage der Implementierung.

Der Konstruktor des `Delta` erhält als einziges Argument einen `Graph`. Dieser wird durch die Anwendung des `Delta` verändert. Um ein `Command` hinzuzufügen, wird die Operation `addCommand(Command)` verwendet. Diese Operation stellt sicher, dass kein gleichwertiges `Command` mehrmals hinzugefügt wird. Diese Operation darf nur während der Erzeugung des `Deltas` verwendet werden und ist für Klienten nicht sichtbar. Dadurch soll sichergestellt werden, dass ein `Delta` nach dessen Erzeugung nicht mehr verändert wird.

Ein `Delta` verwaltet und kontrolliert eine `List` von `Commands`. Zwei `Deltas` gelten als gleichwertig, wenn zu jedem `Command` des einen `Delta` ein gleichwertiges `Command` in dem jeweils anderen `Delta` existiert.

Ein `Delta` kontrolliert das Hinzufügen von `Commands` und auch deren Ausführung. Durch die Operation `execute()` werden alle Änderungsanweisungen eines `Delta` ausgeführt. Die Operation stellt sicher, dass die Anweisungen in der korrekten Reihenfolge ausgeführt werden. Kann ein `Command` nicht ausgeführt werden, so wird dies im Fehlerprotokoll des `Delta` protokolliert. Die Operation `getErrorLog()` ermöglicht den Zugriff auf das Fehlerprotokoll.

```
1 protected Command getCommand(int commandId)
2 protected int getIdentifier(Command cmd)
3 protected Command getLastCommand()
4 public List<LogEntry> getErrorLog()
5 public Graph getGraph()
6 public boolean isEmpty()
```

Listing 3-1: Signaturen: Zugriff auf die Informationen eines `Delta`

Durch den Aufruf der Operation `isEmpty()` kann festgestellt werden, ob ein `Delta` keine Änderungsanweisungen enthält. Diese Operation wird von dem Programm `TGDiff` benötigt, um den Rückgabewert zu ermitteln. Das Programm liefert den Wert 0, wenn das `Delta` keine Änderungsanweisungen enthält, ansonsten wird der Wert 1 zurückgegeben.

Durch die Operation `getCommand(int)` kann auf ein `Command` über dessen Identifikationsnummer zugegriffen werden. Jedes `Command` erhält bei dessen Erzeugung eine Identifikationsnummer, die innerhalb des `Delta` eindeutig ist. Durch die Operation `getIdentifier(Command)` kann diese Identifikationsnummer für ein `Command` abgefragt werden. Die Operation `getLastCommand()` liefert als Rückgabe das `Command`, das sich am Ende der `Command List` des `Delta` befindet. Durch die Operation `getGraph()` kann auf den `Graph` zugegriffen werden, der durch das `Delta` verändert wird.

Die Änderungsanweisungen

Jede Klasse von Änderungsanweisungen muss das `Command`-Interface aus Listing 3-2 implementieren. Dem Entwurf des TGraph-Delta folgend gehört jedes `Command` zu einem `Delta`. Die Ausführung eines `Command` ist nur durch das `Delta` zulässig. Das `Command`-Interface spezifiziert jedoch die öffentliche Operation `execute()`, deren Aufruf die Ausführung des `Commands` zur Folge hat. Aus diesem Grund ist es wichtig, dass keine Referenzen auf `Commands` an Klienten ausgehändigt werden. Lediglich dem `DeltaBuilder` muss der Zugriff auf `Commands` erlaubt werden, damit er ein `Delta` erzeugen kann. Um den Zugriff auf die `Commands` einzuschränken, wurden alle Operationen, die einen Zugriff ermöglichen, als **`protected`** deklariert. Ein feingranularerer Zugriffsschutz wäre an dieser Stelle wünschenswert.

```
1  public interface Command {
2
3    public void addStateListener(CommandStateListener
        commandStateListener);
4
5    public boolean execute();
6
7    public String getIdentifier();
8
9    public int getLocalId();
10
11   public AttributedElement getResult();
12
13   public boolean isApplied();
14
15   public boolean isIdle();
16
17   public boolean isResolved();
18
19   public int loadCommand(Delta delta, Graph tgraph, String
        currentLine,
20       HashMap<String, Integer> cmdRefMap) throws
            LoadCommandException;
21 }
```

Listing 3-2: Das `Command`-Interface

Die `toString()` Operation muss eine textuelle Repräsentation des `Command` erzeugen. Diese wird verwendet, um das `Command` in einer TGraph-Delta Datei zu speichern. Sie muss mit dem eindeutigen Bezeichner des `Command` beginnen, gefolgt von den Argumenten und wird durch ein Semikolon abgeschlossen.

```
1 AddVertex v1t structure.CompositeState ;
```

Bei der Implementierung der `Command`-Klassen mussten einige Probleme gelöst werden, die beim Entwurf nicht bedacht wurden. Die Abhängigkeiten zwischen `Commands` und die Generierung von `Commands`, die sich auf nicht existierende Graphenelemente beziehen. Beide Probleme stehen in engem Zusammenhang und können anhand des folgenden Beispiels erläutert werden.

```
1 AddVertex v0t structure.CompositeState ;
2 UpdateOmega e8 v0t ;
```

Listing 3-3: Ein Beispiel mit zwei Änderungsanweisungen

Die Anweisung in Zeile 1 besagt, dass ein neuer Knoten mit temporärem Bezeichner `v0t` vom Typ `structure.CompositeState` dem TGraphen hinzugefügt werden soll. Die Anweisung in Zeile 2 drückt aus, dass die Kante mit dem Bezeichner `e8` verschoben werden soll. Nach erfolgreicher Ausführung der Anweisung

zeigt die Kante auf den Knoten mit dem temporären Bezeichner `v0t`. Die Anweisung in Zeile 2 ist somit direkt abhängig von der Ausführung der Anweisung in Zeile 1. Bei der Erzeugung eines `Delta` werden zu diesen beiden Anweisungen `Commands` erstellt. Das `Command` der ersten Anweisung ist eine Instanz von `AddVertex`. Wird dieses `Command` erfolgreich ausgeführt, so erzeugt es einen neuen Knoten. Die Operation `getResult()` ermöglicht es auf diesen Knoten zuzugreifen. Das `UpdateOmega Command` zu der Anweisung in Zeile 2 benötigt zur erfolgreichen Ausführung diesen Knoten. Wenn die Ausführung des `AddVertex Commands` fehlschlägt, kann die Kante durch das `UpdateOmega Command` nicht verschoben werden. Die beiden `Commands` sind voneinander abhängig.

Das zweite Problem betrifft die Tatsache, dass zum Zeitpunkt der Erstellung der beiden `Commands` der Knoten, der durch das `AddVertex Command` erzeugt werden soll, noch nicht existiert. Das `UpdateOmega Command` benötigt aber eine Referenz auf diesen Knoten.

Um beide Probleme zu lösen wurde das **Proxy Pattern** in einer vereinfachten Form verwendet. Der Zweck des Pattern ist der folgende:

> Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.[GHJV95]

Für diesen Zweck wurde das Interface `Proxy<T extends GraphElement>`, siehe Listing 3-4, deklariert. Es wird spezialisiert durch die Interfaces `VertexProxy` und `EdgeProxy`.

```
1 public interface Proxy<T extends GraphElement> {
2
3   public void addChangeListener(ChangeListener changeListener);
4
5   public T getElement();
6
7 }
```

Listing 3-4: Das Interface `Proxy`

Bei der Erzeugung der `Commands` werden keine direkten Referenzen auf `GraphElements` verwendet. Statt dessen werden `Proxys` verwendet. Die Klasse `UpdateOmega` erhält zum Beispiel anstelle einer `Edge` und eines `Vertex`, einen `EdgeProxy` und einen `VertexProxy`. Wenn die `UpdateOmega`-Anweisung ausgeführt werden soll, so greift sie über einen `EdgeProxy` auf die zu verschiebende Kante und einen `VertexProxy` auf den neuen Zielknoten zu, indem sie jeweils die Operation `getElement()` aufruft. Der Rückgabetyp dieser Operation ist generisch. Der `VertexProxy` liefert ein Element vom Typ `Vertex` und der `EdgeProxy` ein Element vom Typ `Edge`.

```
1 public interface VertexProxy extends Proxy<Vertex> {
2   void inject(VertexProxy newVertex);
3 }
4
5 public interface EdgeProxy extends Proxy<Edge> {
```

```
6   void inject(EdgeProxy newEdge);
7 }
```

Listing 3-5: Die Interfaces `VertexProxy` und `EdgeProxy`

Zu beiden Interfaces in Listing 3-5 existieren je zwei Implementierungen. Die Klassen `VertexReferenceProxy` und `VertexPhantomProxy` implementieren das `VertexProxy` Interface. Beide geben bei Aufruf der Operation `getElement()` einen `Vertex` zurück. Ein `VertexReferenceProxy` verweist direkt auf einen `Vertex`. Diese Art von `Proxy` wird für Knoten verwendet, die bereits existieren. Der `VertexPhantomProxy` hingegen verwendet ein `Command`, welches als Ergebnis einen `Vertex` zurückliefert. Dies gilt analog für die Implementierungen des `EdgeProxy`-Interface, die als Stellvertreter für Objekte vom Typ `Edge` verwendet werden.

Durch die Verwendung des **Proxy Pattern** ist es somit möglich `Commands` zu erzeugen, die `GraphElements` verarbeiten, unabhängig davon ob diese Elemente bereits im TGraphen existieren oder erst durch ein vorangehendes `Command` erzeugt werden. So erwartet zum Beispiel der Konstruktor des `UpdateOmega Command` als Argumente einen `EdgeProxy` und einen `VertexProxy`.

```
1 protected UpdateOmega(Graph g, EdgeProxy eProxy, VertexProxy
     vProxy)
```

Listing 3-6: Der Konstuktor des `UpdateOmgega Commands`

Das Problem der Abhängigkeiten wird mit Hilfe der `Proxys`, dem **Flyweight Pattern** und dem **Observer Pattern** gelöst. Die `Proxys` werden über die `ProxyFactory` erzeugt. Diese als **Enum Singleton** realisierte Klasse stellt sicher, dass zu jedem `Vertex` und jeder `Edge` ein einziger `Proxy` erzeugt wird. Zwei `Commands` die das gleiche `GraphElement` verarbeiten, erhalten somit den gleichen `Proxy`. Die `Commands` können sich durch die Operation `addChangeListener(ChangeListener)` an den `Proxys` als Observer registrieren. Wird ein `GraphElement` verändert, so informieren die `Proxys` die registrierten `ChangeListener`. Um den `Commands` mitzuteilen, welche Art der Veränderung stattgefunden hat, werden die Elemente der Enumeration `ChangeType` verwendet. Die folgenden Werte können bei einer Veränderung auftreten:

ADDED. Das Element wurde hinzugefügt. Dieser Fall tritt zum Beispiel nach der Ausführung eines `AddVertex Commands` ein.

DELETED. Das Element wurde gelöscht. Dieser Fall tritt zum Beispiel nach der Ausführung eines `DeleteVertex Commands` ein.

REDIRECTED. Das Element wurde ersetzt. Dieser Fall tritt zum Beispiel nach der Ausführung eines `UpdateType Commands` ein, da zur Ausführung dieser Änderungsanweisung das Element ersetzt werden muss.

In dem Beispiel aus Listing 3-3, ist das `UpdateOmega Command` abhängig von der Ausführung des `AddVertex Commands`. Wenn die Ausführung des `AddVertex`

Commands nicht erfolgreich abgeschlossen wird, kann auch das UpdateOmega Command nicht ausgeführt werden. Beide Commands referenzieren den gleichen VertexProxy, der stellvertretend für den Vertex mit dem temporären Bezeichner v0t steht. Nach der erfolgreichen Ausführung des AddVertex Commands wird der VertexProxy benachrichtigt. Dieser informiert seinerseits alle registrierten ChangeListener durch das Element ChangeType.ADDED darüber, dass der Vertex hinzugefügt wurde. Das UpdateOmega Command erhält diese Nachricht und überprüft seine Abhängigkeiten. Wurden alle Abhängigkeiten aufgelöst, kann das Command seinen Zustand ändern und ist bereit zur Ausführung. Die möglichen Zustände eines Command signalisieren dessen Ausführungsbereitschaft.

Ein Command kann sich in den drei folgenden Zuständen befinden, die durch die Enumeration CommandState spezifiziert werden.

IDLE. Dies ist der Zustand, in dem sich jedes Command unmittelbar nach dessen Initialisierung befindet. In diesem Zustand verbleibt ein Command, bis alle Abhängigkeiten aufgelöst wurden.

RESOLVED. In diesem Zustand befindet sich ein Command, wenn alle Abhängigkeiten aufgelöst wurden. Dieser Zustand signalisiert, dass das Command bereit zur Ausführung ist.

APPLIED. Ein Command, welches erfolgreich ausgeführt wurde, befindet sich in diesem Zustand.

Der Zustand eines Command kann über eine der Operationen isIdle(), isResolved() und isApplied() überprüft werden.

5.3.2. Die Berechnung des Deltas

Die Berechnung des Deltas erfolgt durch den **Diff Service**. Dieser wird durch das Interface DiffService spezifiziert. Das Paket de.uni_koblenz .jgralab.utilities.tgraphdiffutils.core.delta enthält die Klasse DiffServiceImpl, eine Implementierung des DiffService.

Für die Berechnung des Deltas verwendet diese Klasse die Difference. Ausgehend von der Difference werden dem Delta schrittweise Commands hinzugefügt. Dafür wird ein DeltaBuilder eingesetzt. Bei der Erzeugung eines Deltas ist die Reihenfolge, in der die Commands hinzugefügt werden, von entscheidender Bedeutung. So dürfen zum Beispiel AddEdge Commands erst dem Delta hinzugefügt werden, nachdem alle benötigten AddVertex Commands hinzugefügt wurden. Diese Reihenfolge ist wichtig, um beim Patchen ein korrektes Ergebnis zu erhalten.

DeltaBuilder

Ein `DeltaBuilder` kann ein `Delta` erzeugen. Er kann nicht für die Erzeugung mehrerer `Deltas` genutzt werden.

Bei der Berechnung eines `Deltas` durch den `DiffService` werden dem Delta Änderungsanweisungen in Form von `Commands` hinzugefügt. Für jede mögliche Änderungsanweisung enthält die Klasse `DeltaBuilder` Operationen zum Hinzufügen. Jede dieser Operationen beginnt mit dem Präfix `create`, gefolgt von dem Bezeichner der Änderungsanweisung. So existiert zum Beispiel für die Klasse `AddVertex` die Operation `createAddVertex`. Jede `create`-Operation liefert als Ergebnis die Identifikationsnummer des `Command` zurück.

Auf das `Delta`, welches durch einen `DeltaBuilder` erzeugt wird, kann durch die Operation `getDelta()` zugegriffen werden.

Es gibt zwei typische Anwendungsfälle bei denen ein `Delta` erzeugt werden muss. Der `DiffService` berechnet zu zwei `Graph`-Instanzen ein `Delta`. Die TGraph-Diffutils ermöglichen es zudem durch die Klasse `DeltaIO` ein `Delta` aus einer TGraph-Delta Datei zu laden. In beiden Anwendungsfällen kann der `DeltaBuilder` genutzt werden.

5.3.3. Das Laden des Deltas aus einer TGraph-Delta Datei

Die Klasse `DeltaIO` bietet durch die Operation `loadDeltaFromFile(String, Graph)` die Möglichkeit an, ein `Delta` aus einer TGraph-Delta Datei zu laden. Als Argumente erwartet die Operation den Pfad der Datei und den `Graph`, auf den das `Delta` angewendet werden soll. Innerhalb der TGraph-Delta Datei wird jede Änderungsanweisung durch eine Zeichenkette repräsentiert. Die Klasse `DeltaBuilder` implementiert die Operation `loadCommand(String)`, die als Argument eine Zeichenkette entgegennimmt und das darin codierte `Command` dem `Delta` hinzufügt.

Zudem bietet die Klasse `DeltaIO` die Möglichkeit ein `Delta` in einer TGraph-Delta Datei zu speichern. Die Operationssignaturen der Klasse zeigt das folgende Listing.

```
1 public static Delta loadDeltaFromFile(String filename, Graph
     leftTGraph) throws DeltaIOException
2 public static void saveDeltaToFile(String filename, Delta delta)
     throws NullPointerException, DeltaIOException
3 public static void saveDeltaToStream(DataOutputStream out, Delta
     delta) throws DeltaIOException
```

Listing 3-7: Die Operationen zum Laden und Speichern von `Deltas`

5.3.4. Die Erweiterung des TGraph-Delta um zusätzliche Änderungsanweisungen

Jede `Delta` Instanz besteht aus einer sequentiell angeordneten Menge von `Commands`. Jedes `Command` steht für eine bestimmte Änderungsanweisung. Die TGraph-Diffutils basieren auf einem Satz von Änderungsanweisungen. Mit Hilfe dieses Operationssatzes werden `Deltas` generiert.

Entwickler können diesen Operationssatz beliebig erweitern. Um eine Änderungsanweisung einzubringen, muss das Interface `Command` implementiert werden. Entwickler können stattdessen auch eine Subklasse von `ChangeCommand` erstellen. Eine andere Möglichkeit besteht darin die abstrakte Klasse `CompositeChangeCommand` zu erweitern, die als Kompositum genutzt werden kann, um eine komplexere Änderungsanweisung aus einer Sequenz von `Commands` zusammenzusetzen.

Ein wichtiger Punkt bei der Entwicklung eines neuen `Command` ist die Wahl eines Bezeichners. Jedes `Command` muss beim Aufruf der Operation `getIdentifier()` seinen Bezeichner zurückliefern. Dieser Bezeichner muss eindeutig sein, damit die unterschiedlichen `Command`-Klassen innerhalb der TGraph-Diffutils voneinander unterschieden werden können. Zudem muss der Bezeichner einen Rückschluss auf den Zweck des `Commands` ermöglichen, damit ein Anwender die textuelle Repräsentation des `Commands` in einer TGraph-Delta Datei interpretieren kann.

Damit ein neues `Command` von den TGraph-Diffutils verwendet werden kann, muss der `DiffService` und der `DeltaBuilder` angepasst, bzw. erweitert werden.

Der Entwickler kann einen neuen `DiffService` implementieren und in dem zentralen `ServiceRepository` registrieren. Die Registrierung des neuen `DiffService` erfolgt in der aktuellen Version durch eine Änderung der `getDiffService()` Operation. Denkbar wäre auch eine automatische Registrierung, wie es bei den `MatchingStrategies` geschieht. Da dieser Anwendungsfall nach aktueller Einschätzung nur in Spezialfällen auftritt, wurde auf die automatische Registrierung der `DiffServices` verzichtet, da dies auch die Performance der TGraph-Diffutils negativ beeinflussen würde.

Damit ein neu entwickeltes `Command` erzeugt und geladen werden kann, muss der `DeltaBuilder` um zusätzliche `create` Operationen erweitert werden, die das neue `Command` einem `Delta` hinzufügen können. Damit das neue `Command` aus einer TGraph-Delta Datei geladen werden kann, muss die Operation `loadCommand (Delta, Graph, String, HashMap<String, Integer>)` des `Command` Interface implementiert werden. Diese Operation wird von dem `DeltaBuilder` aufgerufen, wenn ein `Command` aus einer Zeichenkette geladen wird.

KAPITEL 6

ANWENDUNG UND INTEGRATION

Das Produktkonzept sieht vor, dass dem Anwender unterschiedliche Werkzeuge zur Verwaltung von TGraphen bereitgestellt werden. In diesem Kapitel wird die Anwendung dieser Werkzeuge beschrieben und anhand von Beispielen erläutert. Zudem wird die Integration der TGraph-Diffutils in Software-Systeme behandelt.

Die Werkzeuge der TGraph-Diffutils können über eine **API** verwendet werden. Diese ermöglicht eine Integration der Werkzeuge in TGraph-basierte Anwendungen.

Zudem stellen die TGraph-Diffutils dem Anwender **Programme** zur Verfügung, die von der **Kommandozeile** aus verwendet werden können. Ein Anwender kann diese Programme verwenden, um TGraphen zu verarbeiten. Für die Verwendung dieser Programme muss sich der Anwender nicht mit der API befassen und kann direkt auf die Werkzeuge zugreifen.

Ein weiterer wichtiger Aspekt ist die Integration in VCS, wie zum Beispiel `svn`.

6.1. Die Verwendung der API

In diesem Abschnitt wird die Verwendung der TGraph-Diffutils API beschrieben. Diese Programmierschnittstelle ermöglicht dem Anwender den Zugriff auf die Funktionen und die Datenstrukturen.

Der direkte Zugriff auf die API erfolgt über die Klasse `TGDiffUtils`. Sie dient als **Facade** und bietet dem Anwender statische Methoden an, die einen einfachen Zugriff auf die Werkzeuge ermöglichen.

6.1.1. Die Gemeinsamkeiten

Beim Aufruf der unterschiedlichen Funktionen gibt es Gemeinsamkeiten.

Die Methoden zum Vergleich von TGraphen, zur Berechnung des TGraph-Delta, zum Patchen von TGraphen und zur Zusammenführung von TGraphen, erwarten als Argument ein oder mehrere Instanzen vom Typ `de.uni_koblenz.jgralab.Graph`. Diese kann der Entwickler mit Hilfe der JGraLab API erzeugen oder aus einer TGraph2-Datei laden. Das folgende Listing zeigt zum Beispiel, wie ein Entwickler mit Hilfe der API zwei TGraphen aus TGraph2-Dateien laden und miteinander vergleichen kann.

In den ersten beiden Zeilen des folgenden Listings werden zwei TGraphen geladen. In der dritten Zeile wird die `compare` Operation, zum Vergleich der beiden TGraphen, aufgerufen.

```
1 Graph tgl = GraphIO.loadSchemaAndGraphFromFile("./SodaMachine01.
     tg", CodeGeneratorConfiguration.MINIMAL,
     NullProgressFunctionImpl.getInstance());
2 Graph tgr = GraphIO.loadSchemaAndGraphFromFile("./SodaMachine02.
     tg", CodeGeneratorConfiguration.MINIMAL,
     NullProgressFunctionImpl.getInstance());
3 Difference difference = TGDiffUtils.compare(tgl, tgr);
```

Listing 1-1: Beispiel: Vergleich von zwei TGraphen

Um den Fortschritt der aktuellen Berechnung anzeigen zu lassen, muss eine `ProgressFunction` erzeugt werden. Eine konkrete Implementierung steht mit der Klasse `ProgressFunctionImpl` in dem Paket `de.uni_koblenz.jgralab.impl` bereit. Es steht jedoch jedem Entwickler offen eine eigene Implementierung des Interface zu verwenden. Über die Operation `setProgressFunction(ProgressFunction)` kann die zu verwendende `ProgressFunction` ausgewählt werden. Wird keine `ProgressFunction` ausgewählt, so wird die `NullProgressFunctionImpl` verwendet. Dabei handelt es sich um eine Implementierung des Null Object Pattern. Es wurde verwendet, um den Code zu vereinfachen und um `NullPointerExceptions` zu vermeiden.

```
1 public static void setProgressFunction(ProgressFunction
     progressFunction)
2 public static void setStatusFunction(StatusFunction
     statusFunction)
3 public static ProgressFunction getProgressFunction()
4 public static StatusFunction getStatusFunction()
```

Listing 1-2: Signaturen: Operationen zum Anzeigen des Fortschritts

Die `ProgressFunction` zeigt den aktuellen Fortschritt der Operation an. Zusätzlich kann der aktuelle Status der Berechnung angezeigt werden. Hierfür kann die Operation `setStatusFunction(StatusFunction)` verwendet werden. Bei Verwendung der `StdoutStatusFunctionImpl`

wird zum Beispiel der aktuelle Status auf die Standardausgabe geschrieben. Diese Klasse befindet sich in dem Paket `de.uni_koblenz.jgralab.utilities.tgraphdiffutils.io`, welches weitere Implementierungen des `StatusFunction` Interface enthält.

Das folgende Listing zeigt die Verwendung der Operationen und die Ausgabe auf der Konsole durch die `ProgressFunctionImpl` und die `StdoutStatusFunctionImpl`.

```
1  TGDiffUtils.setProgressFunction(new ProgressFunctionImpl());
2  TGDiffUtils.setStatusFunction(StdoutStatusFunctionImpl.
       getInstance());
3  Difference difference = TGDiffUtils.compare(tgl, tgr);
4
5  Computing the Mapping :
6  processing 30 elements
7  [############################################################]
8  elapsed time: 0.0s
9  processing 8 elements
10 [############################################################]
11 elapsed time: 0.0030s
12 elapsed time: 0.0050s
13 Computing the ElementDifference
14 processing 60 elements
15 [############################################################]
16 elapsed time: 0.0020s
17 Computing the IncidenceDifference
18 processing 15 elements
19 [############################################################]
20 elapsed time: 0.0010s
21 Computing the AttributeDifference
22 processing 24 elements
23 [############################################################]
24 elapsed time: 0.0020s
25 Computing the type difference
26 processing 24 elements
27 [############################################################]
28 elapsed time: 0.0020s
29 Computing the OrderDifference
30 processing 54 elements
31 [############################################################]
32 elapsed time: 0.0030s
```

Eine weitere Gemeinsamkeit bei der Verwendung der API betrifft den Vergleich von TGraphen und die Berechnung des TGraph-Deltas. In beiden Anwendungsfällen kann der Entwickler entscheiden, welche Eigenschaften von TGraphen berücksichtigt werden sollen und welche vernachlässigt werden können. Diese Eigenschaften werden durch die Enumeration `TGraphProperty` angegeben. So steht beispielsweise das Literal `TGraphProperty.ATTRIBUTE` für die Attributierung von TGraphen. Standardmäßig werden alle Eigenschaften beim Vergleich

und bei der Berechnung des TGraph-Delta berücksichtigt. Der Entwickler kann dies einschränken. Dadurch kann er direkten Einfluss auf den Umfang der Berechnung, die Größe der Ausgabe und die Laufzeit nehmen.

Das folgende Listing zeigt den Vergleich zweier TGraphen, bei dem die Anordnung und Typisierung nicht betrachtet werden. In diesem Beispiel wird der Methode zusätzlich zu den beiden TGraphen drei `TGraphProperty` Literale übergeben.

```
1 Difference difference = TGDiffUtils.compare(tgl, tgr,
2 TGraphProperty.ELEMENT,TGraphProperty.ATTRIBUTE, TGraphProperty.
     INCIDENCE);
```

Listing 1-3: Beispiel: Vergleich zweier TGraphen unter Berücksichtigung der Elemente, der Inzidenzen und der Attributierung

Der Entwickler kann die `MatchingStrategy` auswählen. Der Aufruf `TGDiffUtils.getMatchingStrategies()` liefert ein `java.util.Set` vom Typ `String` zurück. Es beinhaltet die Bezeichner der `MatchingStrategies`, die den TGraph-Diffutils zur Verfügung stehen. Diese Bezeichner werden bei der Auswahl des Verfahrens benötigt. Sie dienen der Unterscheidung zwischen den unterschiedlichen Verfahren.

Um eine bestimmte `MatchingStrategy` für die Berechnung des Mappings zu verwenden, kann der Entwickler der Operation als Argument eine Instanz der zu verwendenden `MatchingStrategy` übergeben. Dieses Argument kann gleichermaßen für die Berechnung des Deltas, als auch für den Vergleich zweier TGraphen verwendet werden. Das folgende Listing zeigt die Verwendung der API zum Vergleich von zwei TGraphen, bei dem die `MatchingStrategy` `SimpleFlowMatcher` eingesetzt wird.

```
1 TGDiffUtils.compare(tgl, tgr, new SimpleFlowMatcher());
```

Listing 1-4: Beispiel: Auswahl des Verfahrens für die Berechnung des Mappings, beim Vergleich zweier TGraphen

Um eine `MatchingStrategy` zu verwenden kann der Entwickler diese selbst initialisieren oder über folgenden Aufruf eine Instanz der `MatchingStrategy` mit dem angegebenen Bezeichner erhalten.

```
1 MatchingStrategy matchingStratgey = TGDiffUtils.
     getMatchingStrategy("SimpleFlowMatcher");
```

In diesem Beispiel versucht der Entwickler eine Instanz der `MatchingStrategy` mit dem Bezeichner `SimpleFlowMatcher` zu erhalten. Wenn es keine `MatchingStrategy` mit dem angegebenen Bezeichner gibt, so wird eine `ServiceAccessException` ausgelöst.

Anmerkung zur Thread-Sicherheit

Die Thread-Sicherheit der TGraph-Diffutils API beruht auf der Tatsache, dass die verwendeten Datenstrukturen nach ihrer Berechnung unveränderlich sind. Wenn zum Beispiel der `CompareService` zur Berechnung der `Difference` ein `Mapping` verwendet, so ist sichergestellt, dass das `Mapping` nicht mehr verändert wird. Dies gilt nicht für die `Graphen`. Die TGraph-Diffutils können nicht sicherstellen, dass während der Berechnung eine `Deltas` zu einem `Graph`, kein anderer `Thread` auf den gleichen `Graph` zugreift und diesen verändert. Dies muss durch den Entwickler sichergestellt werden. Aus diesem Grund ist die Verwendung der API als bedingt Thread-sicher, **conditionally thread-safe**, einzustufen.

Tests haben gezeigt, dass der Einsatz der API bedingt Thread-sicher ist. Ein formaler Beweis wurde im Verlauf dieser Diplomarbeit nicht erstellt.

6.1.2. Ein TGraph-Delta erzeugen

Eine der wesentlichen Funktionen der TGraph-Diffutils ist die Berechnung eines TGraph-Delta zwischen zwei TGraphen. Hierfür stellt die Klasse `TGDiffUtils` verschiedene Operationen bereit. Listing 1-5 zeigt deren Operationssignaturen.

```
1 public static Delta diff(Graph leftTGraph, Graph rightTGraph,
     MatchingStrategy matchingStrategy, ProgressFunction
     progressFunction, TGraphProperty... tGraphProperties) throws
     DiffServiceException, ServiceAccessException
2 public static Delta diff(Graph leftTGraph, Graph rightTGraph,
     MatchingStrategy matchingStrategy, TGraphProperty...
     tGraphProperties) throws DiffServiceException,
     ServiceAccessException
3 public static Delta diff(Graph leftTGraph, Graph rightTGraph,
     ProgressFunction progressFunction, TGraphProperty...
     tGraphProperties) throws DiffServiceException,
     ServiceAccessException
4 public static Delta diff(Graph leftTGraph, Graph rightTGraph,
     TGraphProperty... tGraphProperties) throws
     DiffServiceException, ServiceAccessException
```

Listing 1-5: Signaturen: Berechnung des TGraph-Delta

Als Vorbedingung für die Berechnung des TGraph-Delta ist festgelegt, dass beide TGraphen zu dem gleichen TGraph-Schema konform sein müssen. Wird diese Bedingung verletzt, so wird eine `DiffServiceException` ausgelöst. Wird diese Vorbedingung erfüllt, so wird als Ergebnis ein `Delta` zurückgegeben. Dieses kann zur weiteren Verarbeitung verwendet werden.

Die Klasse `DeltaIO` stellt Operationen bereit, die das TGraph-Delta in einer Datei oder einem beliebigen `DataOutputStream` schreiben. Die Generierung eines

TGraph-Delta und die Speicherung in einer Datei zeigt das Beispiel in dem folgenden Listing:

```
1 Graph tgl = GraphIO.loadSchemaAndGraphFromFile("./SodaMachine01.
     tg", CodeGeneratorConfiguration.MINIMAL,
     NullProgressFunctionImpl.getInstance());
2 Graph tgr = GraphIO.loadSchemaAndGraphFromFile("./SodaMachine02.
     tg", CodeGeneratorConfiguration.MINIMAL,
     NullProgressFunctionImpl.getInstance());
3 Delta delta = TGDiffUtils.diff(tgl, tgr);
4 DeltaIO.saveDeltaToFile("./soda0102.delta", delta);
```

Listing 1-6: Beispiel: Generierung eine TGraph-Delta und Speicherung in einer Datei.

6.1.3. Einen TGraphen mit einem TGraph-Delta patchen

Um einen `Graph` zu patchen kann ein Entwickler die Operationssignaturen der `TGDiffUtils` Klasse aus Listing 1-7 verwenden. Als Ergebnis der Operation wird jeweils der veränderte `Graph` zurückgegeben.

```
1 public static Graph patch(Graph graphToBePatched, Delta delta)
2 public static Graph patch(Graph graphToBePatched, String
     deltaFile) throws DeltaIOException
```

Listing 1-7: Signaturen: Patchen eines TGraphen

6.1.4. Zwei TGraphen miteinander vergleichen

Für den Vergleich von zwei TGraphen stehen die in Listing 1-8 aufgezählten Operationssignaturen der `TGDiffUtils` Klasse zur Verfügung. Ihre Verwendung ist beispielsweise in Listing 1-1 auf Seite 122 und Listing 1-3 auf Seite 124 zu sehen.

```
1 public static Difference compare(Graph leftTGraph, Graph
     rightTGraph, MatchingStrategy matchingStrategy,
     ProgressFunction progressFunction, TGraphProperty...
     tGraphProperties) throws ServiceAccessException
2 public static Difference compare(Graph leftTGraph, Graph
     rightTGraph, MatchingStrategy matchingStrategy,
     TGraphProperty... tGraphProperties) throws
     ServiceAccessException
3 public static Difference compare(Graph leftTGraph, Graph
     rightTGraph, ProgressFunction progressFunction,
     TGraphProperty... tGraphProperties) throws
     ServiceAccessException
4 public static Difference compare(Graph leftTGraph, Graph
     rightTGraph, TGraphProperty... tGraphProperties) throws
     ServiceAccessException
```

Listing 1-8: Signaturen: Vergleich von TGraphen

Das Ergebnis ist in jedem Fall eine `Difference` . Diese besteht aus mehreren Komponenten, die die Abweichungen beider TGraphen hinsichtlich einer bestimmten TGraph-Eigenschaft repräsentieren. Zu den einzelnen Teilkomponenten zählen:

- `AttributeDifference`
- `ElementDifference`
- `IncidenceDifference`
- `OrderDifference`
- `TypeDifference`

Ein `Difference` Objekt kann dazu genutzt werden, um auf seine Komponenten zuzugreifen und somit zwei TGraphen hinsichtlich bestimmter Abweichungen hin zu untersuchen. So liefert z.B. der Aufruf `getElementDifference().getUnmatchedVerticesInRightTGraph()` ein `Set` vom Typ `Vertex`, welches alle Knoten des rechten TGraphen enthält, zu denen keine übereinstimmenden Knoten im linken TGraphen gefunden wurden.

6.2. Die Benutzung der TGraph-Diffutils von der Kommandozeile

Die TGraph-Diffutils Werkzeug-Sammlung ist vergleichbar mit den GNU-Diffutils[1]. Diese Werkzeuge können von der Kommandozeile aus verwendet werden, um textuelle Artefakte zu verwalten. Die TGraph-Diffutils Werkzeuge orientieren sich an der Handhabung der GNU-Diffutils.

Die Verwendung der einzelnen Werkzeuge wird in den folgenden Abschnitten erläutert.

6.2.1. Die TGraph-Diffutils Werkzeug-Sammlung

Die TGraph-Diffutils werden in einem Archiv zusammengefasst. Es besteht aus

tgcompare.{sh,bat} einem Skript zum Vergleich von TGraphen,

[1] `http://www.gnu.org/software/diffutils/diffutils.html` Stand: 1. Juli 2010

tgdiff.{sh,bat} einem Skript zur Generierung von TGraph-Deltas,

tgpatch.{sh,bat} einem Skript zum Patchen von TGraphen,

tgraphdiffutils.ini einer Konfigurationsdatei und

tgraphdiffutils.jar einem JAR-Archiv, welches die TGraph-Diffutils enthält

Die Batch-Skripte sind für die Microsoft Windows Plattform ausgelegt. Die Shell-Skripte sind auf Unix-Systemen lauffähig. Die TGraph-Diffutils können Plattformunabhängig verwendet werden.

Die Skripte können von dem Anwender direkt verwendet werden, um von der Kommandozeile aus das gewünschte Programm zu starten. Alternativ hierzu kann das gewünschte Programm auch wie folgt gestartet werden:

```
1 java -cp jgralab.jar:tgraphdiffutils.jar  de.uni_koblenz.jgralab
     .utilities.tgraphdiffutils.TGDiff
```

Dieser Aufruf entspricht der Ausführung des `tgdiff` Skript. Der Vollständigkeit halber zeigen die folgenden beiden Auflistungen den Start des Programms zum Vergleich von TGraphen und zum Patchen von TGraphen.

```
1 java -cp jgralab.jar:tgraphdiffutils.jar  de.uni_koblenz.jgralab
     .utilities.tgraphdiffutils.TGCompare
```

```
1 java -cp jgralab.jar:tgraphdiffutils.jar de.uni_koblenz.jgralab.
     utilities.tgraphdiffutils.TGPatch
```

Systemvorraussetzungen

Um die TGraph-Diffutils verwenden zu können, muss eine Java Runtime Environment[2], mindestens in der Version 1.5, in der Umgebung installiert sein. Außerdem benötigt man eine JGraLab-Version[3]. Die Diffutils wurden auf Basis des JGraLab Dimetrodon Release entwickelt und getestet.

[2] `http://www.java.com/de/download/manual.jsp` Stand: 1. Juli 2010

[3] Die aktuelle Version von JGraLab findet man unter
`http://www.uni-koblenz.de/~ist/JGraLab_Download` Stand: 1. Juli 2010

Die Konfiguration der Werkzeuge

Durch die Konfigurationsdatei `tgraphdiffutils.ini` kann der Anwender verschiedene Einstellungen vornehmen. Der Pfad zu den JGralLab und TGraph-Diffutils Jar-Archiven kann über die Variable `tgdiff_classpath` festlegt werden. Zudem kann durch die Variable `tgdiff_maxmem` die maximale Größe des zu verwendenden Java Heap Space eingestellt werden.

Die Gemeinsamkeiten beim Aufruf der Werkzeuge

Da die verschiedenen Werkzeuge oftmals Argumente mit identischer oder ähnlicher Bedeutung haben, wurden diese auch mit gleichnamigen Kommandozeilen-Optionen realisiert.

In den Beispielen wird davon ausgegangen, dass sich die Werkzeuge im `PATH` befinden, und ohne relative oder absolute Pfadangabe aufgerufen werden können.

Mit der Option `-h`, bzw. `--help`, kann zu jedem Programm dessen Hilfe aufgerufen werden, wie sie in Anhang B zu sehen sind. Zu jeder Option findet man dort eine kurze Beschreibung und es wird explizit angegeben, welche Optionen obligatorisch und welche optional sind. Diese Hilfe wird ebenfalls angezeigt, wenn ein obligatorisches Argument nicht angegeben wurde.

Die Option `-v`, bzw. `--version`, zeigt zu einem Programm dessen Bezeichner und Version an.

Mit der Option `-l <dateiname>`, bzw. `--left <dateiname>`, wird der Speicherort des linken TGraphen angegeben. Je nachdem welches Programm verwendet wird, kann diesem TGraph eine unterschiedliche Bedeutung zugeschrieben werden. Im Falle von `tgpatch` ist dies derjenige TGraph, der gepatcht wird. Im Falle von `tgdiff`, ist dies der TGraph, der den Ausgangspunkt für die Berechnung des TGraph-Deltas darstellt.

Mit der Option `-r <dateiname>`, bzw. `--right <dateiname>`, wird der rechte TGraph angegeben. Diese Option wird nur von `tgcompare` und `tgdiff` verwendet. Sie gibt den Speicherort des rechten TGraphen an. Im Falle von `tgdiff` ist dieser TGraph das Ziel der Delta-Berechnung. Dies bedeutet, dass das generierte Delta den linken TGraph in den rechten TGraph überführt. Genauer gesagt kann der linke TGraph durch das berechnete Delta gepatcht werden und er wird durch das Patchen derart verändert, dass er unter bestimmten Aspekten als äquivalent zu dem rechten TGraphen betrachtet werden kann.

Jedes der drei Werkzeuge schreibt sein Ergebnis über die Standardausgabe auf die Konsole. Das Ergebnis kann von der Standardausgabe umgeleitet werden. Mit der Option `-f <dateiname>` ist es zudem möglich eine Datei für das Ergebnis anzugeben. Wird diese Option verwendet, so wird das Ergebnis nur in

die Datei geschrieben und nicht über die Standardausgabe auf der Konsole ausgegeben.

Alle drei Werkzeuge können mit der Option `-p`, bzw. `--progress` dazu veranlasst werden, den aktuellen Fortschritt anzuzeigen. Wird ein Programm mit dieser Option gestartet, so zeigt es auf der Konsole über die Fehlerausgabe den aktuellen Fortschritt der Berechnung an. Hierfür wird nicht die Standardausgabe verwendet, damit das Ergebnis über die Standardausgabe weiter verwendet werden kann. So zeigt beispielsweise der folgende Aufruf den Fortschritt an und leitet das erzeugte TGraph-Delta über die Standardausgabe an die Datei SodaMachine.tgdelta um.

```
1 tgdiff -l SodaMachine01.tg -r SodaMachine02.tg -p > SodaMachine.
     tgdelta
```

Listing 2-1: Beispiel: Anzeige des Fortschritts und Umleitung des TGraph-Delta

Das Verhalten der Werkzeuge

Das Verhalten der Werkzeuge `tgdiff` und `tgcompare` kann durch den Benutzer von der Kommandozeile aus beeinflusst werden.

Standardmäßig werden bei der Verwendung der Programme alle Eigenschaften von TGraphen für die Berechnung berücksichtigt, ihre Elemente, die Inzidenzen, die Attributierung, die Typisierung und die Anordnung. Es kann aber für einen Anwender sinnvoll sein, lediglich bestimmte Eigenschaften zu berücksichtigen.

So werden zum Beispiel durch den folgenden Aufruf lediglich die Elemente, die Inzidenzen und die Attributierung berücksichtigt, während die Typisierung und Anordnung nicht beachtet werden.

```
1 tgdiff -l SodaMachine01.tg -r SodaMachine02.tg -e -a -i
```

Listing 2-2: Beispiel: Berechnung eines TGraph-Delta unter Berücksichtigung der Elemente, der Attributierung und der Inzidenzen der beiden TGraphen

Äquivalent zu diesem Aufruf ist der folgende Aufruf:

```
1 tgdiff -l SodaMachine01.tg -r SodaMachine02.tg -eai
```

Durch die Einschränkung auf bestimmte Eigenschaften wird ein kleineres Delta generiert und der Aufwand der Berechnung verringert. Dies kann für den Anwender sinnvoll sein, wenn bestimmte Eigenschaften, wie beispielsweise die Anordnung, für seinen Anwendungsfall nicht benötigt werden.

Ein weiteres Beispiel zeigt der Aufruf von `tgcompare` in dem folgenden Listing. Dieser Aufruf bewirkt, dass lediglich die Element-Differenz der beiden TGraphen berechnet wird. Ihre Anordnung, die Attributierung, Typisierung und die Inzidenzen werden in diesem Fall nicht berücksichtigt.

```
1 tgcompare -l CityGraph01.tg -r CityGraph02.tg -e
```

Listing 2-3: Beispiel: Vergleich zweier TGraphen in Bezug auf ihre Elemente

Das Ergebnis und die Laufzeit der Programme sind zu einem Großteil von dem verwendeten Verfahren zur Berechung des Mappings abhängig. Die Verfahren unterscheiden sich in der Genauigkeit des Ergebnisses und ihrer Laufzeit. Je nach Anwendungsszenario kann es sinnvoll sein, ein bestimmtes Verfahren zu verwenden. Die TGraph-Diffutils ermöglichen es, unterschiedliche Verfahren zu verwenden. Entwickler können die Diffutils um weitere Verfahren erweitern. Der Anwender selbst kann zwischen diesen Verfahren wählen, indem er mit der Option `-m <name>` den Namen des zu verwendenden Verfahrens angibt. Die wählbaren Verfahren werden in der Hilfe unter der Option `-m` aufgelistet. Das folgende Listing zeigt die Verwendung von `tgdiff` mit dem `SimpleFlowMatcher`.

```
1 tgdiff -l SodaMachine01.tg -r SodaMachine02.tg -m
     SimpleFlowMatcher
```

Listing 2-4: Beispiel:Auswahl des Verfahrens zur Berechnung des Mapping

Die Vorraussetzungen zur Verwendung der Werkzeuge

Die TGraph-Diffutils können nur TGraphen verarbeiten, die im TGraph2-Dateiformat vorliegen. Ist dies nicht der Fall, so kann mit Hilfe des Werkzeugs `TGraphToTGraph2Converter` der TGraph in das neue Format überführt werden. Dies gilt für alle Programme, beim Aufruf von der Kommandozeile. Zudem müssen bei jedem Programmaufruf die obligatorischen Argumente angegeben werden. Vorraussetzungen, die nur einzelne Programme betreffen, werden in den folgenden Abschnitten erläutert.

6.2.2. `tgdiff` - Ein TGraph-Delta erzeugen

Die Berechnung des TGraph-Delta erfolgt durch den **diff** Operator. Die TGraph-Diffutils bieten eine Implementierung des diff Operators an. Der Anwender kann diesen Operator von der Kommandozeile aus verwenden. Um das Programm aufzurufen, kann das bereits erwähnte `tgdiff` Skript genutzt werden.

Die Ausführung von `tgdiff` ist nur unter Angabe des linken und des rechten TGraphen möglich. Der Aufruf in dem folgenden Listing berechnet ein TGraph-Delta der beiden angegebenen TGraphen. Das Delta wird auf der Kommandozeile über die Standardausgabe ausgegeben.

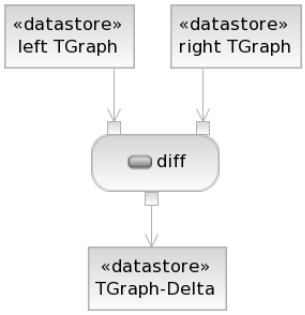


Abbildung 6.1.: Die Berechnung des TGraph-Delta durch den `diff` Operator

```
1 tgdiff.sh -l SodaMachine01.tg -r SodaMachine02.tg
```

Listing 2-5: Beispiel: Berechnung des TGraph-Delta

Es gibt eine spezielle Vorbedingung, für die Ausführung des `tgdiff` Programms. Beide TGraphen müssen zu dem gleichen TGraph-Schema konform sein.

Die Ausgabe des Programms

Die Ausgabe des Programms ist ein TGraph-Delta, welches dem in Anhang A.3 spezifizierten Format entspricht. Ein solches TGraph-Delta kann von den TGraph-Diffutils geladen und verwendet werden.

Ein Beispiel

In dem folgenden Beispiel wird mit `tgdiff` ein TGraph-Delta erzeugt. Die Anweisung in dem folgenden Listing wird auf die beiden TGraphen aus Abbildung 6.2 ausgeführt.

```
1 tgdiff -l CityGraph01.tg -r CityGraph02.tg
```

Das Ergebnis der Berechnung zeigt das folgende Listing.

```
1 // JGraLab - The Java graph laboratory
2 //   Version : Dimetrodon
3 //   Revision: 2610
4 //   Build ID: 68
5
6 // TGraphDiffutils
7 // Version: 1.0.1
8
9 Graph "CityGraph01" Version:13 CityGraph
```

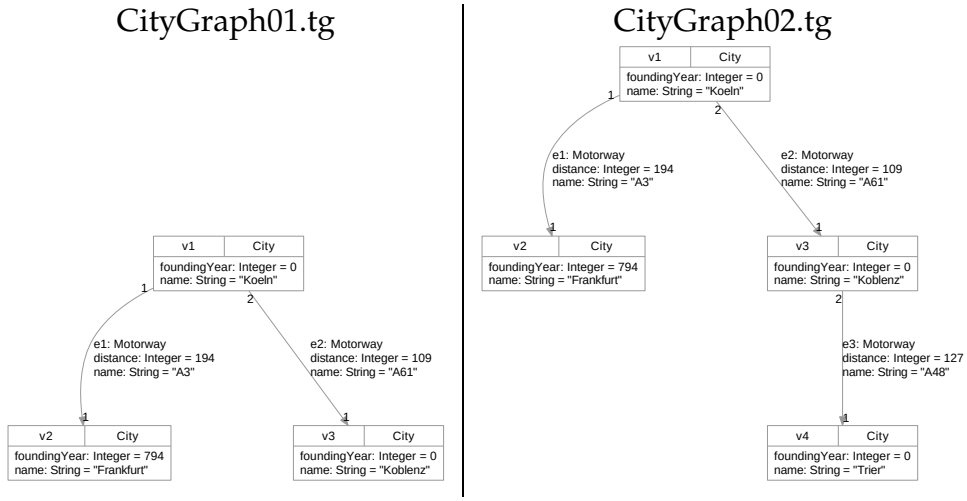


Abbildung 6.2.: Der linke und der rechte TGraph weisen kleine Abweichungen auf

```
10 AddVertex v0t structure.City ;
11 UpdateAttributeValue v0t foundingYear 0 ;
12 UpdateAttributeValue v0t name "Trier" ;
13 AddEdge e3t structure.Motorway from v3 to v0t ;
14 UpdateAttributeValue e3t distance 127 ;
15 UpdateAttributeValue e3t name "A48" ;
```

Listing 2-6: Das berechnete TGraph-Delta der beiden TGraphen aus Abbildung 6.2

In den ersten sieben Zeilen steht der Header, mit Informationen zu der verwendeten JGraLab Version und der für die Berechnung des Deltas verwendeten TGraph-Diffutils Version. In Zeile 9 stehen die obligatorischen Informationen zu dem linken TGraphen. Für diesen TGraphen wurde das Delta berechnet und für die Anwendung auf diesen TGraphen ist es vorgesehen. In den Zeilen 10 bis 15 stehen die Änderungsanweisungen. Sie werden in der Reihenfolge ihrer vorhergesehenen Ausführung aufgelistet.

Die Änderungsanweisungen werden derart im TGraph-Delta Dateiformat dargestellt, dass sie möglichst verständlich für einen Anwender sind. Die Anweisung in Zeile 10 deutet darauf hin, dass ein Knoten vom Typ `structure.City` mit der temporären ID `v0t` hinzugefügt werden soll. In den beiden darauf folgenden Zeilen werden dem neuen Knoten Attributwerte zugewiesen. In Zeile 13 steht eine Anweisung zum Hinzufügen einer Kante. Diese Kante soll vom Typ `structure.Motorway` sein und von dem Knoten `v3` zu dem neu erzeugten Knoten `v0t` zeigen. In den Zeilen 14 und 15 stehen Anweisungen zur Zuweisung von Attributwerten zu der neuen Kante.

6.2.3. `tgpatch` - Einen TGraphen mit einem TGraph-Delta patchen

Ein TGraph-Delta kann dazu verwendet werden, um einen TGraphen zu Patchen. Einen TGraphen zu Patchen bedeutet die schrittweise Durchführung der Änderungsanweisungen eines gegebenen Deltas auf den TGraphen. Der Anwender kann hierfür das Skript `tgpatch` verwenden.

Die Verwendung des Programms

Um einen TGraphen mit `tgpatch` zu patchen, müssen zumindest der TGraph mit der Option `-l <filename>` und das TGraph-Delta mit der Option `-d <filename>` angegeben werden. Diese beiden Angaben sind obligatorisch. Das TGraph-Delta muss sich in dem in Anhang A.3 auf Seite 169 spezifizierten Format befinden.

Um z.B. den linken TGraphen aus Abbildung 6.2 auf der vorherigen Seite mit dem Delta aus Listing 2-6 zu patchen genügt folgender Aufruf:

```
1 tgpatch.sh -l CityGraph01.tg -d cityGraph.tgdelta
```

Listing 2-7: Beispiel: Patchen eines TGraphen

Hierbei wurde das Delta zuvor in der Datei `cityGraph.tgdelta` gespeichert.

Die Ausgabe des Programms

Die Ausgabe des Programms ist ein veränderter TGraph. Er wird auf der Standardausgabe ausgegeben und kann beispielsweise in eine Datei umgeleitet werden, falls die Option `-f <filename>` nicht verwendet wird.

So erzeugt beispielsweise der folgende Aufruf einen gepatchten TGraphen und speichert diesen in der Datei `CityGraph01.patched.tg` .

```
1 tgpatch.sh -l CityGraph01.tg -d cityGraph.tgdelta -f CityGraph01
     .patched.tg
```

Listing 2-8: Patchen eines TGraphen und Speicherung des Ergebnis in einer Datei

6.2.4. `tgcompare` - Zwei TGraphen miteinander vergleichen

Ein weiterer Anwendungsfall der TGraph-Diffutils ist der Vergleich von TGraphen.

Um TGraphen miteinander zu vergleichen kann der Anwender das Programm `tgcompare` verwenden.

Für den Vergleich zweier TGraphen genügt der folgende Aufruf.

```
1 tgcompare.sh -l CityGraph01.tg -r CityGraph02.tg
```

Listing 2-9: Beispiel: Vergleich zweier TGraphen

Das Programm vergleicht beide TGraphen, berechnet ihre Differenz und gibt die textuelle Repräsentation der TGraph-Differenz auf der Konsole über die Standardausgabe aus.

Die beiden TGraphen müssen nicht zu dem gleichen Schema konform sein. Im Gegensatz zu `tgdiff`, kann `tgcompare` auch "Schema-fremde" TGraphen miteinander vergleichen.

Die Ausgabe des Programms

Die Ausgabe des Programms ist eine textuelle Repräsentation der TGraph-Differenz. Sie wurde entworfen, um dem Anwender einen schnellen, verständlichen Überblick über die Abweichungen zweier TGraphen zu ermöglichen. Die Spezifikation zu der textuellen Repräsentation der TGraph-Differenz befindet sich in Anhang A.4.

Das folgende Listing zeigt die Ausgabe von `tgcompare`. Dabei wurden die TGraphen aus dem Beispiel in Abschnitt 6.2.2 als Eingabe verwendet. In diesem Fall gibt es zwischen beiden TGraphen lediglich zwei Abweichungen bezüglich ihrer Elemente. Der Knoten `v4: structure.City` und die Kante `e3: structure.Motorway` befinden sich nur in dem rechten TGraphen. Vergleicht man die TGraph-Differenz mit dem TGraph-Delta aus Listing 2-6 auf Seite 132, so stellt man fest, dass beide sehr ähnliche Informationen beinhalten. Das TGraph-Delta ist jedoch gerichtet und muss zusätzliche Informationen über die Elemente beinhalten, damit diese erzeugt werden können.

```
1 //ElementDifference:
2 //Unmatched Vertices of the right TGraph :
3 v4: structure.City
4 //Unmatched Edges of the right TGraph :
5 +e3: structure.Motorway
```

Listing 2-10: Die Ausgabe von `tgcompare`, anhand des Beispiels

KAPITEL 7

EVALUATION

7.1. Erfüllung der Anforderungen

7.1.1. Anforderungen an Produktfunktionen

Im Folgenden wird die Erfüllung der Anforderungen an die Funktionen des Systems analysiert.

Anforderung 1: Delta berechnen Der Fokus dieser Diplomarbeit lag auf der Entwicklung eines Werkzeugs zur Berechnung eines Deltas zu zwei TGraphen. Diese in Anforderung 1 auf Seite 76 erfasste Aufgabe wird durch die TGraph-Diffutils erfüllt.

Anforderung 2: Delta exportieren Im Verlaufe dieser Diplomarbeit wurde ein Format zur Repräsentation von Deltas entworfen. Dessen Spezifikation wurde in Anhang A.3 auf Seite 169 dokumentiert. Der Entwurf des TGraph-Delta Formats wurde in Abschnitt 4.4 beschrieben.
Die API der TGraph-Diffutils und das Programm `TGDiff` ermöglichen den Export eines berechneten Deltas in dem spezifizierten TGraph-Delta Format. Zudem enthalten die Diffutils das `tgdiff`-Skript, welches den Aufruf des Programms von der Kommandozeile aus vereinfacht. Diese Anforderung wird somit erfüllt.

Anforderung 3: TGraph patchen Die TGraph-Diffutils ermöglichen das Patchen von TGraphen. Hierzu wurde der **Patch Service** entwickelt. Dessen Entwurf wurde in Abschnitt 4.3 festgehalten. Die TGraph-Diffutils bieten auch hier die Möglichkeit an, die Funktion durch Verwendung der API zu nutzen, oder durch das Programm `TGPatch`. Die Anforderung wird somit erfüllt.

Anforderung 4: TGraphen vergleichen Die Berechnung der Differenz durch den Vergleich zweier TGraphen stellt eine zentrale Anforderung dar. Die Differenz wird für die Berechnung des Deltas benötigt.
Der in Abschnitt 4.3 entworfene **Compare Service** und dessen Implementierung ermöglicht die Berechnung der Differenz. Ein Entwickler kann über die API diese Dienstleistung verwenden. Außerdem kann das Programm `TGCompare` verwendet werden. Diese Anforderung wird erfüllt.

Anforderungen 5 und 6: Two-Way Merge und Three-Way Merge Die Zusammenführung zweier TGraphen durch die Umsetzung des in Abschnitt 2.7.2 definierten **Two-Way Merge** wurde nicht implementiert. Diese Anforderung wurde als Wunsch klassifiziert. In Abschnitt 4.3 wurde der **TGraph-Merge Service** entworfen. Auch enthalten die TGraph-Diffutils das entsprechende `MergeService` Interface. Eine Implementierung des Service konnte in dieser Diplomarbeit aus zeitlichen Gründen nicht realisiert werden. Gleiches gilt für den **Three-Way Merge**.

7.1.2. Anforderungen an die Daten

Die Erfüllung der Anforderungen an die Daten ist von großer Bedeutung für die weitere Entwicklung der TGraph-Diffutils. Zusätzliche Werkzeuge, die in folgenden Arbeiten den TGraph-Diffutils hinzugefügt werden, können auf den entwickelten Datenstrukturen aufbauen.

Repräsentation des TGraph-Delta

Anforderung 7: Standard Format Das TGraph-Delta Format wurde im Verlauf der Diplomarbeit entworfen und formal spezifiziert, siehe Anhang A.3 auf Seite 169. Die TGraph-Diffutils verwenden dieses Format zur Delta-Repräsentation. Die Anforderung wird somit erfüllt.

Anforderung 8: Verständlichkeit Der Entwurf des TGraph-Delta Formats orientiert sich an der Spezifiktion des TGraph2 Formats. Für Anwender, die mit dem TGraph2 Format vertraut sind, ist das TGraph-Delta Format leicht verständlich.

Anforderung 9: Speicherplatz-Effizienz Diese Anforderung wird nicht erfüllt. Es wurde zwar bei dem Entwurf der TGraph-Delta Format darauf geachtet, dass ein Delta nur die notwendigsten Informationen enthält, dennoch kann die Speicherplatz-Ausnutzung optimiert werden.

Anforderung 10: Schema-Unabhängigkeit Die Repräsentation eines Deltas ist unabhängig von dem TGraph-Schema der verglichenen TGraphen. Diese Anforderung wird erfüllt.

Anforderung 11: TGraph-Unabhängigkeit Ein TGraph-Delta kann nicht unabhängig von dessen Entstehungs-Kontext auf beliebige TGraphen angewandt werden. Diese Anforderung, die als Vorschlag eingebracht wurde, wurde nicht umgesetzt.

Repräsentation der TGraph-Differenz

Anforderung 13: Verständlichkeit Bei dem in Abschnitt 4.5 entworfenen und in Anhang A.4 spezifizierten Format lag das Hauptaugenmerk auf der Verständlichkeit.

7.1.3. Anforderungen an die Anpassbarkeit und Erweiterbarkeit

Anforderung 14: Einbringung neuer Mapping-Verfahren Die TGraph-Diffutils können durch zusätzliche Verfahren zur Berechnung des Mappings erweitert werden. In Abschnitt 5.1.4 wird dieser Erweiterungspunkt beschrieben. Diese Anforderung wird erfüllt.

Anforderung 15: Einbringung neuer Änderungsanweisungen Die TGraph-Diffutils beinhalten einen grundlegenden Operationssatz für die Erzeugung von Deltas. Dieser Operationssatz kann von Entwicklern, wie in Abschnitt 5.3.4 beschrieben, erweitert werden. Diese Erweiterungsmöglichkeit erlaubt es Entwicklern beliebig komplexe Änderungsanweisungen einzubringen und dadurch das TGraph-Delta den eigenen Bedürfnissen anzupassen.

Anforderung 16: TGraph-Äquivalenz für die Delta-Berechnung Diese Anforderung wird von den TGraph-Diffutils umgesetzt. Sowohl die API, als auch die Kommandozeilen-Werkzeuge ermöglichen es einem Anwender das Level der TGraph-Äquivalenz selbst zu bestimmen. Dadurch kann ein Anwender die Performanz und den Umfang des Ergebnisses mit bestimmen und seinen Anforderungen entsprechend anpassen.
Diese Anforderung wird somit erfüllt.

7.1.4. Anforderungen an die Kompatibilität und Interoperabilität

Anforderung 17: Integration in VCS Aus zeitlichen Gründen konnte eine Integration in ein VCS nicht realisiert werden. Diese Anforderung wird nicht erfüllt.

Anforderung 18: Integration in JGraLab Diese Anforderung wird erfüllt.

7.1.5. Anforderungen an die Benutzerschnittstelle

Anforderung 21: Aufruf der Werkzeuge Jedes im Verlauf der Diplomarbeit entwickelte Werkzeug kann über ein Programm aufgerufen werden. Die TGraph-Diffutils[1] enthalten die folgenden Programme:

TGCompare. Ein Programm zum Vergleich von TGraphen. Es berechnet zu zwei gegebenen TGraphen die Differenz.

TGDiff. Ein Programm zur Berechnung des TGraph-Deltas.

TGPatch. Ein Programm zum Patchen von TGraphen.

Die Verwendung dieser Programme wurde in Abschnitt 6.2 ausführlich beschrieben.

Anforderung 22: Bereitstellung einer API Ein Entwickler kann über eine API die Produktfunktionen der TGraph-Diffutils in eigenen Anwendungen verwenden. Die Verwendung der API wurde in Abschnitt 6.1 erläutert. Ein tieferes Verständnis der internen Abläufe liefert das Kapitel 5, welches die Implementierung der TGraph-Diffutils betrachtet.

Anforderung 23: Anzeige des Fortschritts Die Anzeige des Fortschritts ist bei Verwendung der API und beim direkten Aufruf der Werkzeuge möglich. Die API ermöglicht es zudem Entwicklern eigene Implementierungen der `ProgressFunction` und der `StatusFunction` zur Anzeige des Fortschritts einzubringen. Die TGraph-Diffutils können somit auch in dieser Hinsicht den Anforderungen und Wünschen von Anwendern angepasst werden.

Anforderung 24: Visualisierung der Differenz von TGraphen In dieser Diplomarbeit konnte aus zeitlichen Gründen die Visualisierung der Differenz nicht

[1] In der Version 1.0.4

berücksichtigt werden. Die Visualisierung stellt ein wichtiges Problem bei der weiteren Entwicklung der TGraph-Diffutils dar.

7.2. Das Laufzeitverhalten der TGraph-Diffutils

Keine der Anforderungen bezog sich auf das Laufzeitverhalten der TGraph-Diffutils. Diese wichtige qualitative Eigenschaft wird in diesem Abschnitt getrennt betrachtet.

Das **Eclipse Test And Performance Tools Platform Project** stellt Werkzeuge zum Tracing und Profiling von Java Anwendungen zur Verfügung. Diese Werkzeuge wurden für die Untersuchung der Ausführungszeit und des Laufzeitverhaltens der TGraph-Diffutils eingesetzt.

Mit Hilfe der Werkzeuge können Statistiken zur Ausführungszeit erstellt werden. Diese Statistiken enthalten für jeden aufgeführten Baustein die folgenden Werte:

Base Time. Die Ausführungszeit des Bausteins in Sekunden. Die Ausführungszeit anderer Bausteine, die durch den Baustein aufgerufen wurden, werden nicht berücksichtigt.

Average base time. Die durchschnittliche Ausführungszeit eines Methodenaufrufs in Sekunden.

Cumulative base time. Die kumulierte Ausführungszeit des Bausteins in Sekunden. Die Ausführungszeit anderer Bausteine, die durch den Baustein aufgerufen wurden, werden mit berücksichtigt.

Calls. Die Anzahl der Methodenaufrufe.

Die TPTP ermöglichen es unterschiedliche Granularitätsstufen der Bausteine zu betrachten. Es ist möglich die Statistiken auf der Granularitätsstufe von Paketen, von Klassen oder von Methoden zu berechnen.

Tabelle 7.1 auf Seite 143 zeigt die Statistik der Ausführungszeit zu dem Aufruf des `TGCompare`-Programms. Dem Programm wurden als Argumente die TGraphen des Fallbeispiels übergeben. Die Statistik wurde nach der Spalte BASE TIME sortiert. Die Statistik basiert auf der Granularitätsstufe von Paketen.

Da es sich um kleine TGraphen mit jeweils 18 Kanten und 12, bzw. 11, Knoten handelt, fällt der Anteil der Ausführungszeit, die für das Laden der TGraphen und deren Schemata benötigt wird im Vergleich zu der Ausführungszeit der TGraph-Diffutils relativ hoch aus. Aus diesem Grund weisen 4 Pakete, die JGraLab zuzuordnen sind, die höchste Ausführungszeit auf. Betrachtet man jedoch die kumulierte Ausführungszeit der Pakete `de.uni_koblenz.jgralab`

und de.uni_koblenz.jgralab.utilities.tgraphdiffutils, so zeigt sich, dass die gesamte Ausführungszeit der TGraph-Diffutils die Ausführungszeit von JGraLab übersteigt.

Unter den Paketen, die zu den TGraph-Diffutils zählen, benötigt das Paket de.uni_koblenz.jgralab.utilities.tgraphdiffutils.core.mapping die meiste Ausführungszeit. Dieses Paket enthält die Klassen und Methoden zur Berechnung des TGraph-Mapping.

Wenn man die gleichen Messwerte auf der Granularitätsebene von Klassen untersucht und die 10 Klassen der TGraph-Diffutils mit der höchsten Ausführungszeit aus der Statistik filtert, so erhält man die in Tabelle 7.2 abgebildete Statistik.

Diese Statistik ermöglicht es die Ausführungszeit der TGraph-Diffutils detaillierter zu betrachten und liefert Hinweise darauf, welche Klassen im besonderen Ausmaß die Ausführungszeit der TGraph-Diffutils beeinflussen.

Wie die Statistik belegt, benötigt die TypeAttrMatchingStrategy die meiste Ausführungszeit. Diese Klasse erscheint mehrmals in der Statistik, da sie mehrere Threads verwendet, um die Berechnung des Mappings zu beschleunigen.

Ebenfalls auffällig ist die hohe Ausführungszeit des ServiceRepository. Um diese genauer zu betrachten wurden die Messwerte des Profiling auf der Granularitätsebene von Methoden untersucht. Die Statistik wird in Tabelle 7.3 dargestellt. Um die Übersichtlichkeit zu erhöhen wurden in der Tabelle die Argumente und Rückgabewerte der Methoden und einige Spalten herausgefiltert.

Wie die Messerte der Tabelle zeigen, benötigen die Methodenaufrufe der TypeAttrMatchingStrategy zur Berechnung des Mappings die meiste Ausführungszeit.

Der Grund für die hohe Ausführungszeit des ServiceRepository liegt in dem Aufruf der Methode registerMatchingStrategiesInPackage und des Konstruktors. Dies ist auf den hohen Aufwand zur Initialisierung der Services und insbesondere der automatischen Registrierung der MatchingStrategies zurückzuführen.

Da die Größe der Eingabedaten, die Anzahl der Elemente der zu verarbeitenden TGraphen, keinerlei Einfluss auf die Ausführungszeit des ServiceRepository hat, sind Optimierungen an diesem Baustein der TGraph-Diffutils nicht notwendig.

7.2.1. Die Untersuchung des Laufzeitverhaltens

Um das Laufzeitverhalten der TGraph-Diffutils in Abhängigkeit von der Größe der Eingabedaten zu untersuchen, wurden unterschiedlich große TGraphen als Eingabe für das TGCompare-Programm verwendet und die Laufzeit gemessen.

Package	Base Time (seconds)	Average Base Time (seconds)	Cumulative Time (seconds)	Calls
de.uni_koblenz.jgralab.schema.impl	7,799602	0,000142	11,569590	54804
de.uni_koblenz.jgralab.codegenerator	3,735920	0,000080	5,624583	46655
de.uni_koblenz.jgralab	3,694873	0,000082	15,620468	44993
de.uni_koblenz.jgralab.impl	0,863237	0,000164	0,976969	5266
de.uni_koblenz.jgralab.utilities.tgraphdiffutils.core.mapping	0,586276	0,001112	1,207688	527
de.uni_koblenz.jgralab.utilities.tgraphdiffutils	0,211092	0,010555	16,885732	20
de.uni_koblenz.jgralab.utilities.tgraphdiffutils.core.difference	0,124315	0,002537	0,826105	49
de.uni_koblenz.jgralab.utilities.tgraphdiffutils.service	0,118197	0,010745	0,123203	11
de.uni_koblenz.ist.utilities.option_handler	0,106520	0,002803	0,106520	38
de.uni_koblenz.jgralab.impl.std	0,076972	0,000029	0,155349	2612

Tabelle 7.1.: Auszug aus der Statistik zur Ausführungszeit des `TGCompare`-Programms

Für die Generierung der Testdaten wurde ein Programm entwickelt, welches einen beliebig großen TGraphen zu einem vorgegebenen Schema erstellen kann.

CLASS	BASE TIME (SECONDS)	AVERAGE BASE TIME (SECONDS)	CUMULATIVE TIME (SECONDS)	CALLS
TypeAttrMatchingStrategy	0,270337	0,003180	0,391217	85
ServiceRepository	0,118197	0,010745	0,123203	11
TypeAttrMatchingStrategy$6	0,098628	0,049314	0,123352	2
TypeAttrMatchingStrategy$7	0,091753	0,045877	0,133662	2
Difference	0,047938	0,003424	0,541019	14
Mapping	0,039120	0,000094	0,392911	417
TypeAttrMatchingStrategy$3	0,034902	0,017451	0,111731	2
TypeAttrMatchingStrategy$4	0,031258	0,015629	0,102659	2
CompareServiceImpl	0,030472	0,007618	0,810575	4
AttributeDifference	0,016222	0,004056	0,064340	4

Tabelle 7.2.: Auszug aus der Statistik zur Ausführungszeit der einzelnen Klassen beim Aufruf des **TGCompare**-Programms

METHOD	CLASS	BASE TIME (SECONDS)
match	TypeAttrMatchingStrategy	0,250350
run()	TypeAttrMatchingStrategy$6	0,098529
run()	TypeAttrMatchingStrategy$7	0,091031
registerMatchingStrategiesInPackage	ServiceRepository	0,068862
ServiceRepository	ServiceRepository	0,045515
run()	TypeAttrMatchingStrategy$3	0,034799
run()	TypeAttrMatchingStrategy$4	0,031157
compare	CompareServiceImpl	0,030224
getMatchedEdgesInLeftGraph	Mapping	0,019575
computeAttributeDifference	Difference	0,012272

Tabelle 7.3.: Auszug aus der Statistik zur Ausführungszeit der einzelnen Methoden beim Aufruf des **TGCompare**-Programms

Bei der Generierung der Knoten und Kanten werden die Typen und Attributwerte durch Zufallswerte bestimmt. Ebenso werden die Inzidenzen zufällig festgelegt. Dadurch soll sichergestellt werden, dass der generierte linke TGraph und der rechte TGraph möglichst keine identischen Elemente beinhalten. Die Ausführungszeit der `TypeAttrMatchingStrategy` verkürzt sich, je mehr identische Elemente in den zu vergleichenden TGraphen enthalten sind. Beinhalten zwei TGraphen keine identischen Elemente, so entspricht dies dem Worst case für die Ausführungszeit dieser `MatchingStrategy`.

Abbildung 7.1 zeigt das Laufzeitverhalten des `TGCompare`-Programms unter Verwendung der `TypeAttrMatchingStrategy`. Wie man an dem abgebildeten Graphen erkennen kann, steigt die Laufzeit des Programms in Abhängigkeit von der Größe der Eingabe quadratisch an.

Wie stark das Laufzeitverhalten der TGraph-Diffutils von der `MatchingStrategy` abhängt, wird durch die Abbildung 7.2 auf Seite 146 deut-

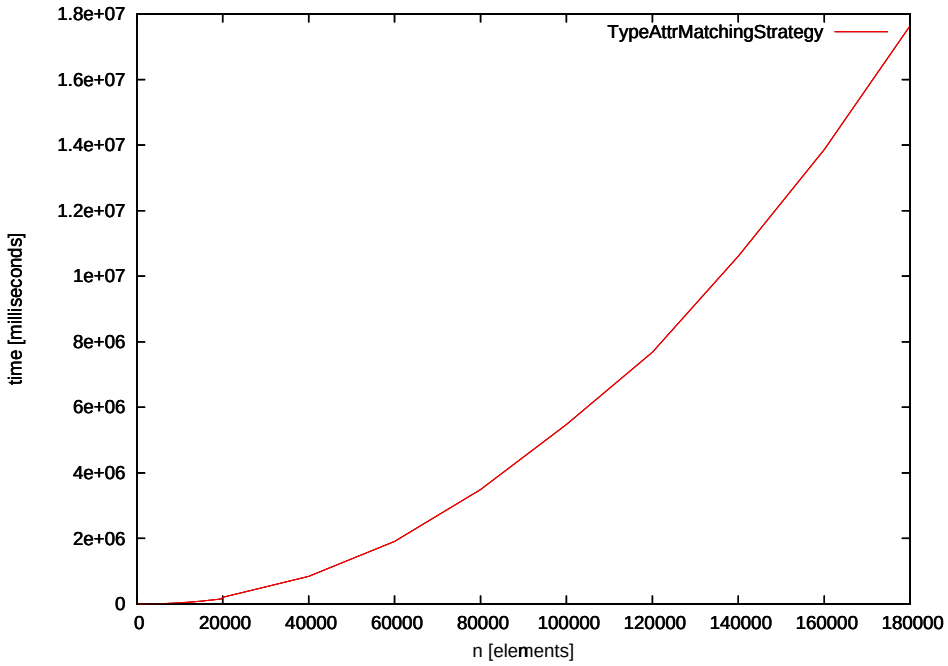


Abbildung 7.1.: Das Laufzeitverhalten des `TGCompare`-Programms unter Verwendung der `TypeAttrMatchingStrategy`

lich. Diese zeigt das Laufzeitverhalten von `TGCompare` unter der Verwendung verschiedener `MatchingStrategies`. Der rapide Anstieg der Ausführungszeit unter Verwendung des `SimpleFlowMatcher` im Vergleich zu den anderen beiden `MatchingStrategies` wird durch die Abbildung deutlich.

In Abbildung 7.3 wird durch die Angabe einer unteren und oberen Grenze verdeutlicht, dass die Laufzeit des `SimpleFlowMatcher` quadratisch in Abhängigkeit von der Größe der Eingabe ist. Der Anstieg der Ausführungszeit ist im Vergleich zur `TypeAttrMatchingStrategy` wesentlich höher.

Eine Anmerkung zum Laufzeitverhalten

Die Untersuchung des Laufzeitverhaltens der Programme liefert wichtige Erkenntnisse zum Verhalten der unterschiedlichen `MatchingStrategies` im Worst Case. Dieser Fall tritt in der Regel bei der Verwendung der TGraph-Diffutils nicht auf. In den meisten Anwendungsfällen werden die TGraph-Diffutils für den Vergleich zweier Versionen des gleichen TGraphen eingesetzt. Die Berechnung der Differenz zu zwei völlig verschiedenen TGraphen ist hingegen als ein untypischer Anwendungsfall einzustufen.

Wie praktische Tests der TGraph-Diffutils unter Verwendung der `TypeAttrMatchingStrategy` zeigen können Heuristiken, wie in Abschnitt 5.1.3 beschrieben, die Ausführungszeit der Berechnungen deutlich verkürzen.

Die Berechnung der Differenz zu zwei minimal veränderten Versionen eines TGraphen, mit jeweils über 1000000 Elemente, ist in weniger als 2 Minuten ab-

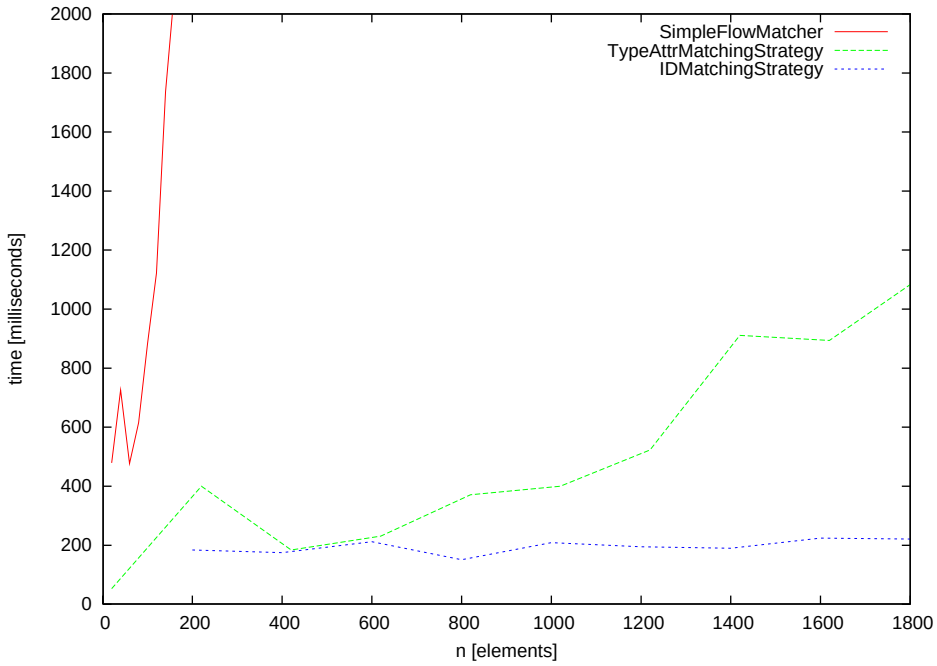


Abbildung 7.2.: Vergleich des Laufzeitverhaltens des `TGCompare`-Programms unter Verwendung unterschiedlicher `MatchingStrategies`

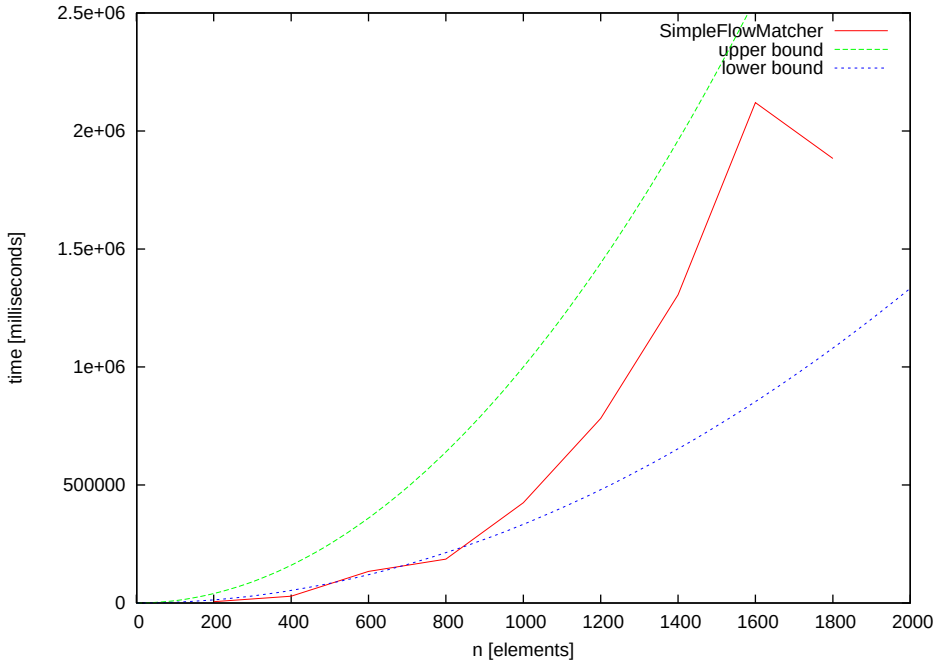


Abbildung 7.3.: Das Laufzeitverhaltens des `TGCompare`-Programms unter Verwendung des `SimpleFlowMatcher`

geschlossen. Bei einem anderen Test wurden zwei TGraphen unter Verwendung des `JavaScanner` erzeugt. Diese Anwendung kann TGraphen erstellen, die ein Java Programm repräsentieren. Für den Test wurde mit dem `JavaScanner` ein TGraph zu dem JGraLab Release Anatotitan und ein TGraph zu dem JGraLab Release Dimetrodon erstellt. Diese beiden TGraphen umfassen 4204 und 11288

Elemente. Das Programm `TGDiff` benötigt für die Berechnung des Deltas 1 Minute und 42 Sekunden.

KAPITEL 8

FAZIT UND AUSBLICK

Im Verlauf dieser Diplomarbeit wurde die Terminologie der TGraphen erweitert. Die zentralen Begriffe des Mappings, der Differenz und des Deltas wurden klar definiert.

Die im weiteren Verlauf der Arbeit entwickelten TGraph-Diffutils ermöglichen den Vergleich von TGraph-basierten Modellen. Diese Werkzeugsammlung ermöglicht den Vergleich von TGraphen, die Generierung von TGraph-Deltas und das Patchen von TGraphen.

Die Zusammenführung von TGraphen wurde aus zeitlichen Gründen im Verlauf der Arbeit nicht implementiert. Diese Funktionalität kann in einer folgenden Arbeit ohne Probleme in die TGraph-Diffutils integriert werden. Hierzu muss ein Entwickler das `MergeService` Interface implementieren, die Schnittstelle der `TGDiffUtils` um entsprechende Operationen erweitern und das `TGMerge`-Programm implementieren. Außerdem ist die Erstellung eines Skripts, welches den Aufruf des Programms von der Kommandozeile vereinfacht, empfehlenswert.

Die von den TGraph-Diffutils bereitgestellte API, ermöglicht eine Verwendung der TGraph-Diffutils in anderen Anwendungen. Über die **Facade** `TGDiffUtils` kann ein Entwickler auf die Funktionen der TGraph-Diffutils zugreifen. Dadurch werden Klienten von den TGraph-Diffutils abgekapselt. Veränderungen und Weiterentwicklungen haben somit keinen Einfluss auf Klienten. Ein Entwickler muss sich zudem nicht mit den Details der Implementierung befassen, um auf die Funktionen der TGraph-Diffutils zuzugreifen.

Die Architektur der TGraph-Diffutils ermöglichen einen unkomplizierten Austausch der Services. Eine Optimierung oder Anpassung des `DiffService` ist bspw. problemlos realisierbar.

Im Verlauf dieser Arbeit wurden drei unterschiedliche `MatchingStrategies` entwickelt. Diese wurden ad hoc entwickelt und weisen dementsprechend einige Schwachstellen auf. Keines der implementierten Verfahren wurde formal spezifiziert.

Wie die Untersuchung der Laufzeit gezeigt hat, ist das Laufzeitverhalten der TGraph-Diffutils zu einem großen Teil von dem Laufzeitverhalten der verwendeten `MatchingStrategy` abhängig. Diese beeinflusst zudem das Ergebnis der Differenz- und Delta-Berechnung. Die TGraph-Diffutils können ohne großen Aufwand um neue `MatchingStrategies` erweitert werden.

Eine mögliche Aufgabenstellung für eine auf den TGraph-Diffutils aufbauende Arbeit ist die Entwicklung einer `MatchingStrategy`. Diese sollte ein vertretbares Laufzeitverhalten aufweisen und exakte Ergebnisse liefern. Die Entwicklung eines formal spezifizierten Kostenmodells könnte dabei als Grundlage dienen.

Eine weitere interessante Problemstellung ist die Visualisierung der Differenz. Aufbauend auf den TGraph-Diffutils könnte ein Verfahren zur Visualisierung der Differenz entwickelt werden. Dieses sollte einem Anwender einen schnellen Überblick über die Abweichungen zweier TGraphen ermöglichen. Ein zu lösendes Problem ist dabei, dass die visuelle Darstellung von TGraphen mit zunehmender Größe zunehmend unübersichtlich wird. Ein Ansatz für die Visualisierung der Differenz könnte die Verwendung von **Polymetric Views** sein, wie es in [Wen08] erörtert wird.

ANHANG A

ENTWURFSDOKUMENTE

A.1. Die Spezifikation der Bausteine

Die in Kapitel 4 entworfenen Bausteine werden in dem nun folgenden Abschnitt genauer spezifiziert. Die Spezifikation ermöglicht es die Anforderungen an einen Baustein zu präzisieren[Ebe06]. Sie ist der Ausgangspunkt für die spätere Implementierung der Bausteine. Aus der Spezifikation können zudem Testfälle abgeleitet werden.

Für die Spezifikation der Bausteine wird ObjectZ verwendet. Diese Form der Spezifikation erlaubt es die gültigen Zustände eines Bausteins durch Invarianten zu spezifizieren. Es ermöglicht auch Vor- und Nachbedingungen für Operationen explizit und formal anzugeben. Durch die `visibility list` der ObjectZ Spezifikation wird die Schnittstelle eines Bausteins explizit angegeben.

A.1.1. Der Mapping-Baustein

Dem Entwurf entsprechend wurde die Spezifikation des Mapping-Bausteins aus der Definition 10 auf Seite 34 hergeleitet. Der Baustein wird durch die Klasse `Mapping` repräsentiert. Abbildung A.1 zeigt die ObjectZ Spezifikation des Bausteins.

Die Schnittstelle umfasst die Zustandsvariablen `mV` und `mE`, sowie die Operationen `addVertexMatch`, `addEdgeMatch`, `getVertexMatch`, und `getEdgeMatch`. Der Zustand eines `Mappings` wird durch die partiellen Injektionen `mV` und `mE`, sowie die TGraphen `leftTGraph` und `rightTGraph` bestimmt. Diese befinden sich in der visibility list und sind somit durch andere Bausteine zugreifbar. Jedoch können diese Attribute nicht von anderen Objekten direkt verändert werden. Die Semantik von ObjectZ erlaubt es zwar, dass Attribute referenzierter Objekte in der Nachbedingung einer Operation angegeben werden. Jedoch wird durch eine Referenzierung immer auf den Vorzustand des Attributes zugegriffen, niemals auf dessen Nachzustand.[Smi00]

Eine Veränderung der Attribute mV und mE ist nur durch Operationen des Mappings möglich, die explizit, durch Angabe in der Δ-Liste, eine Veränderung dieser Attribute bewirken. Die einzigen Operationen die mV und mE verändern, sind die addVertexMatch und addEdgeMatch Operationen. Dementsprechend wurden für diese Operationen Vorbedingungen und Nachbedingungen spezifiziert, so dass ein Mapping sich immer in einem konsistenten Zustand befindet.

A.1.2. Der Differenz-Baustein

Dem Entwurf in Abbildung 4.3 entsprechend und der Definition 16 auf Seite 53 folgend, wird der Differenz-Baustein als Aggregate behandelt. Seine AggregateRoot Difference kapselt die Objekte, aus denen sich die Differenz zusammensetzt. Abbildung A.2 zeigt die ObjectZ Spezifikation des Bausteins.

Die Element-Differenz

Ein Difference-Objekt referenziert ein Objekt der Klasse ElementDifference . Abbildung A.3 zeigt die ObjectZ Spezifikation dieses Bausteins. Die Klasse ElementDifference repräsentiert die aus Definition 11 auf Seite 42 bekannte Element-Differenz. In ihrer Schnittstelle wurde die Operation compute angegeben. Diese Operation ist jedoch nur für ein Objekt der AggregateRoot Difference sichtbar. Ein Difference-Objekt kontrolliert die Berechnung der Element-Differenz durch dessen computeElementDifference Operation. Nur durch den Aufruf dieser Operation wird die Element-Differenz berechnet. Das entsprechende ElementDifference-Objekt kann nach Ausführung seiner compute Operation nicht mehr verändert werden. Dieser Zustand wird durch das Attribut computed signalisiert. In diesem Zustand wird auch verhindert, dass ein Klient versehentlich die Berechnung der Element-Differenz mehrmals anstößt.

Die Inzidenz-Differenz

Die in Definition 12 auf Seite 44 formulierte Inzidenz-Differenz wird durch die Klasse IncidenceDifference spezifiziert. Abbildung A.4 zeigt die ObjectZ Spezifikation dieses Bausteins. Dem Entwurf entsprechend enthält die AggregateRoot Difference eine Referenz auf ein IncidenceDifference-Objekt.

Analog zur Klasse ElementDifference enthält diese Klasse eine compute Operation, die zur Berechnung der Inzidenz-Differenz vom AggregateRoot angestoßen werden kann. Ebenfalls wird durch das Attribut computed der Zustand eines IncidenceDifference-Objekts festgelegt und eine mehrfache Ausführung der compute Operation verhindert. Die compute Operation kann ausschließlich

Mapping

$\lceil(addVertexMatch, addEdgeMatch, getVertexMatch, getEdgeMatch, mV, mE)$

$leftTGraph, rightTGraph : TGraph$
$mV : V_L \rightarrowtail V_R$
$mE : E_L \rightarrowtail E_R$

INIT

$\forall x \in Vertex : mV(x) = \perp$
$\forall y \in Edge : mE(y) = \perp$

addVertexMatch

$\Delta(mV)$
$leftVertex?, rightVertex? : Vertex$

$leftTGraph \neq \perp \land rightTGraph \neq \perp$
$mV(leftVertex?) = \perp$
$leftVertex? \in V_L \land rightVertex? \in V_R$
$rightVertex? \notin \text{ran}(mV)$
$\forall v \in (V_L \setminus leftVertex) : mV'(v) = mV(v)$
$m'_V(leftVertex?) = rightVertex?$

addEdgeMatch

$\Delta(mE)$
$e1?, e2? : Edge$

$leftTGraph \neq \perp \land rightTGraph \neq \perp$
$mE(e1?) = \perp$
$e1? \in E_L \land e2? \in E_R$
$e2? \notin \text{ran}(mE)$
$\forall e \in (E_L \setminus e1) : mE'(e) = mE(e)$
$mE'(e1?) = e2?$

getVertexMatch

$vertex? : Vertex$
$matchingVertex! : Vertex$

$mV(vertex?) \neq \perp$
$mV(vertex?) = matchingVertex!$

getEdgeMatch

$e1? : Edge$
$e2! : Edge$

$mE(e1?) \neq \perp$
$mE(e1?) = e2!$

Abbildung A.1.: Spezifikation des Mapping Bausteins

Difference

$\lceil(\text{INIT}, eDiff, iDiff, tDiff, aDiff, oDiff)$

$eDiff : ElementDifference$
$iDiff : IncidenceDifference$
$tDiff : TypeDifference$
$aDiff : AttributeDifference$
$oDiff : OrderDifference$
$m : Mapping$
$leftTGraph, rightTGraph : TGraph$

INIT

$eDiff.\text{INIT} \land iDiff.\text{INIT} \land tDiff.\text{INIT} \land aDiff.\text{INIT} \land oDiff.\text{INIT}$

$computeElementDifference \mathrel{\widehat=}$
$eDiff.compute[m/m?, leftTGraph/leftTGraph?, rightTGraph/rightTGraph?]$
$computeIncidenceDifference \mathrel{\widehat=}$
$iDiff.compute[m/m?, leftTGraph/leftTGraph?, rightTGraph/rightTGraph?]$
$computeTypeDifference \mathrel{\widehat=}$
$tDiff.compute[m/m?, leftTGraph/leftTGraph?, rightTGraph/rightTGraph?]$
$computeAttributeDifference \mathrel{\widehat=}$
$aDiff.compute[m/m?, leftTGraph/leftTGraph?, rightTGraph/rightTGraph?]$
$computeOrderDifference \mathrel{\widehat=}$
$oDiff.compute[m/m?, leftTGraph/leftTGraph?, rightTGraph/rightTGraph?]$

Abbildung A.2.: Spezifikation der Klasse `Difference`

ElementDifference

$\restriction(compute, V_{LU}, V_{RU}, E_{LU}, E_{RU})$

$V_{LU} \subseteq Vertex$
$V_{RU} \subseteq Vertex$
$E_{LU} \subseteq Edge$
$E_{RU} \subseteq Edge$
$computed : \mathbb{B}$

$V_{LU} \cap V_{RU} = \varnothing$
$E_{LU} \cap E_{RU} = \varnothing$

INIT

$V_{LU} = \varnothing$
$V_{RU} = \varnothing$
$E_{LU} = \varnothing$
$E_{RU} = \varnothing$
$computed = false$

compute

$\Delta(V_{LU}, V_{RU}, E_{LU}, E_{RU}, computed)$
$m? : Mapping$
$leftTGraph?, rightTGraph? : TGraph$

$computed = false$
$\forall v \in V_L \mid m.getVertexMatch(v) = \bullet\!\perp\!\bullet\ v \in V'_{LU}$
$\forall v \in V_R \mid m.getVertexMatch(v) = \bullet\!\perp\!\bullet\ v \in V'_{RU}$
$\forall e \in E_L \mid m.getEdgeMatch(e) = \bullet\!\perp\!\bullet\ e \in E'_{LU}$
$\forall e \in E_R \mid m.getEdgeMatch(e) = \bullet\!\perp\!\bullet\ e \in E'_{RU}$
$computed' = true$

Abbildung A.3.: Spezifikation der Klasse `ElementDifference`

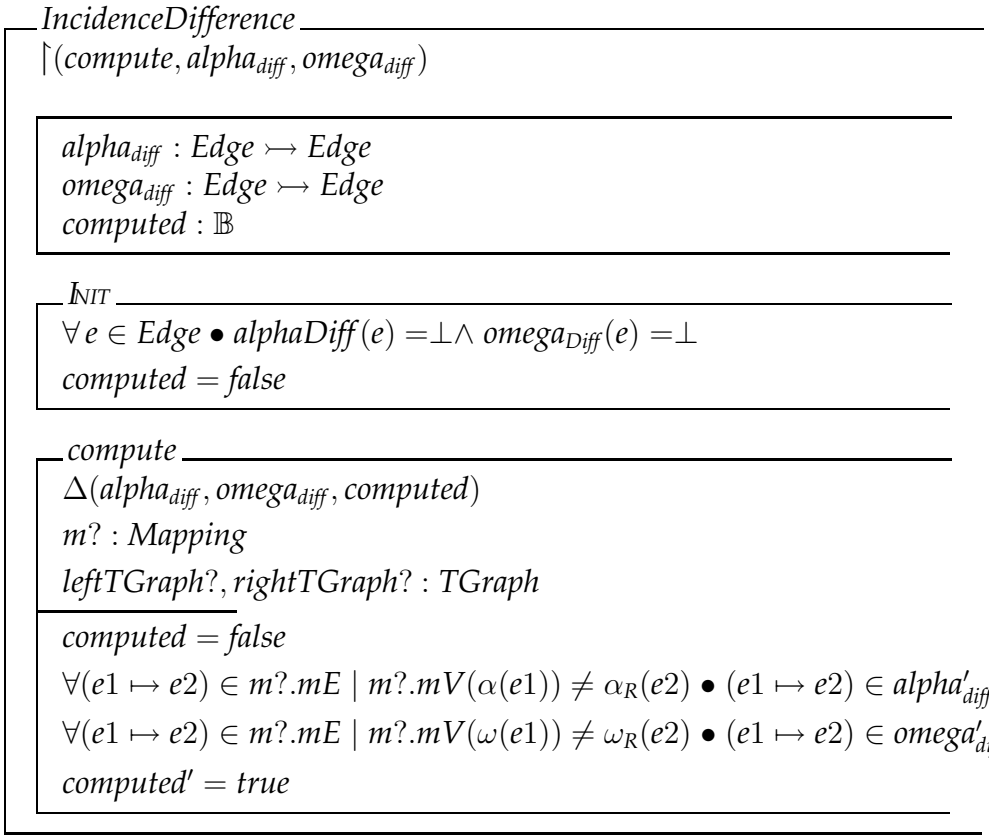


Abbildung A.4.: Spezifikation der Klasse `IncidenceDifference`

durch die `computeIncidenceDifference` Operation eines `Difference`-Objekt aufgerufen werden.

Die Typ-Difference

Analog zu den beiden vorangegangenen Spezifikationen wird auch die in Definition 13 auf Seite 46 formulierte Typ-Differenz durch eine entsprechende `TypeDifference` Klasse spezifiziert. Abbildung A.5 zeigt die ObjectZ Spezifikation dieses Bausteins. Auch die `compute` Operation eines `TypeDifference`-Objekts kann ausschließlich durch die `computeTypeDifference` Operation des entsprechenden `Difference`-Objekts aufgerufen werden.

Die in der Definition angegebene Abbildung $type_{diff}$, die Abweichungen der Typisierung bzgl. der Knoten und Kanten erfasst, wurde in der Spezifikation in zwei Abbildungen unterteilt. Die Abbildung $type_{VDiff} : Vertex \rightarrowtail Vertex$ erfasst die Abweichungen der Knoten. Die Abbildung $type_{EDiff} : Edge \rightarrowtail Edge$ erfasst die Abweichungen der Kanten. Diese Unterteilung erleichtert die Spezifikation und die spätere Implementierung.

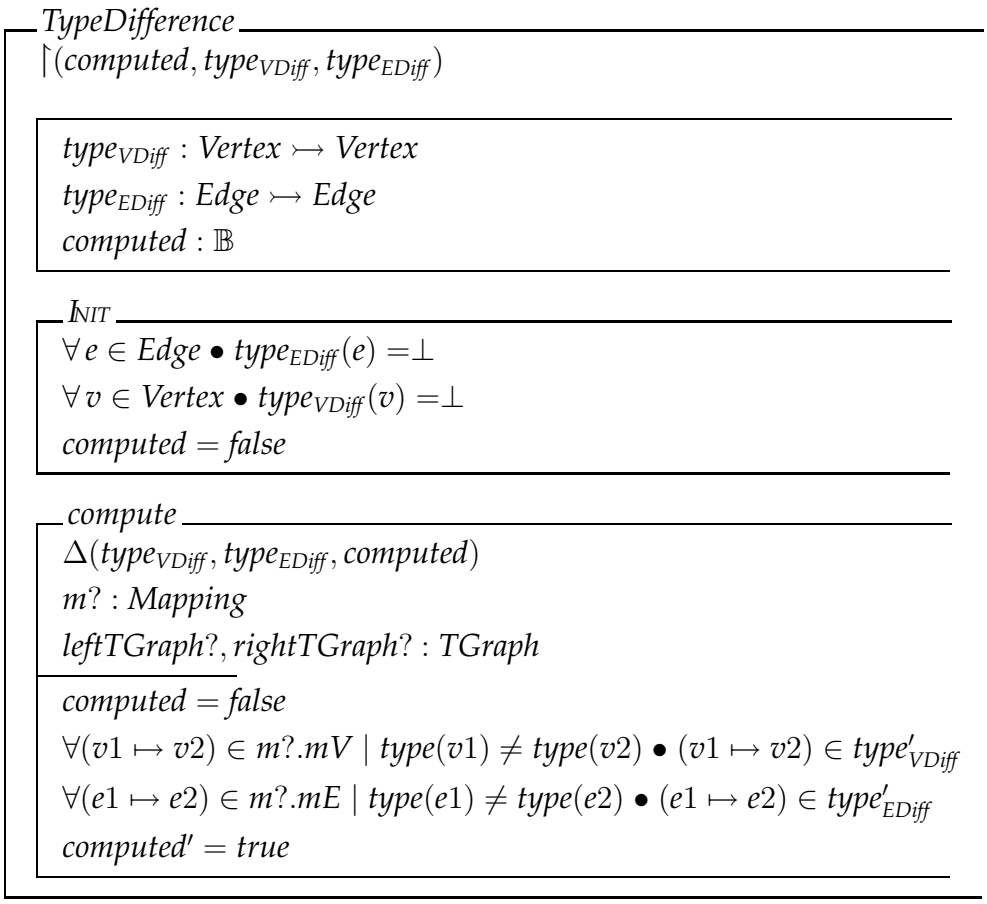


Abbildung A.5.: Die Spezifikation der Klasse `TypeDifference`

Die Attribut-Differenz

Analog erfolgt die Spezifikation der Attribut-Differenz aus Definition 14 auf Seite 48, durch eine entsprechende `AttributeDifference` Klasse. Ihre Spezifikation zeigt Abbildung A.6. Auch die `compute` Operation eines `AttributeDifference`-Objekt, kann ausschließlich durch die `computeAttributeDifference` Operation seines AggregateRoot-Objekt aufgerufen werden.

Auch hier wurden die Abbildungen $value_{diff}, attrL_{diff}, attrR_{diff}$, aus denen die Attribut-Differenz besteht, entsprechend in Abbildungen für Knoten und Kanten aufgeteilt. So werden die Abweichungen der Attributwerte, die durch die Abbildung $value_{diff}$ erfasst werden, in der Spezifikation durch die Abbildung $value_{VDiff}$ für Knoten und $value_{EDiff}$ für Kanten erfasst.

Die Anordnungs-Differenz

Entsprechend der Definition 15 auf Seite 51 der Anordnungs-Differenz, wurde die Klasse `OrderDifference` spezifiziert . Die Spezifikation zeigt Abbildung A.7. Die `compute` Operation dieser Klasse ist für die Berechnung der Anordnungs-Differenz verantwortlich. Ein Objekt der Klasse `OrderDifference` wird durch ein `Difference` Objekt, die Aggregate Root, kontrolliert. Nur die Aggregate Root kann die Berechnung der Anordnungs-Differenz starten.

AttributeDifference

$\lceil(computed, value_{VDiff}, value_{EDiff}, attrL_{VDiff}, attrL_{EDiff}, attrR_{VDiff}, attrR_{EDiff})$

$value_{VDiff} : Vertex \times AttrId \times Value \rightarrowtail Vertex \times Value$
$value_{EDiff} : Edge \times AttrId \times Value \rightarrowtail Edge \times Value$
$attrL_{VDiff} : Vertex \rightarrowtail AttrId \times Value$
$attrL_{EDiff} : Edge \rightarrowtail AttrId \times Value$
$attrR_{VDiff} : Vertex \rightarrowtail AttrId \times Value$
$attrR_{EDiff} : Edge \rightarrowtail AttrId \times Value$
$computed : \mathbb{B}$

INIT

$\forall v \in Vertex \bullet value_{VDiff}(v) = \bot \land attrL_{VDiff}(v) = \bot \land attrR_{VDiff}(v) = \bot$
$\forall e \in Edge \bullet value_{EDiff}(e) = \bot \land attrL_{EDiff}(e) = \bot \land attrR_{EDiff}(v) = \bot$
$computed = false$

compute

$\Delta(value_{VDiff}, value_{EDiff}, computed)$
$m? : Mapping$
$leftTGraph?, rightTGraph? : TGraph$

$computed = false$
$\forall (v1 \mapsto v2) \in m?.mV, \forall a \in AttrId \mid value(v1)(a) \neq value(v2)(a) \bullet$
$(v1, a, value(v1, a) \mapsto (v2, value(v2)(a)) \in value'_{VDiff}$
$\forall (e1 \mapsto e2) \in m?.mE, \forall a \in AttrId \mid value(e1)(a) \neq value(e2)(a) \bullet$
$(e1, a, value(e1)(a)) \mapsto (e2, value(v2)(a)) \in value'_{EDiff}$
$\forall (v1 \mapsto v2) \in m?.mV, \forall a \in AttrId \mid a \in \text{ran}(value(v1)) \land a \notin \text{ran}(value(v2)) \bullet$
$(v1 \mapsto a, value(v1)(a)) \in attrL'_{VDiff}$
$\forall (v1 \mapsto v2) \in m?.mV, \forall a \in AttrId \mid a \notin \text{ran}(value(v1)) \land a \in \text{ran}(value(v2)) \bullet$
$(v2 \mapsto a, value(v2)(a)) \in attrR'_{VDiff}$
$\forall (e1 \mapsto e2) \in m?.mE, \forall a \in AttrId \mid a \in \text{ran}(value(e1)) \land a \notin \text{ran}(value(e2)) \bullet$
$(e1 \mapsto a, value(e1)(a)) \in attrL'_{EDiff}$
$\forall (e1 \mapsto e2) \in m?.mE, \forall a \in AttrId \mid a \notin \text{ran}(value(e1)) \land a \in \text{ran}(value(e2)) \bullet$
$(e1 \mapsto a, value(e1)(a)) \in attrR'_{EDiff}$
$computed' = true$

Abbildung A.6.: Spezifikation der Klasse `AttributeDifference`

OrderDifference

$\lceil(V_{SeqDiff}, E_{SeqDiff}, \Lambda_{SeqDiff}, computed)$

$V_{SeqDiff} : \text{iseq}\, Vertex \nrightarrow Vertex$
$E_{SeqDiff} : \text{iseq}\, Edge \nrightarrow Edge$
$\Lambda_{SeqDiff} : Vertex \times \text{iseq}(Edge \times \{in, out\}) \nrightarrow Vertex \times Edge \times \{in, out\}$
$computed : \mathbb{B}$

INIT

$computed = false$

compute

$\Delta(V_{SeqDiff}, E_{SeqDiff}, \Lambda_{SeqDiff}, computed)$
$m? : Mapping$
$leftTGraph?, rightTGraph? : TGraph$

$computed = false$
$V'_{SeqDiff} = \{\langle v_i, .., v_j\rangle \mapsto v_k \in V_{LSeq} \times V_R \mid$
$i, j, k \in \mathbb{N} \land mV(v_i) = v_{k+1} \land mV(v_{i-1}) \neq v_k\}$
$\forall\, v \in x = \langle v_i, \ldots, v_j\rangle \in \text{dom}(V'_{SeqDiff}) : mV(v) \notin \text{ran}(V'_{SeqDiff})$
$E_{SeqDiff} := \{\langle e_i, .., e_j\rangle \mapsto e_k \in E_{LSeq} \times E_R \mid$
$i, j, k \in \mathbb{N} \land mE(e_i) = e_{k+1} \land mE(e_{i-1}) \neq e_k\}$
$\forall\, e \in x = \langle e_i, \ldots, e_j\rangle \in \text{dom}(E_{SeqDiff}) : mE(e) \notin \text{ran}(E_{SeqDiff})$
$\Lambda_{SeqDiff} := \{v_1, \langle(e_i, d_i), .., (e_j, d_j)\rangle \mapsto v_2, e_k, d_k \in V_L \times \Lambda_{LSeq} \nrightarrow V_R \times E_R \times \{in, out\} \mid$
$i, j, k \in \mathbb{N} \land d \in \{in, out\} \land mV(v_1) = v_2 \land mE(e_i) = e_{l+1} \land mE(e_{i-1}) \neq e_l\}$
$computed' = true$

Abbildung A.7.: Die Spezifikation der Klasse `OrderDifference`

A.1.3. Der Delta-Baustein

Nach Definition 23 auf Seite 67 ist ein Delta eine Sequenz von Änderungsoperationsanweisungen. Eine Änderungsoperationsanweisung wird durch ein `ChangeCommand`-Objekt repräsentiert. Ein `Delta`-Objekt enthält somit eine Sequenz von `ChangeCommand`-Objekten. Da es sich bei dem Delta-Baustein um ein Aggregate handelt, wird der Zugriff auf die `ChangeCommand`-Objekte durch die AggregateRoot, das `Delta`-Objekt, gekapselt.

Der Zustand eines `Delta`-Objekt besteht zudem aus dem Attribut closed, das einen booleschen Wert annehmen kann. Während der Erzeugung eines Deltas ist das Objekt veränderbar, dies wird durch die Belegung des Attributes closed mit dem Wert false ausgedrückt. In diesem Zustand können dem Delta Änderungsoperationsanweisungen angehangen werden. Die Operation `addCommand` ermöglicht es ein `ChangeCommand`-Objekt an die `commandSequence` anzuhängen.

Wurde ein Delta vollständig erzeugt, so wird das Attribut closed mit dem Wert true belegt, indem die Operation `close` aufgerufen wird. In diesem Zustand kann das Delta nicht mehr verändert werden, es ist immutable. Es ist nun abgeschlossen und kann z.B. von einem Patch Service verwendet werden.

Delta

$\restriction(commandSequence, addCommand)$

$commandSequence : \operatorname{seq} ChangeCommand$

$closed : \mathbb{B}$

INIT

$commandSequence = \langle\rangle$

$close = false$

addCommand

$\Delta(commandSequence)$

$cmd? : ChangeCommand$

$closed = false$

$commandSequence' = commandSequence \frown cmd?$

close

$\Delta(closed)$

$closed' = true$

A.1.4. Die Spezifikation der Änderungsoperationsanweisungen

Die Spezifikation der `ChangeCommand`-Objekte zeigt das folgende Listing.

```
1  public interface Command {
2
3    public void addStateListener(CommandStateListener
        commandStateListener);
4
5    public boolean execute();
6
7    public int getLocalId();
8
9    public AttributedElement getResult();
10
11   public int loadCommand(Delta delta, Graph tgraph, String
        currentLine, HashMap<String, Integer> cmdRefMap) throws
        LoadCommandException;
12
13 }
```

Listing 1-1: Das Interface Command

Hier ist keine Spezifikation mittels ObjectZ notwendig. Die Semantik der grundlegenden TGraph-Änderungsoperationen wurden bereits in Abschnitt 2.6 spezifiziert. Nach dem Entwurf des Delta-Bausteins, siehe Abbildung 4.4 auf Seite 90, muss jede TGraph-Änderungsoperation das Interface `ChangeCommand` implementieren.

Eine Änderungsoperation, wie z.B. `AddVertex`, wird durch den Aufruf der `execute` Operation ausgeführt.

A.2. Die Anwendungsfälle

Anwendungsfall 1 *Delta berechnen*

Akteure: VCS, Entwickler

Beschreibung: Der Akteur möchte zu zwei gegebenen TGraphen *TG*1 und *TG*2 das TGraph-Delta berechnen.

Vorbedingungen: *TG*1 und *TG*2 sind zu dem gleichen TGraph-Schema konform.

Nachbedingungen (Erfolg) : Der Akteur erhält ein TGraph-Delta zu den angegebenen TGraphen *TG*1 und *TG*2. Mit Hilfe des berechneten Deltas ist es möglich den TGraphen *TG*1 zu patchen (siehe Anwendungsfall 3), so dass ein zu *TG*2 äquivalenter TGraph *TG*2′ entsteht.

Nachbedingungen (Fehlschlag) :

Ablauf:

1. Schema prüfen
2. include (Differenz berechnen)
3. Änderungsoperationen generieren und sequentiell im Delta anordnen
4. (Prüfung durchführen)
5. include (Delta exportieren)

Ausnahmen:

Anwendungsfall 2 *Delta exportieren*

Akteure:

Beschreibung: Dieser Anwendungsfall erzielt ein Ergebnis, welches in erster Linie von dem Anwendungsfall 1 verwendet wird. Es dient der Umwandlung und der Ausgabe eines gegebenen TGraph-Deltas in einem vorgegebenen Format.

Vorbedingungen:

Nachbedingungen (Erfolg) :

Nachbedingungen (Fehlschlag) :

Ablauf:

1. Format auswählen
2. TGraph-Delta in das Format umwandeln

Ausnahmen:

Anwendungsfall 3 *TGraphen patchen*

Akteure: Entwickler, VCS

Beschreibung: Der Akteur möchte das System verwenden, um einen TGraphen mit einem Delta zu patchen. Dazu wird ein Patch Prozess angestoßen, der die im Delta enthaltenen Änderungsoperationen an dem TGraphen ausführt.

Vorbedingungen: -

Nachbedingungen (Erfolg) : Der Akteur erhält den vollständig erzeugten TGraphen. Alle Änderungsoperationen wurden ausgeführt.

Nachbedingungen (Fehlschlag) : Der Akteur erhält den teilweise erzeugten TGraphen und das Fehlerprotokoll, welches alle nicht ausgeführten Änderungsoperationen enthält.

Ablauf:

1. Der Akteur gibt den TGraphen und das zu verarbeitende Delta an.
2. wiederhole (bis alle Änderungsoperationen durchgeführt wurden)
   a nächste Änderungsoperation aus dem Delta laden
   b Anwendbarkeit der Operation prüfen
   c Änderungsoperation ausführen, wenn diese angewendet werden kann
3. Ausgabe des erzeugten TGraphen

Ausnahmen: -

Anwendungsfall 4 *TGraphen vergleichen*

Akteure: Entwickler

Beschreibung: Dieser Anwendungsfall wird vom System bereitgestellt, um es einem Akteur zu ermöglichen zwei TGraphen miteinander zu vergleichen und das Ergebnis des Vergleichs weiter zu verarbeiten.

Vorbedingungen: -

Nachbedingungen (Erfolg) : Dem Akteur wird die berechnete Differenz in einem geeigneten Format zur Verfügung gestellt.

Nachbedingungen (Fehlschlag) : -

Ablauf:

1. include (Differenz berechnen)
2. include (Differenz exportieren)

Ausnahmen:

Anwendungsfall 5 *Differenz berechnen*

Akteure: -

Beschreibung: Die Berechnung der Differenz zweier TGraphen stellt ein wichtiges Zwischenergebnis zur Erfüllung vieler anderer Anwendungsfälle bereit. Akteure nutzen diesen Anwendungsfall nur indirekt, wenn ein anderer Anwendungsfall, insbesondere die Anwendungsfälle 1 und 4, genutzt wird

Vorbedingungen: Beide TGraphen sind zu dem gleichen Schema konform.

Nachbedingungen (Erfolg) : Die berechnete Differenz liegt im System vor und kann weiter verarbeitet werden.

Nachbedingungen (Fehlschlag) : -

Ablauf:

1. Schema prüfen
2. include (Mapping berechnen)
3. Differenz anhand des Mapping ermitteln:
   a Element-Differenz berechnen
   b Inzidenz-Differenz berechnen
   c Typ-Differenz berechnen
   d Attribut-Differenz berechnen
   e Anordnungs-Differenz berechnen

Ausnahmen:

Anwendungsfall 6 *Mapping berechnen*

Akteure: -

Beschreibung: Die Berechnung des Mapping erfolgt mit dem Ziel ein Mapping zweier vom Akteur angegebenen TGraphen zu ermitteln. Dieser Anwendungsfall und sein Ergebnis ist insbesondere für die Durchführung des Anwendungsfall 5 wichtig.

Vorbedingungen: -

Nachbedingungen (Erfolg) : Das System erzeugt ein Mapping der beiden TGraphen.

Nachbedingungen (Fehlschlag) : -

Ablauf:

1. Mapping-Verfahren auswählen
2. Mapping-Verfahren ausführen

Ausnahmen:

Anwendungsfall 7 *Two-Way Merge*

Akteure: VCS, Entwickler

Beschreibung: Der Akteur stößt diesen Anwendungsfall an, wenn er zwei TGraphen zu einem neuen TGraphen zusammenführen möchte.

Vorbedingungen: Die beiden zusammenzuführenden TGraphen sind zu dem gleichen Schema konform.

Nachbedingungen (Erfolg) : Das Ergebnis ist ein zusammengeführter TGraph, der aus Elementen beider TGraphen besteht.

Nachbedingungen (Fehlschlag) : -

Ablauf:

1. Schema prüfen
2. include (Differenz berechnen)
3. include (Konflikte auflösen)
4. TGraph erzeugen

Ausnahmen:

Anwendungsfall 8 *Three-Way Merge*

Akteure: VCS, Entwickler

Beschreibung: Der Akteur möchte zwei TGraphen zusammenführen, die aus einem gemeinsamen Basis-TGraphen entstanden sind.

Vorbedingungen: Alle TGraphen sind zu dem gleichen Schema konform.

Nachbedingungen (Erfolg) : Das Ergebnis ist ein neuer TGraph, der aus den Elementen beider TGraphen besteht.

Nachbedingungen (Fehlschlag) : -

Ablauf:

1. Schema prüfen
2. include (Differenz berechnen)
3. include (Konflikte auflösen)
4. TGraph erzeugen

Ausnahmen:

A.3. Das TGraph-Delta Format

```
1  TGDeltaFile ::= TGDeltaFileHeader OperationSequence ;
2
3  TGraphFileHeader ::= ....
4                       TGraphSchema
5                       TGraphHeader ";" ;
6
7  TGraphHeader ::= "TGraph" GraphId GraphVersion GraphClassName ;
8
9  OperationSequence ::= { OperationInstruction };
10
11 OperationInstruction ::= ( AddVertexDeclaration
12                           | DeleteVertexDeclaration
13                           | AddEdgeDeclaration
14                           | DeleteEdgeDeclaration
15                           | UpdateAlphaDeclaration
16                           | UpdateOmegaDeclaration
17                           | UpdateTypeDeclaration
18                           | UpdateAttributeValueDeclaration
19                           | PutVertexAfterDeclaration
20                           | PutVertexBeforeDeclaration
21                           | PutEdgeAfterDeclaration
22                           | PutEdgeBeforeDeclaration
23                           | PutIncidenceAfterDeclaration
```

```
                        | PutEdgeIncidenceBeforeDeclaration
                        )

AddVertexDeclaration ::= "AddVertex" tempVertexId
   VertexClassName ";" ;

DeleteVertexDeclaration ::= "DeleteVertex" VertexId ";" ;

AddEdgeDeclaration ::= "AddEdge" tempEdgeId EdgeClassName "from"
(VertexId|tempVertexId) "to" (VertexId|tempVertexId) ";" ;

DeleteEdgeDeclaration ::= "DeleteEdge" EdgeId ";" ;

UpdateAlphaDeclaration ::= "UpdateAlpha"
(EdgeId|tempEdgeId) (VertexId|tempVertexId) ";" ;

UpdateOmegaDeclaration ::= "UpdateOmega"
(EdgeId|tempEdgeId) (VertexId|tempVertexId) ";" ;

UpdateAttributeValueDeclaration ::= "UpdateAttributeValue"
GraphElementId AttrId Value ";" ;

UpdateTypeDeclaration ::= "UpdateType" GraphElementId
(VertexClassName|EdgeClassName) ";" ;

PutVertexAfterDeclaration ::= "PutVertexAfter"
(VertexId|tempVertexId) (VertexId|tempVertexId) ";" ;

PutVertexBeforeDeclaration ::= "PutVertexBefore"
(VertexId|tempVertexId) (VertexId|tempVertexId) ";" ;

PutEdgeAfterDeclaration ::= "PutEdgeAfter"
(EdgeId|tempEdgeId) (EdgeId|tempEdgeId) ";" ;

PutEdgeBeforeDeclaration ::= "PutEdgeBefore"
(EdgeId|tempEdgeId) (EdgeId|tempEdgeId) ";" ;

PutIncidenceAfterDeclaration ::= "PutIncidenceAfter"
(EdgeId|tempEdgeId) (EdgeId|tempEdgeId) ";" ;

VertexId ::= "v" IntegerValue ;

tempVertexId ::= "v" IntegerValue "t" ;

EdgeId ::= "e" IntegerValue ;

tempEdgeId ::= "e" IntegerValue "t" ;

GraphElementId ::= (VertexId|tempVertexId|EdgeId|tempEdgeId) ;

Attribute ::= AttributeName ":" Domain;

```

```
75 AttributeName ::= IdentifierString;
76
77 Value ::= BooleanValue
78           | IntegerValue
79           | LongValue
80           | DoubleValue
81           | StringValue
82           | ListValue
83           | SetValue
84           | MapValue
85           | RecordValue
86           | NullValue;
87
88 NullValue ::= "n";
89
90 BooleanValue ::= "f" | "t";
```

Listing 3-1: EBNF für TGraph-Delta Dateien

A.4. Das TGraph-Differenz Format

```
1  TGDifference ::= TGDifferenceHeader DifferenceSequence ;
2
3  TGraphFileHeader ::= JGraLabHeader
4                       TGDiffUtilsHeader
5                       TGraphHeaderLeft
6                       TGraphHeaderRight ";" ;
7
8  TGraphHeaderLeft ::= "left Graph" GraphId GraphVersion
      GraphClassName ;
9
10 TGraphHeaderRight ::= "right Graph" GraphId GraphVersion
      GraphClassName ;
11
12 DifferenceSequence ::= [ ElementDifference ] [
      AttributeDifference ] [ IncidenceDifference ] [
      TypeDifference ] [ OrderDifference ];
13
14 ElementDifference ::= ElementDifferenceHeader [
      UnmatchedVerticesLeft ] [ UnmatchedVerticesRight ] [
      UnmatchedEdgesLeft ] [ UnmatchedEdgesRight ];
15
16 ElementDifferenceHeader ::= "//ElementDifference:" ;
17
18 UnmatchedVerticesLeft ::= "//Unmatched vertices of the left
      TGraph:" { VertexId ":" VertexClassName } ;
19
20 UnmatchedVerticesRight ::= "//Unmatched vertices of the right
      TGraph:" { VertexId ":" VertexClassName } ;
21
```

```
22 UnmatchedEdgesLeft ::= "//Unmatched edges of the left TGraph:" {
       EdgeId ":" EdgeClassName } ;
23
24 UnmatchedEdgesRight ::= "//Unmatched edges of the right TGraph:"
       { EdgeId ":" EdgeClassName } ;
25
26 AttributeDifference ::= AttributeDifferenceHeader
     AttributeDifferenceTableHeading {UpdatedElementAttributes} ;
27
28 AttributeDifferenceHeader ::= "//AttributeDifference:" ;
29
30 AttributeDifferenceTableHeading ::= "left El. | right El. | Attr
     . | left Value | right Value" ;
31
32 UpdatedElementAttributes ::= (EdgeId | VertexId ) " | " (EdgeId
     | VertexId ) " | " AttributeName " | " Value " | " Value} ;
33
34 IncidenceDifference ::= IncidenceDifferenceHeader {
     AlphaVertexDeviations} {OmegaVertexDeviations} ;
35
36 IncidenceDifferenceHeader ::= "//IncidenceDifference:" ;
37
38 AlphaVertexDeviations := "//AlphaVertex deviations:"
     AlphaDifferenceTableHeading {IncidenceDeviations} ;
39
40 AlphaDifferenceTableHeading ::= "eL | a(eL) | m(a(eL) | eR | a(
     eR) | m(a(eR))" ;
41
42 IncidenceDeviations ::= EdgeId " | " VertexId " | " VertexId " |
       " EdgeId " | " VertexId " | " VertexId ;
43
44 OmegaVertexDeviations := "//OmegaVertex deviations:"
     OmegaDifferenceTableHeading {IncidenceDeviations} ;
45
46 OmegaDifferenceTableHeading ::= "eL | o(eL) | m(o(eL) | eR | o(
     eR) | m(o(eR))" ;
47
48 TypeDifference ::= TypeDifferenceHeader
     TypeDifferenceTableHeader VertexTypeDeviations
     EdgeTypeDeviations;
49
50 TypeDifferenceHeader ::= "//TypeDifference:" ;
51
52 TypeDifferenceTableHeader ::= "Left Graph | Right Graph" ;
53
54 VertexTypeDeviations ::= {VertexId ":" VertexClassName " | "
     VertexId ":" VertexClassName} ;
55
56 EdgeTypeDeviations ::= {EdgeId ":" EdgeClassName " | " EdgeId ":
     " EdgeClassName} ;
57
```

```
58 OrderDifference ::= OrderDifferenceHeader VertexOrderDeviations
      EdgeOrderDeviations ;
59
60 OrderDifferenceHeader ::= "//OrderDifference:" ;
61
62 VertexOrderDeviations ::= {VertexId "put after" VertexId} ;
63
64 EdgeOrderDeviations ::= {EdgeId "put after" EdgeId} ;
65
66 VertexId ::= "v" IntegerValue ;
67
68 EdgeId ::= "e" IntegerValue ;
69
70 GraphElementId ::= (VertexId|tempVertexId|EdgeId|tempEdgeId) ;
71
72 Vertex ::= VertexClassName AttributeValuePairs;
73
74 Edge ::= EdgeClassName AttributeValuePairs;
75
76 AttributeValuePairs ::= { Attribute "=" Value };
77
78 Attribute ::= AttributeName ":" Domain;
79
80 AttributeName ::= IdentifierString;
81 Value ::= BooleanValue
82           | IntegerValue
83           | LongValue
84           | DoubleValue
85           | StringValue
86           | ListValue
87           | SetValue
88           | MapValue
89           | RecordValue
90           | NullValue;
91
92 NullValue ::= "n";
93
94 BooleanValue ::= "f" | "t";
```

Listing 4-1: EBNF für die textuelle Repräsentation der TGraph-Differenz

ANHANG B

DIE AUSGABE DER HILFE ZU DEN PROGRAMMEN

```
usage: java de.uni_koblenz.jgralab.utilities.tgraphdiffutils.TGDiff [-h ]
            [-v ] -l <filename> -r <filename> [-f <filename>] [-p ] [-m
            <name>] [-e ] [-a ] [-i ] [-t ] [-o ]
 -a,--attributeDifference   (optional): If this option is set, the
                            attribute difference will be computed.
 -e,--elementDifference     (optional): If this option is set, the element
                            difference will be computed.
 -f,--outfile <filename>    (optional): Write the delta to the given
                            <filename>. If this option is not set, the
                            delta will be written so the console.
 -h,--help                  (optional): print this help message.
 -i,--incidenceDifference   (optional): If this option is set, the
                            incidence difference will be computed.
 -l,--left <filename>       (required): The tg file that contains the left
                            TGraph, that will be compared to the right
                            TGraph.
 -m <name>                  (optional): Choose one of the available
                            matching strategies.The following matching
                            strategies can be choosen :
                            SimpleTypeAttrMatching
                            SimpleFlowMatcher
                            IDMatching
 -o,--orderDifference       (optional): If this option is set, the order
                            difference will be computed.
 -p,--progress              (optional): If set, the tool will show the
                            progress on the console.
 -r,--right <filename>      (required): The tg file that contains the
                            right TGraph.
 -t,--typeDifference        (optional): If this option is set, the type
                            difference will be computed.
 -v,--version               (optional): print version information
```

Listing 0-1: Die Hilfe zum Starten von tgdiff

```
usage: java de.uni_koblenz.jgralab.utilities.tgraphdiffutils.TGPatch [-
    h ]
```

```
2            [-v ] -l <filename> -d <filename> [-f <filename>] [-p ]
3  -d,--delta <filename>     (required): The delta that will be used to
4                            patch the TGraph.
5  -f,--outfile <filename>   (optional): write the patched TGraph to the
6                            given <filename>
7  -h,--help                 (optional): print this help message.
8  -l,--left <filename>      (required): The tg file that will be patched
       .
9  -p,--progress             (optional): If set, the tool will show the
10                           progress of performing the patch on the
11                           console.
12 -v,--version              (optional): print version information
```

Listing 0-2: Die Hilfe zum Starten von tgpatch

```
1  usage: java de.uni_koblenz.jgralab.utilities.tgraphdiffutils.TGCompare
       [-h
2            ] [-v ] -l <filename> -r <filename> [-f <filename>] [-p ]
                [-e
3            ] [-a ] [-i ] [-t ] [-o ] [-m <name>] [-g <filename>]
4  -a,--attributeDifference  (optional): If this option is set, the
5                            attribute difference will be computed.
6  -e,--elementDifference    (optional): If this option is set, the
       element
7                            difference will be computed.
8  -f,--outfile <filename>   (optional): Write the difference to the
       given
9                            <filename>. If this option is not set, the
10                           difference will be written to the console.
11 -g,--dotFile <filename>   (optional): Prints a dot file to the given
12                           filename. This dot file should be used to
13                           create a postscript, which shows the
14                           difference using two layers.
15 -h,--help                 (optional): print this help message.
16 -i,--incidenceDifference  (optional): If this option is set, the
17                           incidence difference will be computed.
18 -l,--left <filename>      (required): The tg file that contains the
       left
19                           TGraph, that will be compared to the right
20                           TGraph.
21 -m <name>                 (optional): Choose one of the available
22                           matching strategies.The following matching
23                           strategies can be choosen :
24                           SimpleTypeAttrMatching
25                           SimpleFlowMatcher
26                           IDMatching
27 -o,--orderDifference      (optional): If this option is set, the
       order
28                           difference will be computed.
29 -p,--progress             (optional): If set, the tool will show the
30                           progress on the console.
31 -r,--right <filename>      (required): The tg file that contains the
32                           right TGraph.
33 -t,--typeDifference       (optional): If this option is set, the type
34                           difference will be computed.
35 -v,--version              (optional): print version information
```

Listing 0-3: Die Hilfe zum Starten von tgcompare

GLOSSAR

A

Artefakt Ein Artefakt ist ein Objekt (i.w.S.), das einen konkreten Sachverhalt beschreibt und maschinenlesbar ist.[Ebe06], S. 17.

C

Comparison and Versioning of Software Models (CVSM) , S. 13.

E

Eclipse Test And Performance Tools Platform Project (TPTP) , S. 141.

G

GNU is not Unix (GNU) Das GNU Projekt wurde 1984 ins Leben gerufen um ein komplett freies Betriebssystem, genannt GNU, zu entwickeln. `http://www.gnu.org/`, S. 74.

Graph Repository Query Language 2 (GReQL2) Eine Abfragesprache für TGraphen. Sie dient zur Extraktion von inhaltlichen, struk- turellen oder aggregierten Informationen aus TGraphen[Mar06]., S. 24.

Graph Repository Transformation Language (GReTL) Eine Transformationssprache, die den operationalen Ansatz zur Modelltransformation umsetzt[Wei09]. Sie ermöglicht es Transformationen von TGraphen auf einem höheren Abstraktionsniveau zu definieren., S. 24.

Graph Visualization Software (Graphviz) , S. 24.

graphUML (grUML) Ein Profil der UML2 Klassendiagramme, zur Modellierung von TGraph-Schemata mit Hilfe von UML2 Editoren [BHR+10]., S. 28.

I

International Conference on Software Engineering (ICSE) , S. 13.

J

Java Graph Laboratory (JGraLab) Es handelt sich hierbei um eine Java Klassenbibliothek, die eine hoch effiziente API zur Verarbeitung von TGraphen bereitstellt. JGraLab ermöglicht die Erzeugung, den Zugriff und die Manipulation von TGraphen zur Laufzeit., S. 3.

V

Version Eine Version ist die konkrete Ausprägung eines Artefakts., S. 18.

Version Control System (VCS) , S. 121.

LITERATURVERZEICHNIS

[BCE+06] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh, *A manifesto for model merging*, GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management (New York, NY, USA), ACM, 2006, pp. 5–12.

[BE08] Daniel Bildhauer and Jürgen Ebert, *Querying Software Abstraction Graphs*, Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008), collocated with ICPC 2008, 2008.

[BH05] Jean Bézivin and Reiko Heckel (eds.), *Language Engineering for Model-Driven Software Development, 29. February - 5. March 2004*, Dagstuhl Seminar Proceedings, vol. 04101, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[BHE09] Daniel Bildhauer, Tassilo Horn, and Jurgen Ebert, *Similarity-driven software reuse*, CVSM '09: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (Washington, DC, USA), IEEE Computer Society, 2009, pp. 31–36.

[BHR+10] Daniel Bildhauer, Tassilo Horn, Volker Riediger, Hannes Schwarz, and Sascha Strauß, *grUML - A UML based modelling language for TGraphs*, 2010, Version 0.0, Stand: 1. Juli 2010.

[Blo08] Joshua Bloch, *Effective java (2nd edition) (the java series)*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[CW98] Reidar Conradi and Bernhard Westfechtel, *Version models for software configuration management*, ACM Comput. Surv. **30** (1998), no. 2, 232–282.

[dAS09] Brian de Alwis and Jonathan Sillito, *Why are software projects moving from centralized to decentralized version control systems?*, CHASE '09: Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering (Washington, DC, USA), IEEE Computer Society, 2009, pp. 36–39.

[DBL09] *31st international conference on software engineering, icse 2009, may 16-24, 2009, vancouver, canada, companion volume*, IEEE, 2009.

[DW98] Peter Dahm and Friedbert Wiedmann, *Das Graphenlabor*, Koblenz-Landau, Univ., 1998.

[Ebe87] Jürgen Ebert, *A versatile data structure for edge-oriented graph algorithms*, Commun. ACM **30** (1987), no. 6, 513–519.

[Ebe06] Jürgen Ebert, *Mitschrift zur Vorlesung Softwaretechnik I, Wintersemester 2005/2006*, 2006, Vorlesungsunterlagen, Stand: 1. Juli 2010.

[Ebe07] ———, *Mitschrift zur Vorlesung Softwaretechnik II, Wintersemester 2006/2007*, 2007, Vorlesungsunterlagen, Stand: 1. Juli 2010.

[EBSR07] Jürgen Ebert, Daniel Bildhauer, Hannes Schwarz, and Volker Riediger, *Using Difference Information to Reuse Software Cases*, Softwaretechnik-Trends **27** (2007), no. 2.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of algebraic graph transformation (monographs in theoretical computer science. an eatcs series)*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EKS09] Jürgen Ebert, Udo Kelter, and Tarja Systä, *Workshop on comparison and versioning of software models (cvsm 2009)*, in *ICSE Companion* [DBL09], pp. 457–458.

[ERSB08] Jürgen Ebert, Volker Riediger, Hannes Schwarz, and Daniel Bildhauer, *Using the TGraph Approach for Model Fact Repositories*, Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2008), 2008, pp. 9–18.

[ERW08] Jürgen Ebert, Volker Riediger, and Andreas Winter, *Graph Technology in Reverse Engineering, The TGraph Approach*, 10th Workshop Software Reengineering (WSR 2008) (Bonn) (Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter, eds.), vol. 126, GI, 2008, pp. 67–81.

[Eva03] Evans, *Domain-driven design: Tacking complexity in the heart of software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FHL+98] Eckhard D. Falkenberg, Wolfgang Hesse, Paul Lindgreen, Björn E. Nilsson, J. L. Han Oei, Colette Rolland, Ronald K. Stamper, Frans J. M. Van Assche, Alexander A. Verrijn-Stuart, and Klaus Voss, *A FRAMEWORK OF INFORMATION SYSTEM CONCEPTS*, 1998.

[FW07] Sabrina Förtsch and Bernhard Westfechtel, *Differencing and Merging of Software Diagrams - State of the Art and Challenges*, ICSOFT (SE),

2007, pp. 90–99.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[HE10] Tassilo Horn and Jürgen Ebert, *The GReTL Transformation Language*, 2010, Stand: 1. Juli 2010.

[Her09] Markus Herrmannsdoerfer, *Operation-based versioning of metamodels with COPE*, CVSM '09: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (Washington, DC, USA), IEEE Computer Society, 2009, pp. 49–54.

[HM76] J. W. Hunt and M. D. McIlroy, *An algorithm for differential file comparison*, Tech. Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[HM08] Wolfgang Hesse and Heinrich C. Mayr, *Modellierung in der Softwaretechnik: eine Bestandsaufnahme*, Informatik Spektrum **31** (2008), no. 5, 377–393.

[Hor08] Tassilo Horn, *Ein Optimierer für GReQL2*, Koblenz-Landau, Univ., 2008.

[jav06] *Java Platform, Standard Edition 6 API Specification*, 2006, Sun Microsystems.

[KBS04] Dirk Krafzig, Karl Banke, and Dirk Slama, *Enterprise soa: Service-oriented architecture best practices (the coad series)*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[Kel07] Udo Kelter, *Begriffliche Grundlagen von Modelldifferenzen*, Softwaretechnik-Trends 27:2 (2007).

[KHS09] Maximilian Kögel, Jonas Helming, and Stephan Seyboth, *Operation-based conflict detection and resolution*, CVSM '09: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (Washington, DC, USA), IEEE Computer Society, 2009, pp. 43–48.

[Küh04] Thomas Kühne, *What is a model?*, in Bézivin and Heckel [BH05].

[Kö08] Maximilian Kögel, *Towards software configuration management for unified models*, CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models (New York, NY, USA), ACM, 2008, pp. 19–24.

[Mar06] Katrin Marchewka, *Entwurf und Definition der Graphanfragesprache GReQL 2*, Koblenz-Landau, Univ., 2006.

[MES02] David MacKenzie, Paul Eggert, and Richard Stallman, *Comparing and Merging Files*, 2002, for Diffutils 2.8.1 and patch 2.5.4, Stand: 1. Juli 2010.

[MGMR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm, *Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching*, 18th International Conference on Data Engineering (ICDE 2002), 2002.

[Ohs04] Dirk Ohst, *Versionierungskonzepte mit Unterstützung für Differenz- und Mischwerkzeuge*, Siegen, Univ., Diss., 2004.

[Roz97] Grzegorz Rozenberg (ed.), *Handbook of graph grammars and computing by graph transformation: volume i. foundations*, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[SERW08] Hannes Schwarz, Jürgen Ebert, Volker Riediger, and Andreas Winter, *Towards Querying of Traceability Information in the Context of Software Evolution*, 10th Workshop Software Reengineering (WSR 2008) (Bonn) (Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter, eds.), vol. 126, 2008, pp. 144–148.

[SM08] Martín Soto and Jürgen Münch, *Using model comparison to maintain model-to-standard compliance*, CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models (New York, NY, USA), ACM, 2008, pp. 35–40.

[Smi00] Graeme Smith, *The object-z specification language / by graeme smith*, Kluwer Academic, Boston :, 2000 (English).

[Sot07] Martin Soto, *Delta-P: Model comparison using semantic web standards*, Softwaretechnik-Trends 27 (2007), 72–73.

[Sta73] Herbert Stachowiak, *Allgemeine Modelltheorie*, Springer, Wien, 1973.

[Uhr08] Sabrina Uhrig, *Matching Class Diagrams: With Estimated Costs Towards the Exact Solution?*, CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models (New York, NY, USA), ACM, 2008, pp. 7–12.

[Wei09] Oliver Weichert, *GReTL- Entwurf und Implementierung eines operationalen Ansatzes für Modelltransformationen*, Koblenz-Landau, Univ., 2009.

[Wen08] Sven Wenzel, *Scalable visualization of model differences*, CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models (New York, NY, USA), ACM, 2008, pp. 41–46.