Studienarbeit

Vereinfachung und Automatisierung der Datenverwaltung für Semedico

Philipp Lucas

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik
28. Mai 2013

Inhaltsverzeichnis

1 Einleitung **4**

2 Problemstellung **6**

3 Grundlagen **8**
3.1 MeSH . . . 8
3.1.1 MeSH Record Types . . . 8
3.1.2 MeSH Komponenten und XML . . . 9
3.1.3 Jährliche Änderungen . . . 11
3.2 Graphentheorie . . . 12
3.3 Der MeSH als Graph . . . 14
3.3.1 Operationen und Transformationen . . . 15
3.3.2 Kommutativität von Kompositionen der Transformationen . . . 17
3.3.3 Relationen . . . 18
3.3.4 Ähnlichkeit . . . 19

4 Transformationsrekonstruktion bei MeSH-Graphen **20**
4.1 Spezifikation . . . 20
4.2 Motivation der Spezifikation . . . 20
4.3 Algorithmus . . . 21
4.4 Vorgehen für Transformationstypen . . . 21
4.4.1 Descriptor-Transformationen . . . 23
4.4.2 Vertex-Neuverknüpfung . . . 25
4.4.3 Weitere Vertex-Transformationen . . . 25
4.4.4 Tree-Vertex-Löschungen . . . 27
4.4.5 Descriptor-Löschungen . . . 27
4.5 Bewertung des Algorithmus . . . 28

5 Aktualisierung von Transformationsfolgen **29**
5.1 Problembeschreibung . . . 29
5.2 Grundidee . . . 30
5.3 Vorgehen für Aktualisierung einer Transformation . . . 31
5.3.1 Descriptor-Additionen . . . 32
5.3.2 Descriptor-Umbenennungen . . . 34
5.3.3 Vertex-Additionen . . . 34
5.3.4 Vertex-Verschiebungen . . . 34
5.3.5 Vertex-Löschungen . . . 38
5.3.6 Descriptor-Löschungen . . . 38
5.3.7 Descriptor-Relabellings und Vertex-Umbenennungen . . . 40

6 Umsetzung **41**
6.1 Wesentlichen Eigenschaften 41
6.2 Design 42
6.3 Package-Übersicht 43
6.4 Klassen-Übersicht 43
6.5 Interessante Details 44
6.6 Testing 46
6.7 Benutzung 47

7 Auswertung **49**

Danksagung **51**

Selbstständigkeitserklärung **52**

Literatur **53**

1 Einleitung

An dem Language and Information Engineering Lab (JULIE Lab) der Friedrich-Schiller-Universität Jena (FSU Jena) wird seit 2008 an dem Semedico-Projekt gearbeitet. Ziel ist es, eine semantische Suchmaschine zu entwickeln, welche es dem Nutzer ermöglicht schneller und gezielter als mit klassischen Suchmaschinen relevante Veröffentlichungen der Biomedizin zu finden.

Semedico baut dabei auf eine Vielzahl von Daten auf, insbesondere aber auf die Medical Subject Headings (MeSH), ein Thesaurus zur Sacherschließung von Texten aus der Medizin und den Biowissenschaften.

In der hier vorgelegten Studienarbeit wurde eine Software erarbeitet, die die Verwaltung der verschiedenen Datenquellen des Semedico-Systems vereinheitlicht und vereinfacht. Dies schließt Algorithmen ein, die das Upgrade des MeSH von einer Version zur nächsten vollständig automatisieren.

Medical Subject Headings

Der MeSH ist ein jährlich vom National Library of Medicine (NLM) herausgegebener polyhierarchischer Thesaurus. Er dient zum Katalogisieren von Medien sowie insbesondere zum Indizieren von biomedizinischen Veröffentlichungen. Nach Angaben der offiziellen Webseite werden Artikel von 5 400 der weltweit führenden biochemischen Journals mit Hilfe des MeSH indiziert [10].

Der MeSH enthält Entry Terms, welche jeweils einem Descriptors zugeordnet sind. Zusammen bilden sie eine nach Spezifizität geordnete Polyhierarchie [10]. Allgemeine Begriffe wie beispielsweise „Organisms“ sind folglich weit oben in der Hierarchie zu finden, während spezifischere Begriffe wie zum Beispiel „Hepatitis Viruses“ diesen untergeordnet sind.

Für das Semedico-Projekt dient der MeSH als eine der wesentlichen Datenquellen, da die hierarchische Struktur des MeSH in abgewandelter Form auch als Wissensbasis für Semedico dient. Diese Abwandlung ist insbesondere notwendig, um für Semedico nicht relevante Terme zu entfernen. Zu Veranschaulichung: Der MeSH enthält etwa 200 000 Entry Terms, welche jeweils einem der über 26 000 Descriptors zugeordnet sind. Im für Semedio angepassten MeSH finden sich lediglich noch knapp 5 000 Entry Terms und etwa eben so viele Descriptors.

Anmerkung zur Sprache

Aus Gründen der Lesbarkeit und um Verwirrungen zu vermeiden, werden in dieser Arbeit durchgehend die englischen Originalbezeichnungen von MeSH-spezifischen Begriffen verwendet.

Quelltext und Kommentare der entwickelten Software sind vollständig in Englisch. Daher werden Bezeichnungen von Funktionen die eine direkte Entsprechung im Quellcode finden ebenfalls in Englisch bezeichnet.

Abgesehen davon ist die Arbeit vollständig in Deutsch verfasst.

2 Problemstellung

Verwaltung der Semedico-Wissensbasis

Semedico kann als semantische Suchmaschine nur dann effektiv arbeiten, wenn ihr entsprechende Daten als Wissensbasis zur Verfügung stehen. Diese Daten umfassen im Moment unter anderem Daten aus dem MeSH und aus UniProt (eine umfassende Proteindatenbank [4]).

Eine Zielstellung dieser Studienarbeit ist es daher, eine einheitliche und vereinfachte Verwaltung der Datenquellen für Semedico zu entwickeln. Dies umfasst das Importieren von Daten unterschiedlichen Formats, die Repräsentation dieser Daten in internen Datenstrukturen, und auch den Export zu Semedico.

Abschnitt 6 *Umsetzung* beschreibt die dazu entwickelte Software.

Erstellen des Semedico-MeSH

Semedico hilft passende biomedizinische Publikationen zu finden, indem es dem Nutzer ermöglicht, Suchbegriffe mit klar definierter Bedeutung auf hierarchische Weise auszuwählen. Das ist möglich, weil diese Paper zuvor mit Hilfe des MeSH verschlagwortet wurden. Allerdings enthält der MeSH weitaus mehr als nur biomedizinische Fachtermini. Es sind ebenso unzählige andere Begriffe enthalten, welche für diese Anwendung überflüssig sind und sich sogar nachteilig auf die Performance bzw. Benutzbarkeit auswirken würden. Es ist also erforderlich aus der Gesamtheit des MeSH eine Teilmenge auszuwählen. Diese Teilmenge wird fortan Semedico-MeSH genannt.

Im JULIE Lab existiert bereits ein Algorithmus der diese Auswahl vornimmt. Allerdings ist er aufgrund fehlender Dokumentation, aufwendiger, d. h. teil-manueller, bzw. unklarer Handhabung und schlechter Flexibilität auf mittlere und längere Sicht unbrauchbar. Das Ergebnis dieses Algorithmus', angewandt auf den MeSH 2008, liegt vor und soll soweit möglich reproduziert werden.

Zum einen müssen also die aktuell vorhandenen Algorithmen und Daten analysiert werden, um herauszufinden welche Operationen den MeSH 2008 in den Semedico-MeSH 2008 überführen. 3.1 *MeSH* beschreibt dazu die Grundlagen, und 4 *Transformationsrekonstruktion bei MeSH-Graphen* beschäftigt sich hiermit.

Zum anderen müssen Methoden entwickelt werden, die diese Operationen, wie zum Beispiel Hinzufügen, Löschen oder Verschieben, auf importierte Daten anwenden, sowie solche, die diese Operationen extern abspeichern und wieder einlesen können. Siehe dazu Abschnitt 6.

Aktualisierung des Semedico-MeSH

Die National Library of Medicine veröffentlicht jährlich eine neue, aktualisierte Version des MeSH. Diese Aktualisierung umfasst unter anderem das Hinzufügen neuer Begriffe, das Löschen obsoleter Begriffe oder das Aufteilen eines Begriffs in mehrere. Wie im vorangegangenen Abschnitt erklärt, wird der Semedico-MeSH aus dem MeSH erstellt, indem eine Folge von Operationen angewandt wird. Diese Operationen können durch die Aktualisierung des MeSH unanwendbar geworden sein. Beispielsweise kann ein Element, das verschoben werden soll, fehlen.

Um eine aufwendige, manuelle Korrektur der Operationen zu vermeiden, sollen in dieser Studienarbeit Methoden entwickelt werden, welche voll-automatisiert eine syntaktische Korrektur vornehmen, während sie die Semantik so belassen, wie sie im aktualisierten MeSH vorgegeben ist. Das Vorgehen zur Lösung dieses Problems wird in Abschnitt 5 *Aktualisierung von Transformationsfolgen* beschrieben.

3 Grundlagen

3.1 MeSH

In diesem Abschnitt wird die interne Struktur des MeSH beschrieben.

3.1.1 MeSH Record Types

Descriptors

Wie bereits in der Einleitung erwähnt, bildet der MeSH eine hierarchische Struktur, deren zentrales Element sogenannte Descriptors sind. Ein Descriptor ist die Repräsentation eines Begriffs bzw. eines Begriffsraumes für den im MeSH ein Index-Eintrag existiert. Descriptors haben eine teils mehr, teils weniger scharf abgegrenzte Bedeutung, je nach dem wie dies aus Sicht der Autoren des MeSH zur Zeit den Anforderungen des Wissenschaftbetriebs entspricht. Mehr Informationen zu den Organisationsprinzipien finden sich in [11].

Neben den Descriptors gibt es zwei weitere sogenannte MeSH Record Types, die jedoch für diese Arbeit nicht von Interesse sind: Qualifiers und Supplementary Concept Records (SCR).

Qualifiers

Es existieren insgesamt 83 thematische Qualifiers. Sie erlauben es, im Zusammenspiel mit Descriptors, bestimmte Aspekte eines Themas sinnvoll zu gruppieren. Beispielsweise weist der Qualifier „Liver/drug effects“ darauf hin, dass sich ein Artikel nicht allgemein mit der Leber, sondern mit den Auswirkungen von Medikamenten beschäftigt.

Supplementary Concept Records

Die Supplementary Concept Records (SCR) werden vor allem genutzt, um Chemikalien und Medikamente zu indizieren. Während sie grundsätzlich sehr ähnliche Attribute besitzen, besteht der wesentliche Unterschied zu Descriptors darin, dass SCRs keine Tree Numbers besitzen und sie folglich nicht selbst eine Hierarchie bilden. Stattdessen werden sie jeweils einem oder mehreren Descriptors zugeordnet und ergänzen bzw. spezifizieren so dessen Bedeutung.

3.1.2 MeSH Komponenten und XML

Der MeSH ist in zwei Formaten verfügbar: als XML-File sowie als ASCII-File. In dieser Arbeit wird die XML-Version verwendet, da zum einen die XML-Format reicher als die ASCII-Format ist [13], und da zum anderen durch die Verwendung der seitens der NLM bereitgestellten dtd-Datei ein validiertes Parsen möglich ist.

Nachfolgend werden die wichtigsten Begriffe im MeSH, nämlich Descriptor, Tree Number, Concept und Term für sich selbst und als XML-Element erklärt. Abbildung 1 zeigt den strukturellen Aufbau (der hier wichtigen Elemente) des MeSH.

Für eine detaillierte und vollständige Beschreibung aller XML-Elemente wird auf [16] verwiesen.

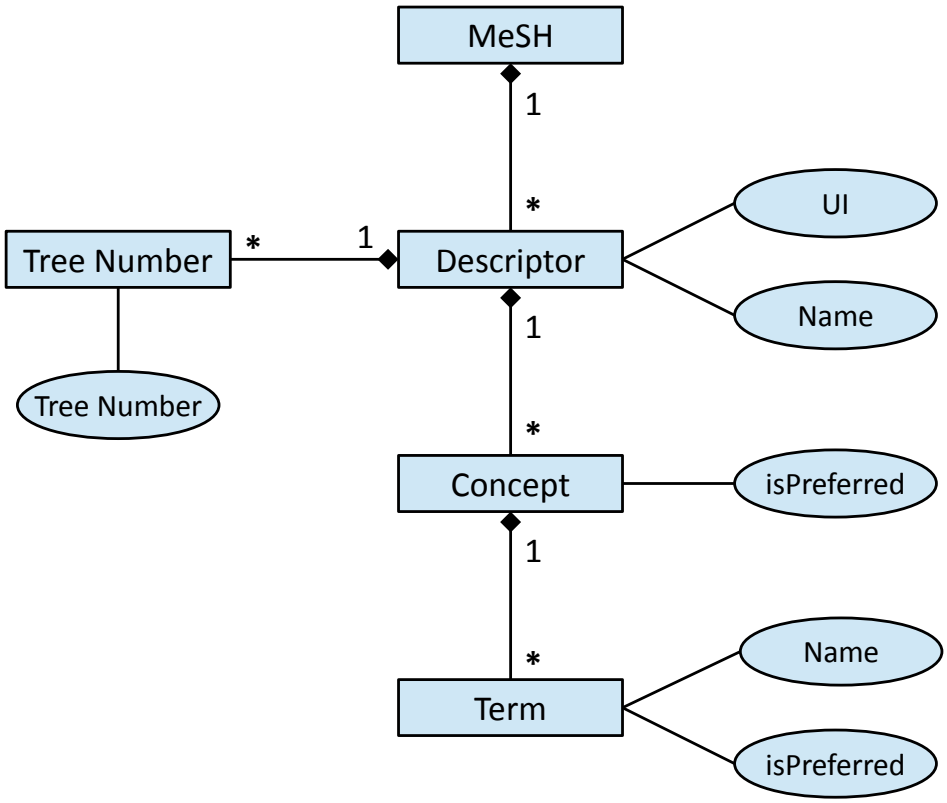

Abbildung 1: Datenstruktur des MeSH

MeSH

Wenn man von SCRs absieht, besteht der MeSH aus einer Menge von Descriptors. Das entsprechende XML-Root-Element ist <DescriptorRecordSet>, welches alle Descriptors als direkte Kinder enthält.

Descriptor

Die Bedeutung der Descriptors wurde bereits weiter oben diskutiert.

Ein Descriptor als XML-Element ist ein <DescriptorRecord> und besteht, unter anderem, aus einer Menge von Tree Numbers (XML-Element: <TreeNumberList>) sowie aus einer Menge von Concepts (XML-Element: <ConceptList>). Tree Numbers betten Descriptors in eine oder mehrere Positionen in der Begriffshierarchie ein. Concepts erlauben eine feinere Einteilung der Bedeutung eines Descriptors.

Ein Descriptor kann eindeutig durch seine UI (XML-Element: <DescriptorUI>) und seinen Namen (XML-Element <DescriptorName>) identifiziert werden. Der Name ist dabei der <PreferredName> des <PrefferedConcepts> des Descriptors.

Es sind keine expliziten Relationen, wie zum Beispiel Inklusion auf Descriptors, definiert. Siehe dazu auch den Abschnitt unten zu Tree Numbers.

Ein Descriptor wird auch als Descriptor Record oder Main Heading bezeichnet.

Concept

Ein Concept besteht aus einer Menge von Terms und kann über seine ID (XML-Element <ConceptUI>) eindeutig identifiziert werden.

Ein Concept beschreibt eine begriffliche Untereinheit eines Descriptors. Alle Terme innerhalb eines Concepts sind synonym zueinander.

Tree Number

Eine Tree Number ist eine alpha-numerische Zeichenkette und verweist an einer bestimmte Position des MeSH auf den Descriptor, der diese Tree Number enthält. Das dazugehörige XML-Element ist <TreeNumber>.

Es ist an dieser Stelle zu bemerken, dass die hierarchische Struktur des MeSH nicht explizit als XML-Element vorliegt. Vielmehr ergibt sich die Struktur allein implizit durch die Namensgebung der Tree Numbers, also eben aus der Eigenschaft, dass eine Tree Number $x.y.z$ dafür steht, dass die Tree Number $x.y$ der Vater der Tree Number $x.y.z$ ist.

Diese Verflechtung von ID einer Tree Number und Strukturinformation ist aus unserer Sicht nicht wünschenswert: Unter Aufrechterhaltung dieser Eigenschaft führt ein lokales Verändern der hierarchischen Struktur zu nicht-lokalen Änderungen der Tree Number. Beispielsweise führt ein Verschieben einer Tree Number dazu, dass auch alle untergeordneten Tree Numbers umbenannt werden. Das macht ein Arbeiten mit der Hierarchie unnötig kompliziert. In Unterabschnitt 3.3 wird eine Möglichkeit diskutiert, wie diese Verflechtung beseitigt werden kann.

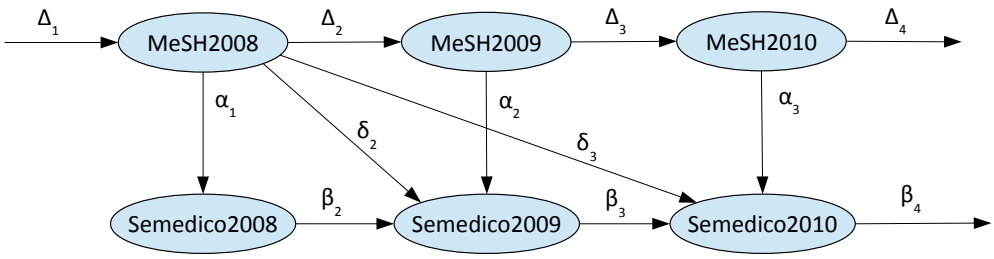

Abbildung 2: Möglichkeiten zur Erzeugung des jährlichen Semedico-MeSH

Term

Ein Term wird auch als Entry Term bezeichnet. Und [6] beschreibt Entry Terms folgendermaßen:

> „Entry terms, sometimes called „See cross-references“ in printed listings, are synonyms, alternate forms, and other closely related terms in a given MeSH record that are generally used interchangeably with the preferred term for the purposes of indexing and retrieval, thus increasing the access points to MeSH-indexed data.“

Das dazugehörige XML-Element ist <Term>.

3.1.3 Jährliche Änderungen

Jedes Jahr erscheint eine neue, an die aktuellen Entwicklungen der Wissenschaft angepasste Version des MeSH. Auch für Semedico soll die jeweils aktuelle Version des MeSH als Datengrundlage verwendet werden. Dazu wird, wie zuvor in Abschnitt 2 erklärt, eine Folge von Operationen auf den MeSH angewandt, um daraus den Semedico-MeSH zu erzeugen. Problematisch ist hierbei, dass mit jeder Änderung am MeSH die Operationen zur Erzeugung des Semedico-MeSH potentiell unmöglich werden. Beispielsweise kann eine Tree Vertex nicht mehr verschoben werden, falls sie in der neuen Version des MeSH nicht mehr existiert.

Lösungsmöglichkeiten

Abbildung 2 zeigt schematisch die Lösungsvarianten, die nachfolgend konzeptionell erläutert werden.

Gegeben sind MeSH2008 sowie Semedico2008, also der MeSH des Jahres 2008 und der darauf basierende Semedico-MeSH. Daraus kann α_1 bestimmt werden und ist damit ebenfalls gegeben. Δ_i, α_i, β_i, δ_i seien Operationsfolgen, die einzelne Versionen des MeSH, wie in Abbildung 2 dargestellt, ineinander überführen. Ziel ist es, den Semedico-MeSH für die Folgejahre zu erstellen. Um das zu erreichen, gibt es wenigstens drei Möglichkeiten:

(α) Den MeSH des aktuellen Jahres in den Semedico-MeSH überführen. Dazu wird α_i als Aktualisierung von α_{i-1} unter Zuhilfenahme von Δ_i erstellt.

(β) Den Semedico-MeSH des vorangegangen Jahres aktualisieren. Hierzu wird β_i als modifizierte Teilmenge von Δ_i erstellt - als Menge all der Operationen aus Δ_i die auf Semedico20xx angewandt werden können.

(δ) Direkt aus dem MeSH2008 den aktuellen Semedico-MeSH erzeugen. Dafür wird δ_i als Vereinigung von δ_{i-1} und Δ_i erzeugt.

Für diese Studienarbeit wurde die erste Variante (α) umgesetzt. Zum einen werden also die jährlichen Änderungen (Δ) des MeSHs benötigt. In Abschnitt 4 *Transformationsrekonstruktion bei MeSH-Graphen* wird beschrieben wie diese ermittelt werden. Zum anderen muss α aktualisiert werden. Das Vorgehen dazu ist in Abschnitt 5 *Aktualisierung von Transformationsfolgen* erklärt. Als Basis werden hier in den restlichen Unterabschnitten dieses Abschnitts grundlegende Begriffe der Graphentheorie, eine Formalisierung des MeSH, sowie verschiedene Operationen und Relationen auf dem formalisierten MeSH eingeführt.

Änderungen verfolgen

Online sind für die jeweils aktuelle Version Änderungslisten verfügbar. Allerdings sind diese weder umfassend noch vollständig[1]. Das heißt sie enthalten nicht alle Informationen zu den Änderungen. Beispielsweise gibt es keine Informationen dazu *welche* Attribute eines Descriptors geändert wurde, sondern nur, *dass* er geändert wurde. Und sie enthalten auch nicht alle Änderungen. Beispielsweise gibt es überhaupt keine Informationen zu Aufteilungen von Descriptors.

Daher ist es notwendig eigene Methoden zu entwickeln, um diese Änderungen zu bestimmen. Siehe dazu Abschnitt 4 *Transformationsrekonstruktion bei MeSH-Graphen*.

3.2 Graphentheorie

Dieser Abschnitt definiert die klassischen Begriffe der Graphentheorie. Sie sind in wesentlichen Teilen [5] entnommen.

Definition (Graph, Knoten, Kante)
Ein Graph G ist ein 2-Tupel (V, E) mit $E \subseteq V^2$. Die Elemente der Menge V werden Knoten genannt. Die Elemente der Menge E werden Kanten genannt. Ist $\{x, y\} \in E$, so bezeichnen wir diese Kante auch mit xy.

Definition (Knotenmenge, Kantenmenge)
Sei $G = (W, F)$ ein Graph. Wir bezeichnen dann $E(G) = F$ als Kantenmenge und $V(G) = W$ als Knotenmenge Gs.

[1] See „4. MeSH Vocabulary Changes for 2013“ of `http://www.nlm.nih.gov/mesh/introduction.html`

Definition (Teilgraph)
Ein Graph S heißt Teilgraph eines Graphen G genau dann, wenn gilt

$$E(S) \subseteq E(G) \wedge V(S) \subseteq V(G)$$

Definition (Pfad, Zyklus, Kreis)
Ein Pfad P in einem Graphen G ist eine Folge von Knoten $(x_0, \ldots, x_k)$, so dass:

$$\forall (i = 0, \ldots, k-1) : x_i x_{i+1} \in E(G)$$

Die Länge des Pfades P ist gleich $k-1$ und wird mit $l(P)$ bezeichnet.

Sei $s = x_0$ und $t = x_k$. Dann heißt P auch $s-t$-Pfad.

Existieren $m, n \in \{0, \ldots, k\}$ so dass $x_m = x_n$, dann heißt P Zyklus.

Gilt $x_0 = x_k$ und sind sonst alle x_i verschieden, so heißt P Kreis.

Definition (Zusammenhängend)
Ein nicht leerer Graph heißt zusammenhängend, wenn er für je zwei seiner Knoten x und y einen $x-y$-Pfad enthält.

Definition (Baum, Wurzel)
Ein Graph G heißt Baum, falls er zusammenhängend ist und es kein $S \subseteq V(G)$ mit $l(S) > 1$ gibt, so dass S ein Kreis in G ist.

In einem Baum G gibt es einen beliebigen aber ausgezeichneten Knoten aus $V(G)$, der Wurzel genannt wird. Wir schreiben dafür auch $root(G)$.

Definition (Tiefe eines Knotens)
Sei G ein Baum und $v \in V(G)$. Sei weiter P ein $root(G) - v$-Pfad. Dann setzen wir die Tiefe vs als $d(v) = l(P)$.

Definition (Verwandtschaft)
Zwei Knoten x, y eines Baumes G heißen benachbart oder adjazent, falls $xy \in E(G)$ gilt.

In einem Baum G heißt ein Knoten x Vater eines Knoten y und der Knoten y heißt Kind des Knotens x, falls gilt:

$$xy \in E(G) \wedge d(x) < d(y)$$

Die Menge aller Kinder eines Knoten x wird auch als $children(x)$ und der Vater eines Knotens x auch als $dad(x)$ bezeichnet.

In einem Baum G heißt ein Knoten x Vorfahre eines Knoten y und der Knoten y heißt Nachkomme des Knotens x falls ein $x-y$-Pfad in G existiert und $d(x) < d(y)$ gilt.

Definition (Label und Typ)
Ein Label besteht aus einem textuellem Schlüssel und einem beliebigen Wert, und wird einem Knoten oder einer Kante zugeordnet.

Die Menge der Schlüssel der Label eines Knoten v bzw. einer Kante k wird als $label(v)$ bzw. $label(k)$ bezeichnet.

Den Wert eines Labels l eines Knotens v bzw. einer Kante k identifizieren wir mit $v.l$ bzw. $k.l$. Ist der Wert von l leer, so sagen wir auch v bzw. k sind vom Typ l.

Ein Label erlaubt es damit Knoten und Kanten beliebige Zusatzinformationen zuzuordnen.

3.3 Der MeSH als Graph

Gegeben sei ein MeSH M. Dieser wird, wie nachfolgend beschrieben, als *MeSH-Graph* G_M formalisiert:

- Jeder Descriptor d aus M wird durch einen Knoten v_d vom Typ Descriptor in G_M repräsentiert. v_d erhält zwei Label name und ui deren Werte der Name bzw. die UI von d sind.
- Jede Tree Number t aus M wird durch einen Knoten v_t vom Typ TreeVertex in G_M repräsentiert. v_t erhält ein Label name dessen Wert die tatsächliche Tree Number t s ist. Weiterhin erhält G_M eine Kante vom Typ Descriptor von v_t nach v_d, wobei v_d der Knoten ist durch den der Descriptor von t repräsentiert wird.
- Für jedes Paar Tree Numbers s und t aus M, für die s eine direkte Unterkategorie von t ist, enthält G_M eine Kante des Typs TreeVertex von $\{v_s, v_t\}$.
  s ist eine direkte Unterkategorie von t wenn die Tree Number von s sich als nachfolgende Konkatenation schreiben lässt. Sei dazu tnr die Tree Number von t:

  $$tnr.[0123456789]^*$$

- G_M erhält einen ausgezeichneten Knoten r vom Typ TreeVertex.
- Für jede Tree Number t die keinen Punkt enthält, gibt es in G_M eine Kante $\{r, t\}$ vom Typ TreeVertex.

Vereinfacht gesagt werden Descriptors und Tree Numbers als Knoten, und die Strukturinformation der Tree Numbers (d.h. der Wert ihrer Tree Number und welcher der dazugehörige Descriptor ist) als Kanten angeordnet. Damit ist G_M eine vollständige Repräsentation von M und es ergeben sich - entsprechend der Typen der Kanten und Knoten - zwei interessante Graphen wie folgt:

Baum der Tree Vertices

Abgesehen vom Wurzelknoten wird jede Tree Number des MeSH durch genau einen Knoten (des Typs TreeVertex) repräsentiert, und umgekehrt. Ebenso wird die Hierarchieinformation jeder Tree Number durch genau eine Kante (des Typs TreeVertex) repräsentiert, und umgekehrt. Da Tree Numbers per Definition höchstens einen Vater besitzen, ist die Menge aller Kanten und Knoten vom Typ TreeVertex mit Knoten r als Wurzel ein Baum und ein Teilgraph von G_M.

Wenn nachfolgend vom MeSH-Tree bzw. verkürzt Tree gesprochen wird, ist dieser Baum gemeint. Ebenso meinen wir nachfolgend mit Descriptor und Tree Vertex im Allgemeinen einen Knoten vom Typ Descriptor bzw. TreeVertex.

Polyhierarchie der Descriptors

Parallel dazu existiert eine Polyhierarchie aus Descriptors: Jeder Tree Number ist genau ein Descriptor zugeordnet und über den MeSH-Tree stehen auch diese Descriptors untereinander in hierarchischer Beziehung. Während allerdings eine Tree Number genau einem Knoten im MeSH-Tree zugeordnet ist, besitzt ein Descriptor so viele Kanten in G_M, wie er Tree Numbers besitzt. Anschaulich gesprochen entsteht die Polyhierarchie der Descriptors aus G_M, indem alle Knoten eines jeden Descriptors zu einem einzigen vereint. Der entstehende Graph ist im Allgemeinen kein Baum.

Diese Polyhierarchie ist eine vereinfachte Repräsentation des MeSH, da hier ein wesentlicher Teil der Hierachie verloren gehen: die genaue Beziehung der Tree Numbers untereinander. Erhalten bleibt nur die Hierarchie auf Descriptor-Ebene.

3.3.1 Operationen und Transformationen

Es folgt eine Auflistung von Definitionen zu Operationen auf MeSH-Graphen. Sei MeSH die Menge aller MeSH-Graphen. Eine MeSH-Operation o ist Operation der Form

$$o : MeSH \times Param \to MeSH$$

wobei $Param$ die Menge der Parameter der Operation o darstellt.

Das Fixieren der Parameter $Param$ einer konkreten Operation o, bildet o auf die *Transformation* t_o ab. t_o ist folglich eine innere unäre Operation über MeSH.

Eine Folge von Transformationen ist eine Menge von Transformationen mit einer totalen Ordnung. Die Anwendung einer Transformationsfolge T auf einen MeSH-Graphen G bedeutet die sequentielle Anwendung aller Transformationen aus T auf G entsprechend ihrer Ordnung.

Nachfolgend beschreiben wir Operationen auf MeSH-Graphen. Sie sind in dem Sinne atomar, als das sie alle grundlegenden Operationen beschreiben, die wir im Kontext

das MeSH als *eine* Operation ansehen. Beispielsweise zwar kann ein Vertex-Verschieben durch eine Kombination von Vertex-Löschung und Vertex-Addition realisiert werden. Allerdings würde ein Verschieben dann mehr als eine Operation benötigen. Das ist zu vermeiden, da wir in Abschnitt 4 möglichst kurze Transformationsfolgen suchen und folglich eine einheitliche Länge für jede elementare Operation voraussetzen.

Sei nachfolgend $G_M \in$ MeSH und T_M der zu G_M gehörende MeSH-Tree, entsprechend Unterabschnitt 3.3.

Definition (Vertex-Addition)
Schreibweise: vertexAdd(name, parent, desc)

Fügt im T_M eine Tree Vertex v mit Namen name *als Kind der Tree Vertex* parent *ein. Diese Tree Vertex wird dabei dem Descriptor* desc *zugeordnet, d. h. in G_M wird eine Kante vom Typ* Descriptor *von v zu* desc *eingefügt.*

Verkürzte Schreibweise: vertexAdd(loc, desc)*, wobei* loc *gerade das Paar aus* name *und* parent *ist.*

Definition (Vertex-Löschung)
Schreibweise: vertexDel(vertex, rec)

Wenn rec = true *werden* vertex *und auch alle Nachkommen von* vertex *aus T_M entfernt. Ansonsten wird nur* vertex *entfernt. Ebenso werden alle Kanten der Form $\{vertex, v\}$ mit $v \in V(G_M)$ entfernt.*

Definition (Vertex-Verschiebung)
Schreibweise: vertexMove(v, oldParent, newParent, oldDesc, newDesc)

Verschiebt eine Tree Vertex v*, so dass* newParent *der Vater von* v *in in T_M und* newDesc *der zu* v *gebundene Descriptor ist. Der bisherige Vater von* vertex *ist* oldParent*, und der bisher zu* v *gebundene Descriptor ist* oldDesc*.*

Damit ist ein Verschieben ein Umhängen von Kanten und folglich werden Tree-Vertices innerhalb eines MeSH-Trees immer mitsamt all ihrer Kinder verschoben. Außerdem müssen sich nicht Descriptor *und* Vater von v ändern. Wird nur der zu v gehörende Descriptor verändert, so nennen wir das als Spezialfall dieser Operation auch Vertex-Neuverknüpfung oder Vertex-Rebinding.

Definition (Vertex-Umbenennung)
Schreibweise: vertexRen(vertex, newName)

Verändert den Wert des Labels name *der Tree-Vertex* vertex *zu* newName*.*

Definition (Descriptor-Addition)
Schreibweise: descAdd(desc, locs)

Fügt den Descriptor desc *hinzu. Dem Descriptor sind eine Menge von Tree Vertices zugeordnet, die ebenfalls hinzugefügt werden. Ihr Name und Position wird durch* locs *beschrieben.* locs *besteht aus Paaren von* name *und* parentVertex*, wobei* name *der Name der Tree Vertex und* parentVertex *der Vater dieser Tree Vertex ist.*

Definition (Descriptor-Löschung)
Schreibweise: descDel(desc)

Löscht den Descriptor desc. Löscht auch alle zu desc gehörigen Tree Vertices sowie entsprechend sonst verwaiste Kanten.

Definition (Descriptor-Relabelling)
Schreibweise: descRel(desc, newName)

Ändert den Wert des Labels name des Descriptors desc nach newName.

Definition (Descriptor-Umbenennung)
Schreibweise: descRen(desc, newUI)

Ändert den Wert des Labels ui des Descriptors desc nach newUI.

3.3.2 Kommutativität von Kompositionen der Transformationen

Die im vorigen Abschnitt definierten Transformationen lassen sich in folgende Gruppen einteilen:

(1) Transformationen die ausschließlich Knoten und Kanten hinzufügen: Vertex-Addition und Descriptor-Addition.

(2) Transformationen die ausschließlich Knoten und Kanten löschen: Vertex-Löschung und Descriptor-Löschung

(3) Transformationen die ausschließlich Labels ändern: Vertex-Umbenennung, Descriptor-Relabelling und Descriptor-Umbenennung

(4) Vertex-Verschiebungen

Kompositionen von Transformationen der ersten Gruppe sind kommutativ, denn die Komposition von Hinzufügen von Knoten oder Kanten ist kommutativ, da die Komposition von Vereinigungen von Mengen kommutativ ist. Analog zur ersten Gruppe sind auch Kompositionen der Transformationen der zweiten Gruppe kommutativ.

Kompositionen von Transformationen der dritten Gruppe sind im Allgemeinen nicht kommutativ: Beispielsweise ist das Ergebnis verschieden, je nachdem ob man die UI eines Descriptors erst in 001 und dann in 002 ändert, oder umgekehrt. Allerdings lässt sich jede solche nicht kommutative Komposition trivial als kürzere kommutative Komposition darstellen, also als Kompositionen von weniger Transformationen. Statt im eben genannten Beispiel die UI erst in 001 und dann in 002 zu ändern, führt ein unmittelbares Ändern nach 002 zum selben Ergebnis. Folglich sind minimal lange Kompositionen von Transformationen der dritten Gruppe kommutativ.

Kompositionen von Vertex-Verschiebungen sind im Allgemeinen nicht kommutativ, gleichwohl sind minimal lange Kompositionen von Vertex-Verschiebungen minimal. Die Argumentation ist analog zu der bei Gruppe 3.

Folglich sind minimal lange Kompositionen der Transformationen innerhalb einer jeden Gruppe kommutativ.

Die zuvor definierten Operationen sind, mit Ausnahme der Descriptor-Addition, keine Funktionen sondern nur partielle Funktionen. Der Grund ist, dass sie nur dann definiert sind, wenn der MeSH-Tree auf den sie angewandt werden, die Tree-Vertices bzw. Descriptors enthält, die als Parameter für die Operation angegeben werden: Beispielsweise ist ein Löschen einer Tree-Vertex nur definiert, wenn sie auch existiert. Folglich ist eine Komposition von Transformationen der unterschiedlichen Gruppen im Allgemeinen nicht kommutativ. Zum Beispiel kann eine Tree-Vertex nicht verschoben werden, bevor sie hinzugefügt wurde. Fordert man aber eine minimal lange Komposition, dann ist die Komposition – selbst wenn sie Transformationen unterschiedlicher Gruppen enthält – kommutativ: Die Nicht-Kommutativität war eine Folge der sequentiellen Abhängigkeit zweiter Transformationen. Diese Abhängigkeit kann aber in minimal langen Transformationsfolgen nicht mehr existieren, da zwei sequentiell abängige Transformationen immer zu (höchstens) einer zusammengefasst werden können: wird ein Tree-Vertex hinzugefügt und dann verschoben, kann sie direkt an ihrer finalen Positionen hinzugefügt werden. Wird eine Tree-Vertex hinzugefügt und dann wieder gelöscht, können diese Transformationen weggelassen werden. Usw.

Minimal lange Kompositionen von Transformationen sind folglich kommutativ.

Eine formale Herleitung dieser Behauptung wird hier ausgelassen, wäre aber wünschenswert.

3.3.3 Relationen

Es folgt eine Auflistung von Relationen im Kontext von MeSH-Trees.

Definition (Gleichheit von MeSH-Trees)
Zwei MeSH-Graphen S und T heißen gleich, genau dann, wenn es für jeden Descriptor und für jede Tree Vertex aus S einen Descriptor bzw. Tree Vertex aus T gibt, sodass diese gleich sind.

Definition (Gleichheit von Descriptors)
Zwei Descriptors sind gleich genau dann, wenn

- *ihre Namen übereinstimmen*
- *ihre UIs übereinstimmen*

Definition (Gleichheit von Tree Vertices)
Zwei Tree Vertices sind gleich genau dann, wenn

- *ihre Namen übereinstimmen*
- *ihr Vater gleich ist*
- *ihre Kinder gleich sind*

3.3.4 Ähnlichkeit

Definition (Ähnlichkeitsmaß für Tree Vertices)
Gegeben seien zwei Tree Vertices v_1 und v_2. Wir definieren die Ähnlichkeit von v_1 und v_2 als $sim(v_1, v_2)$ folgendermaßen rekursiv:

Falls $v_1.name \neq v_2.name$:

$$sim(v_1, v_2) = 0$$

Sonst:

$$sim(v_1, v_2) = |\{(c_1 \in children(v_1), c_2 \in children(v_2)) : c_1.name = c_2.name\}| + \sum_{c_1 \in children(v_1), c_2 \in children(v_2)} sim(c_1, c_2)$$

Da alle Werte stets größer oder gleich 0 sind, ist *sim* eine monotone Funktion. Ihr Wert wird umso größer, umso ähnlicher sich zwei Knoten v_1 und v_2 sind. Ähnlichkeit ist hier ein Maß dafür, wie viele Nachkommen von v_1 und v_2 zusammenhängend namentlich übereinstimmen.

4 Transformationsrekonstruktion bei MeSH-Graphen

In diesem Kapitel wird die Vorgehensweise zur automatischen Rekonstruktion von Transformationen besprochen, welche zwei MeSH-Graphen ineinander überführen. Zuerst spezifizieren und diskutieren wir die Problemstellung. Danach stellen wir unseren Algorithmus vor und bewerten ihn.

4.1 Spezifikation

Gegeben: Zwei MeSH-Graphen S und T, entsprechend Unterabschnitt 3.3.

Gesucht: Folge von Transformationen O, entsprechend Unterunterabschnitt 3.3.1, so dass $O(S) = T$, mit minimalem $|O|$.

4.2 Motivation der Spezifikation

Bei der Rekonstruktion der Überführung eines MeSH-Trees sind wir mit mehreren Problemen konfrontiert:

Nichteindeutigkeit der Lösung: Ohne die Forderung der Minimalität von $|O|$ existieren unendlich viele Lösungen (z. B. durch wiederholtes Hinzufügen und Löschen des gleichen Descriptors). Die Forderung einer Transformationsfolge (anstelle einer Menge von Transformationen) ist notwendig, da Transformationen im Allgemeinen nicht kommutativ sind. Durch die Forderung der Minimalität sind allerdings bestimmte Gruppen von Transformationen kommutativ (siehe Unterunterabschnitt 3.3.2). Damit ist im Allgemeinen die Lösung nicht eindeutig.

Nichteindeutigkeit des Problems: Während wir sicherstellen können, dass immer eine Transformationsfolge gefunden wird die S in T überführt, sind wir nicht in der Lage die Qualität der Lösung objektiv einzuschätzen, da keine Musterlösungen vorliegen bzw. das tatsächliche Problem selbst nicht exakt formal darstellbar ist. Mit anderen Worten: es gibt keine streng rationale Vorgehensweise der Wissenschaftler am NLM nach der sie den MeSH verändern, da der MeSH kontinuierlich den realen Anforderungen und Entwicklungen der Wissenschaft angepasst wird. Entsprechend kann auch kein Algorithmus diesen Vorgang nachvollziehen. Stattdessen versuchen wir eine formale Spezifikation anzugeben, die dem realen Problem möglichst gerecht wird.

Vermischung von Theorie und Anwendung: Aus den vorangegangenen Punkten ergibt sich, dass wir auf Heuristiken angewiesen sind, welche sich nur durch domänenspezifische Plausibilitätsargumente motivieren lassen.

4.3 Algorithmus

Zusammenfassend ist die Vorgehensweise des Algorithmus' die Folgende: Wir gehen von T aus und durchlaufen all dessen Descriptors bzw. Tree Vertices, um nacheinander für jeden Descriptor bzw. jede Tree Vertex die Transformation zu bestimmen, durch die dieser Descriptor bzw. diese Tree Vertex in T vorhanden ist. Zur Bestimmung einer konkreten Transformation verwenden wir Entscheidungsbäume. Dabei vergleichen wir ob bzw. wie S und T lokal übereinstimmen und leiten daraus die durchgeführte Transformation ab.

Wie in 3.3.1 *Operationen und Transformationen* beschrieben, existieren verschiedene Transformationstypen. Wir bestimmten die konkreten Transformation in folgender Reihenfolge:

1. Descriptor-Additionen, Descriptor-Relabellings, Descriptor-Umbenennungen
2. Vertex-Neuverknüpfungen
3. Vertex-Verschiebungen, Vertex-Umbenennungen, Vertex-Additionen
4. Descriptor-Löschungen, Vertex-Löschungen

Die Reihenfolge ergibt sich aus dem Algorithmus, wie folgt:

- Descriptors und Tree-Vertices werden als gelöscht erkannt, genau dann, wenn sie nicht als anderweitig verändert erkannt wurden. Folglich müssen Löschungen nach allen anderen Transformationen bestimmt werden.
- Die beiden Entscheidungsbäume für Vertex-Transformationen (für 2. und 3. oben) setzen voraus, dass alle Descriptor-Transformationen bekannt sind. Daher werden Descriptor-Transformationen vor Vertex-Transformationen bestimmt.
- Die Reihenfolge von 2. und 3. ist beliebig.

Innerhalb der obigen Schritte gehen wir iterativ vor, wie in nachfolgenden Abschnitten für die einzelnen Transformationstypen beschrieben wird. Der Pseudocode in Listing 1 veranschaulicht das Vorgehen.

4.4 Vorgehen für Transformationstypen

Bezeichnungen

Es folgen Erläuterungen zu den Bezeichnungen in diesem Abschnitt.

```
Eingabe: MeSH-Graph S (Ausgangsgraph), MeSH-Graph T (Zielgraph)

// Initialisierung
leere O
setze VT gleich der Menge aller Tree-Vertices in T
setze VS gleich der Menge aller Tree-Vertices in S
setze DS gleich der Menge aller Descriptors in S
setze DT gleich der Menge aller Descriptors in T

// Descriptor-Transformationen (außer Löschungen)
für alle (dT in DT):
  bestimme Transformation o die zum Auftreten von dT in T führt
  füge o an O an

// Vertex-Rebindings
für alle (vT in VT):
  wenn (vT ist Resultat eines Vertex-Rebindings)
    bestimme diese Transformation o
    füge o an O an

// Vertex-Transformationen (außer Löschungen)
traversiere MeSH-Tree of T mit Tiefensuche, sei vT das aktuelle Element:
  bestimme Transformation o die zum Auftreten von vT in T führt
  füge o an O an

// Descriptor-Löschungen
für alle (dS in DS):
  wenn (dS wurde aus S gelöscht um T zu erzeugen)
    füge diese Löschung an O an

// Vertex-Löschungen
für alle (vS in VS):
  wenn (vS wurde aus S gelöscht um T zu erzeugen)
    füge diese Löschung an O an

Ausgabe: Transformationssequenz O
```

Listing 1: Algorithmus zur Transformationsrekonstruktion

vT	gerade betrachtete Tree Vertex aus T
vS	Entsprechung vT s in S, also vT s Ursprung
dT	Descriptor vT s in T, bzw. gerade betrachteter Descriptor
dS	Descriptor vS s in S
dad(v)	Vater der Tree Vertex v
name(d)	Name eines Descriptors d
ui(d)	UI eines Descriptors d
S(ui)	Descriptor in S der UI ui besitzt
S(dname)	Descriptor in S der Name dname besitzt
S(vname)	Tree Vertex in S der Name vname besitzt

4.4.1 Descriptor-Transformationen

Um alle Transformationen der Descriptors (außer Löschungen) herauszufinden, traversieren wir die Menge der Descriptors in beliebiger Reihenfolge und nehmen dann eine Fallunterscheidung vor, wie in Abbildung 3 dargestellt.

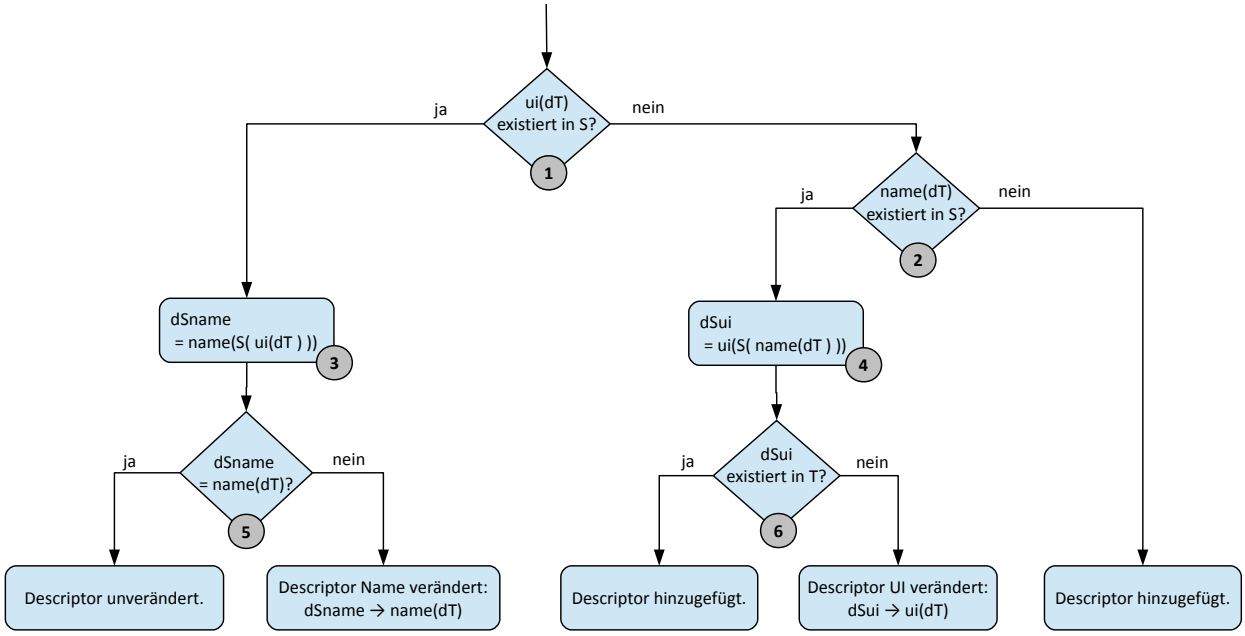

Abbildung 3: Bestimmen der Transformationen auf Descriptors

Es folgen Erklärungen zu den einzelnen Schritten in Abbildung 3.

(1) überprüft, ob es in S einen Descriptor mit selber UI wie dT gibt.

(2) überprüft, ob es in S einen Descriptor mit selbem Namen wie dT gibt. Wenn auch dies nicht der Fall ist, dann ist dT ein neuer Descriptor.

(3) Für Leserlichkeit: setzt dSname gleich dem Namen des Descriptors in S, der die selbe UI wie dT hat.

(4) Für Leserlichkeit: setzt dSui gleich der UI des Descriptors in S, der den selben Namen wie dT hat.

(5) überprüft, ob neben der UI auch der Name dT s und S(ui(dT)) s übereinstimmt. Wenn dem so ist, dann ist dT unverändert. Ansonsten hat sich der Name von dSname zu name(dT) geändert. Siehe dazu auch die Anmerkungen im nachfolgenden Abschnitt *Aufteilen eines Descriptors*.

(6) überprüft, ob es einen Descriptor dS mit UI dSui in T gibt. Ist dies der Fall, dann ist dT entstanden, indem der Name dS s geändert wurde. Ansonsten existiert also kein „ähnlicher“ Descriptor in S, folglich ist dS hinzugefügt wurden. Siehe dazu auch die Anmerkungen im nachfolgenden Abschnitt *Aufteilen eines Descriptors*.

Aufteilen eines Descriptors

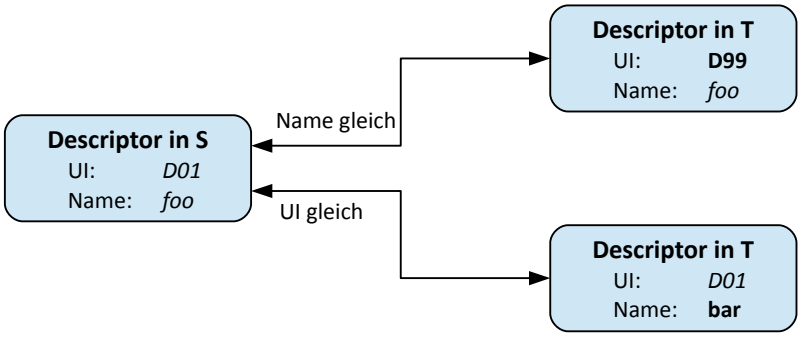

Abbildung 4: Descriptor-Aufteilung

Sei (x,y) für diesen Abschnitt die Bezeichnung für einen Descriptor mit UI x und Name y. Bei der Bestimmung der Descriptor-Transformationen zweier MeSH-Trees S und T kann es zu der in Abbildung 4 gezeigten Situation kommen: Beide Descriptors aus T, (D01,bar) und (D99,foo), sind potentiell aus (D01,foo) aus S entstanden. Wir müssen uns für der beiden Überführungsmöglichkeiten entscheiden:

1. Descriptor (D99,foo) ist neu. Descriptor (D01,bar) ist durch Änderung des Namens (D01,foo) s entstanden.
2. Descriptor (D01,bar) ist neu. Descriptor (D99,foo) ist durch Änderung der UI (D01,foo) s entstanden.

Wir entscheiden uns aus heuristischen Gründen für Variante 1: Ein Term aus (D01,foo) wurde in einen neuen Descriptor verschoben (→ (D99,foo)). Dieser Term war gerade der Preferred Term, weshalb sich der Name geändert hat (→ (D01,bar)). Im Gegensatz dazu sehen wir keinen Grund, warum man im MeSH die UI eines Descriptors ändern sollte.

4.4.2 Vertex-Neuverknüpfung

Vertex-Neuverknüpfungen, also das Verändern, zu welchem Descriptor eine Tree Vertex gehört, sind ein Spezialfall des Vertex-Verschiebens. Dieser Sonderfall ist in 3.3.1 *Operationen und Transformationen* dargestellt.

Wir iterieren über alle die Tree Vertices vT in T, deren Namen sich nicht geändert haben, und erkennen eine Vertex-Neuverknüpfung, falls (i) weder Name noch UI der beiden Descriptors dT und dS übereinstimmen und es außerdem (ii) kein Descriptor Relabelling bzw. Descriptor Renaming gibt, welches name(dS) in name(dT) bzw. ui(dS) in ui(dT) überführt.

Die erste Bedingung ist evident, da sich bei einer Vertex-Neuverknüpfung gerade der Descriptor ändert. Die zweite Bedingung erklärt sich dadurch, dass ein Umbenennen oder Relabeln eines Descriptors zwar den Descriptor verändert, nicht aber eine Vertex-Neuverknüpfung für dessen Tree Vertices darstellt.

4.4.3 Weitere Vertex-Transformationen

Um alle restlichen Vertex-Transformationen zu bestimmen, traversieren wir T mittels Tiefensuche und bestimmen dann für den jeweils aktuell betrachteten Tree Vertex vT mittels des in Abbildung 5 skizzierten Vorgehens dessen Entsprechung vS in S.

Es ist dabei wichtig, dass der Baum in einer solchen Reihenfolge durchlaufen wird, die garantiert, dass alle Vorfahren der aktuellen Tree Vertex bereits durchlaufen wurden. Denn an verschiedenen Stellen wird vorausgesetzt, dass der Ursprung von dad(vT) bekannt und endgültig bestimmt ist. Offensichtlich erfüllt Tiefensuche dieses Kriterium.

Erläuterung von Abbildung 5

(1) Hat sich der Name vTs verändert? Wenn nicht, dann hat sich auch seine Position im MeSH-Tree nicht verändert. Denn im MeSH beschreibt der Name einer Tree Vertex gerade seine Position. Wenn dies zutrifft, ist also nichts zu tun.

(2) Es wird überprüft, ob dT, also der Descriptor vTs, unverändert aus S übernommen wurde.

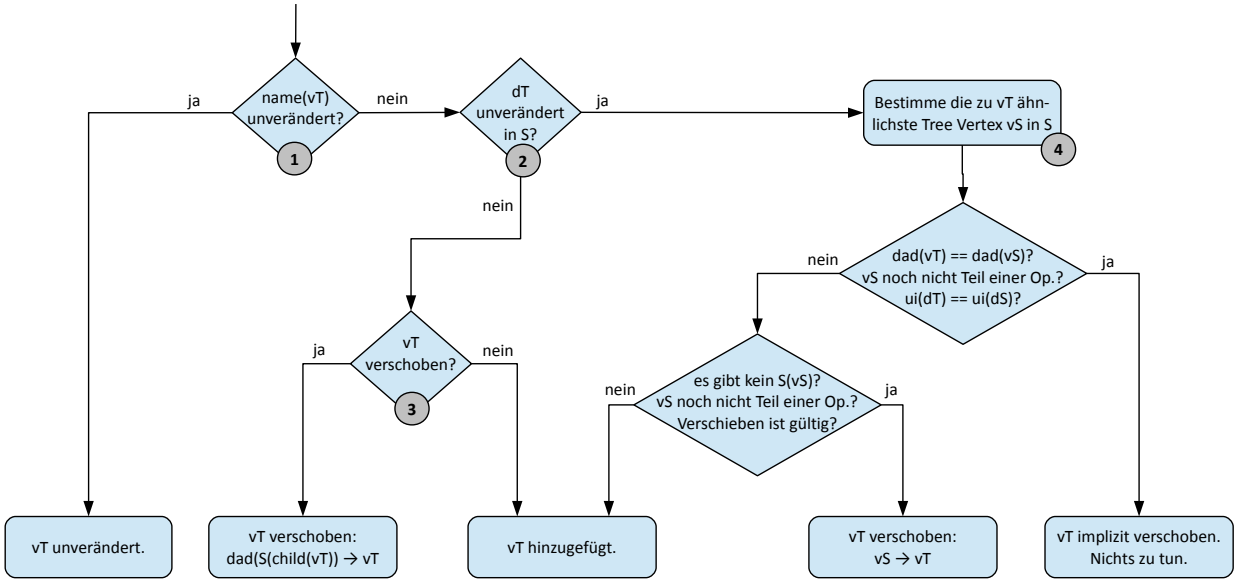

Abbildung 5: Bestimmen der Vertex-Transformationen

(3) Ist der Descriptor verändert worden, stehen wir vor einer schwierigen Aufgabe: Sowohl Descriptor als auch Tree Number vTs haben sich verändert. Dies sind aber die beiden wichtigsten Informationen, um herauszufinden, ob vT nun durch Verschieben oder Hinzufügen in T vorhanden ist. Es bleibt nur die strukturelle Information, d. h. wir analysieren die Kinder und Eltern vTs und versuchen daraus dessen Ursprung abzuleiten. Dabei gehen wir nach folgendem Prinzip vor:

> Angenommen die Kinder vTs sind unverändert (d. h. nur implizit verschoben) und ein solches Kind sei cT.
>
> Wir bestimmen den Vater cSs, also den Vater der Entsprechung cTs in S, als die Tree Vertex des Descriptors cSs, die vT am ähnlichsten ist. Als Ähnlichkeitsmaß verwenden wir das in 21 *Ähnlichkeit* beschriebene.
>
> Wenn nun vT auch in S der Vater cSs gewesen sein sollte, wenn vT also von dort zu seiner aktuellen Position verschoben wurde, dann müssten die $\mathsf{dad(cS)}$ für alle Kinder cS von vT übereinstimmen. Sollten die Eltern der Kinder in S jedoch nicht in ausreichendem Maß übereinstimmen, dann ist vT zu T hinzugefügt wurden.

(4) Es existiert dT also unverändert in S als dS. Wir wollen nun den wahrscheinlichsten Ursprung vTs in S, also vS, bestimmen. Da dT unverändert ist, muss vS eine der Tree Vertices dSs sein. Wir wählen als vS jene Tree Vertex dSs die vT am ähnlichsten ist. Als Ähnlichkeitsmaß verwenden wir das in 21 *Ähnlichkeit* beschriebene.

Es sind nun drei Fälle zu unterschieden:

1. vT wurde implizit verschoben, das heißt sein Name (Tree Number) hat sich nur verändert, weil einer seiner Vorfahren verschoben wurde.
2. vT wurde explizit verschoben.
3. vT ist eine neue Tree Vertex eines existierenden Descriptor.

Fall 1 (vT wurde implizit verschoben) tritt ein, falls folgende Bedingungen erfüllt sind:

1. $\mathsf{dad(vT)} == \mathsf{dad(vS)}$, d. h. vT s Vater ist gleich geblieben.
2. $\mathsf{ui(dT)} == \mathsf{ui(dS)}$, d. h. vT s Descriptor ist gleich geblieben.
3. vS ist nicht bereits Teil einer zuvor bestimmten Transformation.

Fall 2 (vT wurde explizit verschoben) tritt ein, falls gilt:

1. vT wurde nicht implizit verschoben.
2. In T existiert keine Tree Vertex mit gleichem Namen wie vS, d. h. der zuvor bestimmte Ursprung vT s ist überhaupt ein möglicher Ursprung.
3. vS ist nicht bereits Teil einer zuvor bestimmten Transformation.
4. vS ist kein Vorfahre der Entsprechung $\mathsf{dad(vT)}$ s in S, d. h. vS wird nicht nach einem Nachkommen seiner selbst verschoben. Denn das wäre keine gültige Transformation.

Fall 3 (vT wurde hinzugefügt) tritt ein, falls Fall 1 und Fall 2 nicht eingetreten sind.

4.4.4 Tree-Vertex-Löschungen

Um alle Tree-Vertex-Löschungen zu finden, gehen wir wie folgt vor:

Wenn ein Vertex v aus S nicht mehr unverändert in T vorkommt, und außerdem keine Transformation v in eine Tree Vertex in T überführt, dann muss v aus S gelöscht worden sein.

Aus diesem Vorgehen folgt unmittelbar, dass alle anderen Transformationen vor den Löschungen bestimmt werden müssen.

4.4.5 Descriptor-Löschungen

Das Prinzip ist analog zu dem in Unterunterabschnitt 4.4.4:

Wenn ein Descriptor d aus S nicht mehr unverändert in T vorkommt, und außerdem keine Transformation d in einen Descriptor in T überführt, dann muss d aus S gelöscht worden sein.

4.5 Bewertung des Algorithmus

Mit der hier dargestellten Vorgehensweise wird eine Transformationsfolge O gefunden, die S in T überführt, denn:

- jeder Descriptor d bzw. jede Tree Vertex v in T wird genau einmal durchlaufen.
- bei einem solchen Durchlaufen wird
  - immer genau eine Transformation bestimmt, welche zum Auftreten von d bzw. v in T führt.
  - keine zuvor gefundene Transformation gelöscht oder verändert.
  - keine Transformation bestimmt, die in Widerspruch mit einer zuvor bestimmten Transformation stehen würde, d. h. O(S) ist zu jedem Zeitpunkt des Algorithmus' definiert.
- jeder Descriptor d bzw. jede Tree Vertex v in S, die nicht durch eine zuvor ermittelte Transformation nach T übernommen wird, wird als gelöscht erkannt.

Der Algorithmus garantiert keine minimal lange Transformationsfolge, produziert aber kurze Transformationsfolgen:

- Für einen Descriptor bzw. eine Tree Vertex in S welcher bzw. welche unverändert in T vorkommt, enthält O keine Transformation.
- Für jeden Descriptor bzw. jede Tree Vertex in T welcher bzw. welche nicht unverändert in S vorkommt, enthält O höchstens eine Transformation.
- Für jeden Descriptor bzw. jede Tree Vertex in S welcher bzw. welche bei der Überführung nach T gelöscht wird, enthält O höchstens eine Transformation.

Die Darstellungen in Unterunterabschnitt 3.3.2 *Kommutativität von Kompositionen der Transformationen* motivieren die sequentielle Abarbeitung der Transformationstypen. Denn wenn eine minimal lange Transformationsfolge kommutativ ist, dann ist die Reihenfolge der Bestimmung der Transformationen der Folge beliebig.

5 Aktualisierung von Transformationsfolgen

Dieser Abschnitt behandelt das Aktualisieren von Transformationsfolgen, welches aufgrund der jährlichen Änderungen im MeSH notwendig ist, wie in 3.1.3 Jährliche Änderungen beschrieben.

5.1 Problembeschreibung

Das Aktualisierungsproblem ist in Abbildung 6 dargestellt. Es folgt eine formalere Beschreibung:

> Seien S und T zwei MeSH-Graphen.
>
> Sei O eine Folge von Transformationen, so dass $\mathsf{O}(\mathsf{S}) = \mathsf{S}'$.
>
> Gesucht ist eine Folge von Transformationen O', so dass $\mathsf{O}'(\mathsf{T}) = \mathsf{T}'$. Dabei soll das semantische Ergebnis der Anwendung von O auf S und O' auf T möglichst ähnlich sein.

Der Semedico-MeSH wurde nach den Erfordernissen von Semedico erstellt, wurde in diesem Sinne also nach semantischen Kriterien aus der Gesamtheit des MeSH ausgewählt. Man hat bestimmte Teile des MeSH so ausgewählt und neu angeordnet, wie es für Semedico aus inhaltlichen Gründen sinnvoll ist. Das Ziel ist es nun diese Semantik der Auswahl über die jährlichen Veränderungen des MeSHs hinaus zu erhalten – allerdings ausschließlich auf Basis von strukturellen Veränderungen. Es findet keine inhaltliche Analyse der Descriptors oder Tree Vertices, auf die die verschiedenen Transformationen angewandt werden, statt.

Der Begriff der Ähnlichkeit, wie er oben in der Definition verwendet wurde, wird nachfolgend nur indirekt genauer charakterisiert, nämlich durch das in den folgenden Abschnitten beschriebene Vorgehen zur Aktualisierung der Transformationen. Dieses Vorgehen ist gerade so gewählt, dass eine aktualisierte Transformation möglichst ähnlich zur Ausgangsoperation ist.

Dieses Kapitel der Studienarbeit ist damit weniger als wissenschaftliche Arbeit zu sehen. Es ist allein praktisch motiviert, da ein Tool mit eben dieser Funktionalität gebraucht wird, selbst wenn es nur oberflächlich bewertet werden kann: zum einen anhand der Sinnhaftigkeit des Vorgehens und zum anderen anhand einer manuellen, qualitativen Überprüfung des Ergebnisses des Tools.

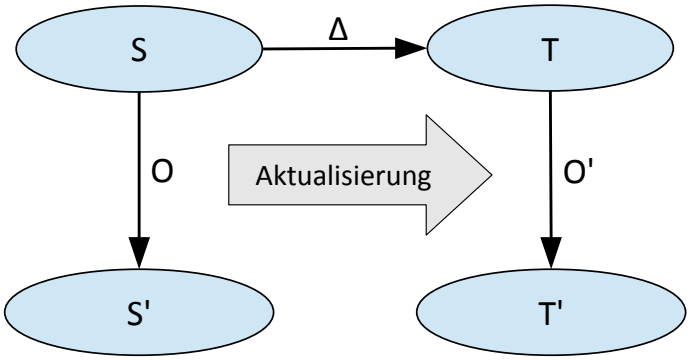

Abbildung 6: Aktualisierung der Transformationsfolge O für einen Ziel-MeSH-Graphen T

5.2 Grundidee

Das Aktualisieren der Transformationen läuft in folgenden Schritten ab:

1. Bestimmen einer Transformationsfolge Δ, welche S in T überführt
2. Aktualisieren der Transformationen in O, als O':
   a) Descriptor-Additionen
   b) Descriptor-Umbenennungen
   c) Descriptor-Relabellings
   d) Vertex-Additionen
   e) Vertex-Verschiebungen
   f) Vertex-Löschungen
   g) Descriptor-Löschungen
   h) Vertex-Umbenennungen

Zuerst bestimmen wir Δ. Diese Informationen über die Änderungen, welche S in dessen neue Version T überführen, sind die Grundlage um alle Transformationen aus O zu aktualisieren.

Danach überführen wir alle Transformationen aus O in ihre für T angepasste Form in O'. Die prinzipielle Vorgehensweise ist dabei immer gleich: es wird über alle Transformationen eines Typs iteriert und dabei, für jede Transformation, die in den jeweilig folgenden Unterabschnitten beschriebe Heuristik angewandt. Die so aktualisierte Transformation wird dann O' hinzugefügt.

Die oben gegebene Reihenfolge ist bis auf eine Ausnahme beliebig: Descriptor-Additionen müssen vor Vertex-Additionen aktualisiert werden, da die Aktualisierung der Descriptor-Additionen eine Veränderung der Vertex-Additionen bewirken kann. Die restliche Reihenfolge ist beliebig, da unser konkretes Vorgehen für die verschiedenen Transformationstypen keine wechselseitigen Abhängigkeiten enthält.

5.3 Vorgehen für Aktualisierung einer Transformation

Die Aktualisierung einer Transformation op läuft immer nach dem selben Prinzip ab:

Man überprüft nacheinander für die einzelnen Parameter ops, ob diese auch in T vorhanden bzw. gültig sind. Möchte man beispielsweise eine Tree Vertex verschieben, so muss die zu verschiebende Tree Vertex in T überhaupt vorhanden sein.

Ist sie das nicht, wird überprüft, ob eine der Transformationen in Δ dafür verantwortlich ist. Zum Beispiel könnte die Tree Vertex verschoben oder gelöscht worden sein. Wenn keine passende Transformation aus Δ gefunden werden kann, wird mit einer Fehlermeldung beendet. Dieser Fall sollte nicht auftreten, da alle Veränderungen an S von Δ erfasst sein sollten. Folglich ist entweder Δ inkorrekt bestimmt worden, oder S oder T sind nach dem Berechnen von Δ verändern worden.

Wenn eine passende Transformation gefunden wurde, ist es unter Umständen möglich das Problem zu beheben. Beispielsweise verschiebt man statt der ursprünglichen, nicht mehr vorhandenen Vertex, die verschobene Vertex. Wenn kein Fix möglich ist, wird eine Fehlermeldung ausgegeben und op nicht zu O′ hinzugefügt.

Da es erstens bei mehr als einem Parameter zu Problemen kommen kann, und zweitens auch ein Fix nicht zwangsweise zu einem gültigen Parameter führt, wiederholt man die Aktualisierung solange, bis entweder ein Fehler auftritt, der nicht gefixt werden kann, oder aber die Transformation durch die angewendeten Fixes gültig geworden ist. In letzterem Fall kann die Transformation nach O′ übernommen werden.

Bezeichnungen

Es folgen Erläuterungen zu den Bezeichnungen in diesem Abschnitt.

dad(v)	Vater der Tree Vertex v
name(d)	Name eines Descriptors d
ui(d)	UI eines Descriptors d
S(ui)	Descriptor in S der UI ui besitzt
S(dname)	Descriptor in S der Name dname besitzt
S(vname)	Tree Vertex in S die Name vname besitzt

In den Grafiken sollen die verschiedenen Farben zum Verständnis beitragen:

gelb	Einstiegspunkt. Zeigt die zu aktualisierende Transformation mit Parametern.
blauer Kreis mit x	Überprüfung eines Parameters x ob dieser angepasst werden muss.
hellblau	„Normale“ Bedingungen und Anweisungen.
grün	Erfolgreicher Abschluss der Aktualisierung.
rot	Fehler.

5.3.1 Descriptor-Additionen

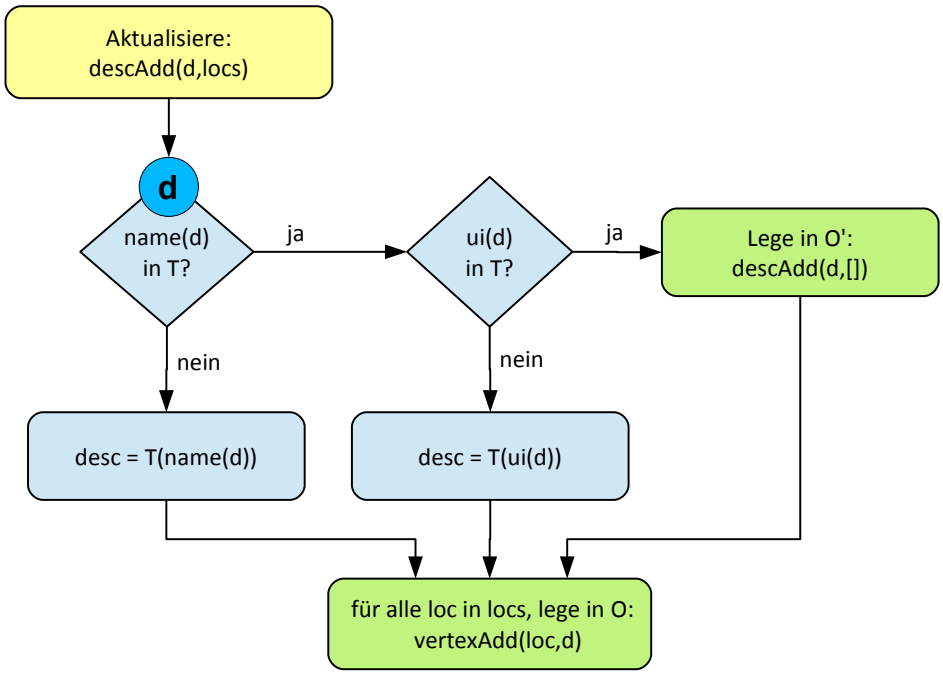

Abbildung 7: Aktualisierung von Descriptor-Additionen

Bei der Aktualisierung des Hinzufügen eines Descriptors werden immer alle Tree Vertices als separate Tree-Vertex-Additionen behandelt und diese zu O (nicht O'!) hinzugefügt. Dadurch werden die Tree-Vertex-Additionen später auf mögliche Probleme überprüft.

Übrig bleibt das Hinzufügen des Descriptors selbst. Wenn dessen Name oder UI bereits in T existieren, wird dieser nicht hinzugefügt. Als Descriptor für die hinzuzufügenden Tree Vertices wird dann der Descriptor des bereits existierenden Namens bzw. der bereits existierenden UI verwandt.

Wenn weder UI noch Name in T existieren, kann die Descriptor-Addition nach O' übernommen werden.

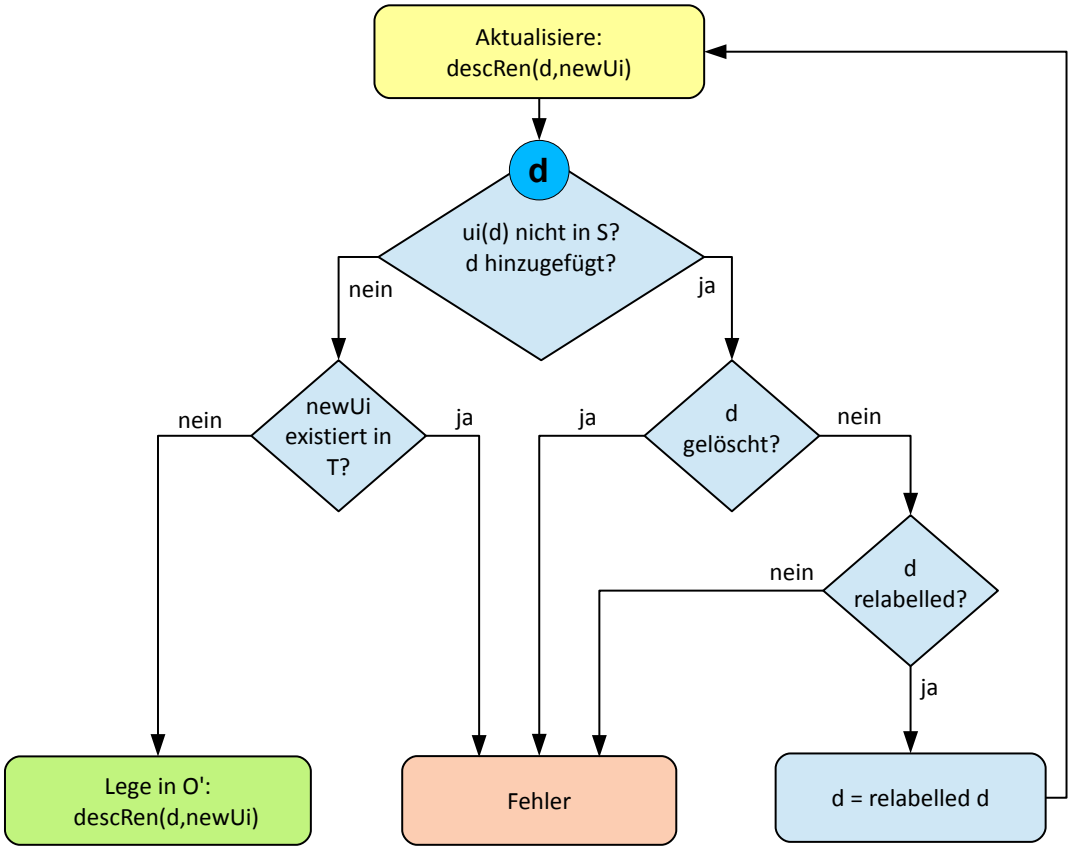

Abbildung 8: Aktualisierung von Descriptor-Umbenennungen

5.3.2 Descriptor-Umbenennungen

Siehe Abbildung 8.

Falls Descriptor desc in T bereits existiert, dann kann es zwei logische Ursachen geben:

1. Der Descriptor wurde gelöscht. Dann ist eine Aktualisierung der Transformation unmöglich.
2. Der Descriptor hat eine andere UI bekommen. Dann verwenden wir diese veränderte UI zur abermaligen Aktualisierung der Transformation.

   Ansonsten wird ein Fehler gemeldet.

Auch wenn desc in T nicht vorhanden ist, kann ein Problem auftreten. Nämlich, wenn die neue UI des Descriptors in T bereits vorhanden ist. In diesem Fall wird ebenfalls eine Fehlermeldung ausgegeben, da hier nicht klar ist, was getan werden sollte.

5.3.3 Vertex-Additionen

Siehe Abbildung 9.

Falls der Descriptor, zu dem eine Vertex hinzugefügt werden soll, umbenannt wurde, dann wird versucht die Vertex zu dem umbenannten Descriptor hinzuzufügen.

Wurde der Descriptor hingegen gelöscht, ist kein Weiterkommen und ein Fehler wird gemeldet.

Kann der p nicht gefunden werden, weil es gelöscht wurde, so wird stattdessen versucht v als Kind von dad(p) einzufügen.

Wenn p umbenannt wurde, fügen wir v als Kind dieser umbenannten Tree Vertex ein.

5.3.4 Vertex-Verschiebungen

Da vertexMove viele Parameter besitzt und dadurch die Grafik zu groß werden würde, ist sie in eine Übersichtsgrafik (Abbildung 10) und 4 Teilgrafiken (Abbildung 11 bis Abbildung 14) aufgeteilt worden: eine Grafik für die Problembehandlung eines jeden Parameters.

Die Abfragen „Aktualisierung durch x?“ in Abbildung 10 entsprechen dem Test, ob ein Parameter x in T nicht vorhanden ist und auch nicht durch eine Transformation aus Δ hinzugefügt wurde.

Die Grafiken Abbildung 11, Abbildung 12, Abbildung 13 und Abbildung 14 sind weitgehend selbsterklärend. Daher folgen nur Erläuterungen zu den Schlüsselstellen.

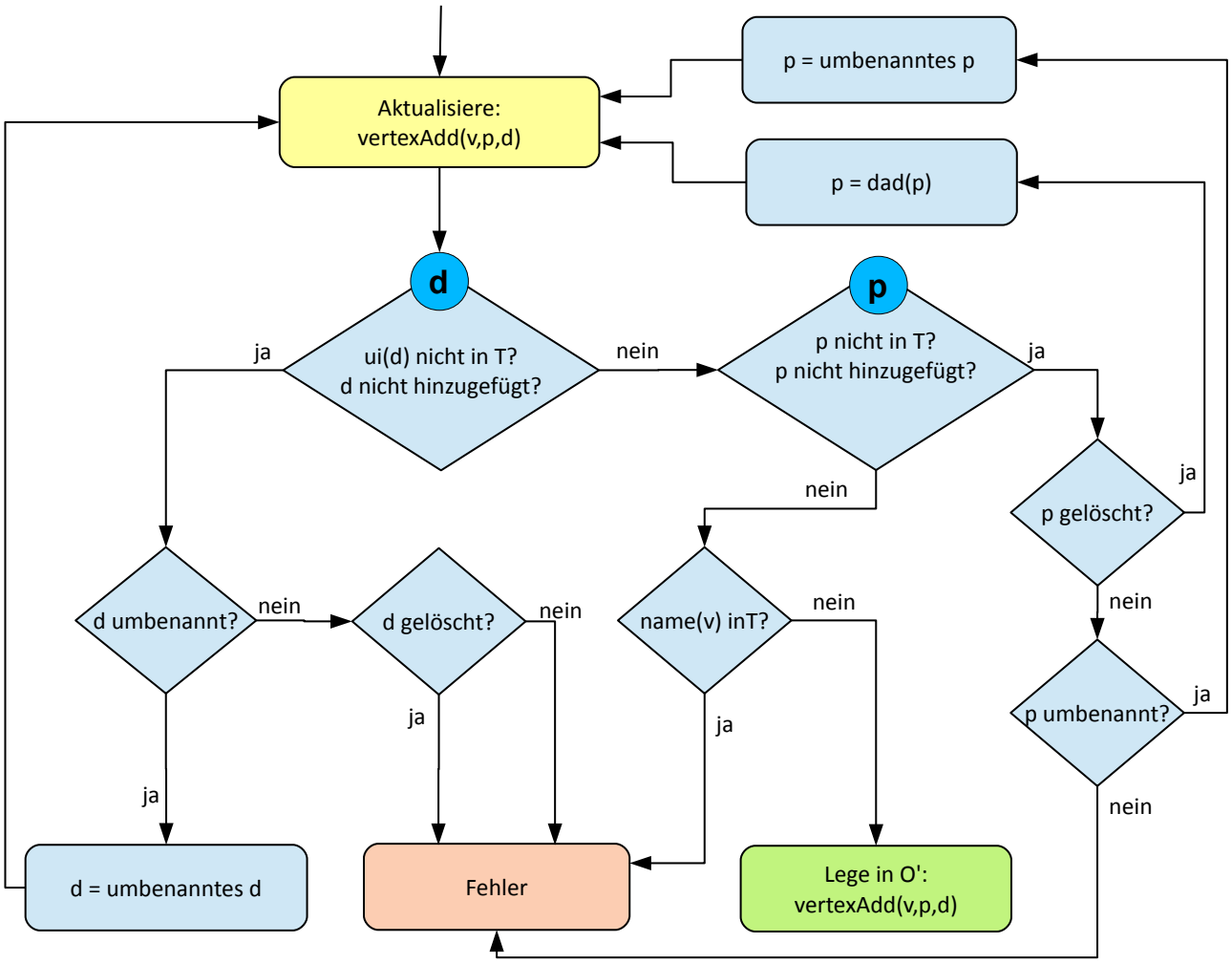

Abbildung 9: Aktualisierung von Vertex-Additionen

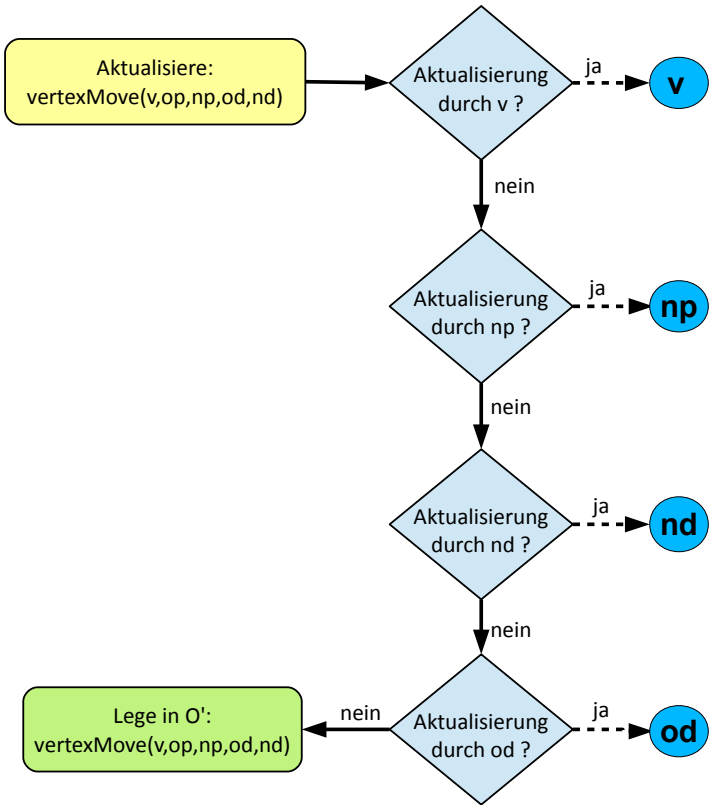

Abbildung 10: Aktualisierung von Vertex-Verschiebungen - Übersicht

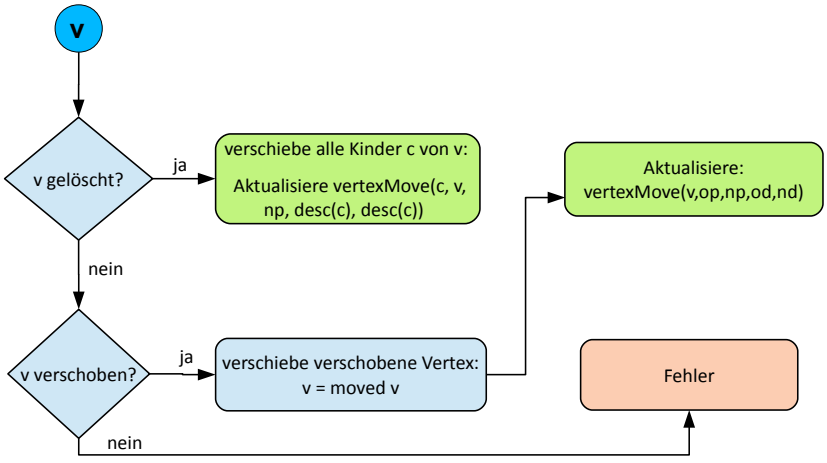

Abbildung 11: Vertex-Verschiebungen - Veränderungen an v

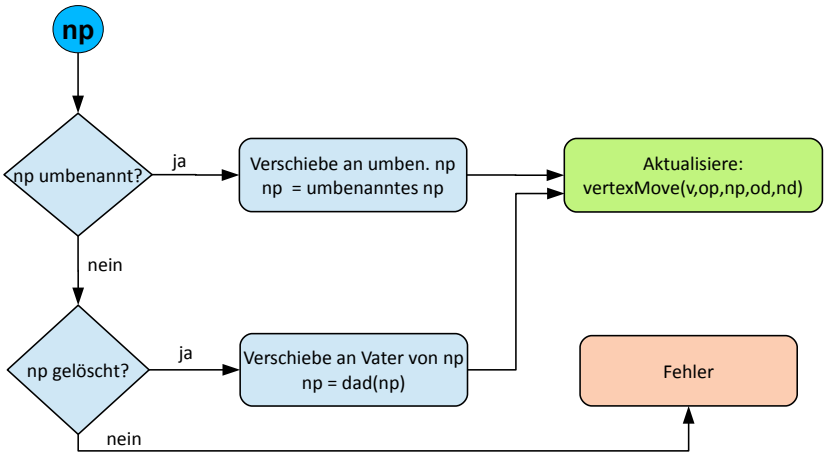

Abbildung 12: Vertex-Verschiebungen - Veränderungen an np

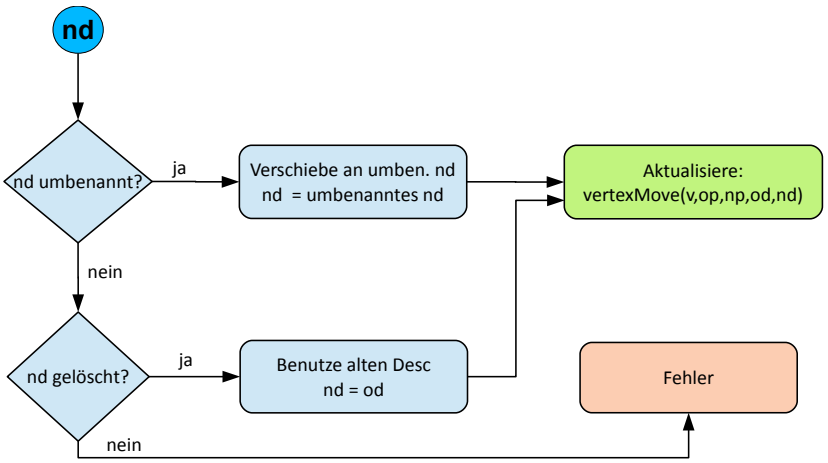

Abbildung 13: Vertex-Verschiebungen - Veränderungen an nd

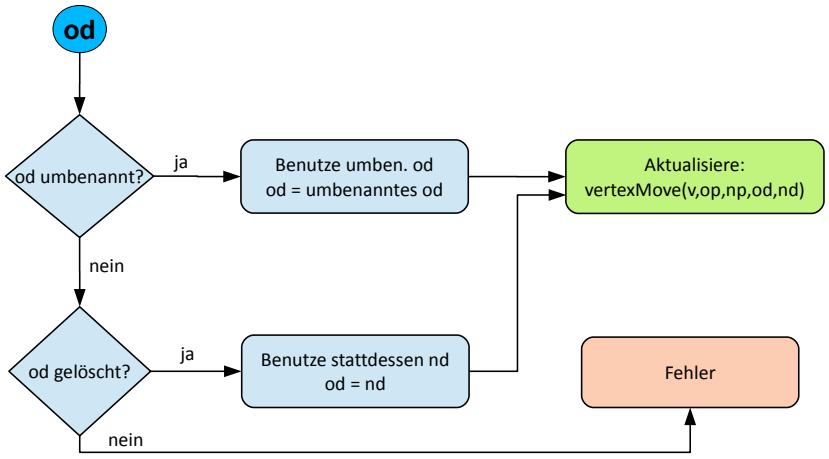

Abbildung 14: Vertex-Verschiebungen - Veränderungen an od

Abbildung 13: Falls der neue Descriptor nd aus T gelöscht wurde, wird als Fix stattdessen der alte Descriptor od verwendet. Dies ist sinnvoll, da das explizite Verschieben v s impliziert, dass v für Semedico wichtig ist. Anstatt hier also mit einer Fehlermeldung abzubrechen, wird versucht die Transformation zu erhalten.

Abbildung 13: Falls der alte Descriptor od aus T gelöscht wurde, wird als Fix stattdessen der neue Descriptor od verwendet. od wird nur benötigt, um od, im Falle einer Vertex-Neuverknüpfung v s, zu informieren, dass od eine Tree Vertex verloren hat. Da od aber entfernt wurde, wird diese Information nicht länger benötigt und es ist korrekt od auf nd zu setzen.

5.3.5 Vertex-Löschungen

Siehe Abbildung 15.

Das Vorgehen ist eingängig: ist v unverändert vorhanden, kann die Vertex-Löschung direkt übernommen werden. Wurde v gelöscht, muss entweder nichts mehr getan werden, falls die Löschung nicht rekursiv war, oder es wird als Fix versucht alle Kinder v s zu löschen. Wurde v hingegen verschoben, so wird versucht verschobene Tree Vertex zu löschen.

5.3.6 Descriptor-Löschungen

Siehe Abbildung 16.

Das Vorgehen hier ist weitgehend analog zu dem für Vertex-Löschungen.

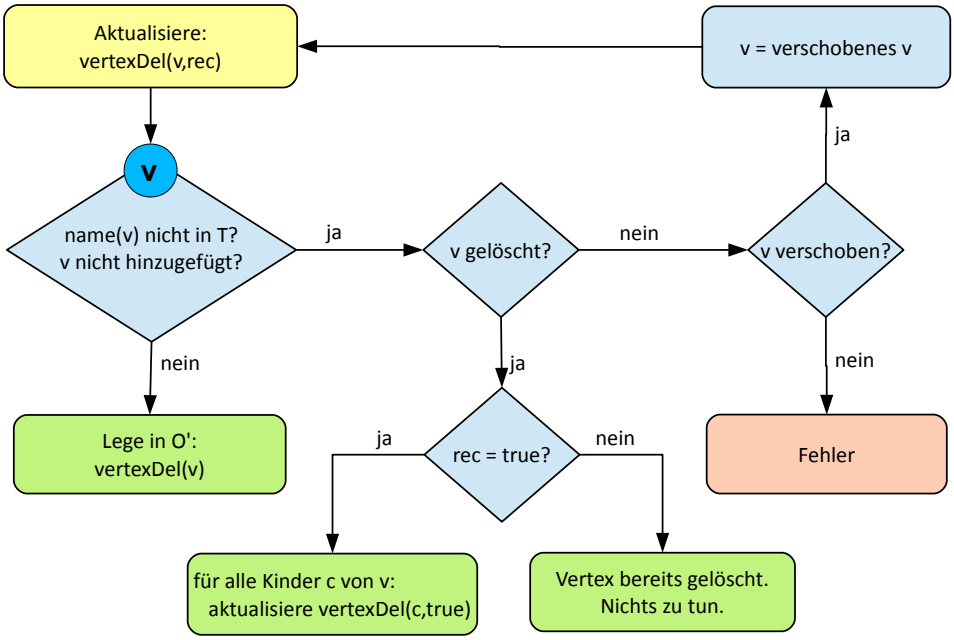

Abbildung 15: Aktualisierung von Vertex-Löschungen

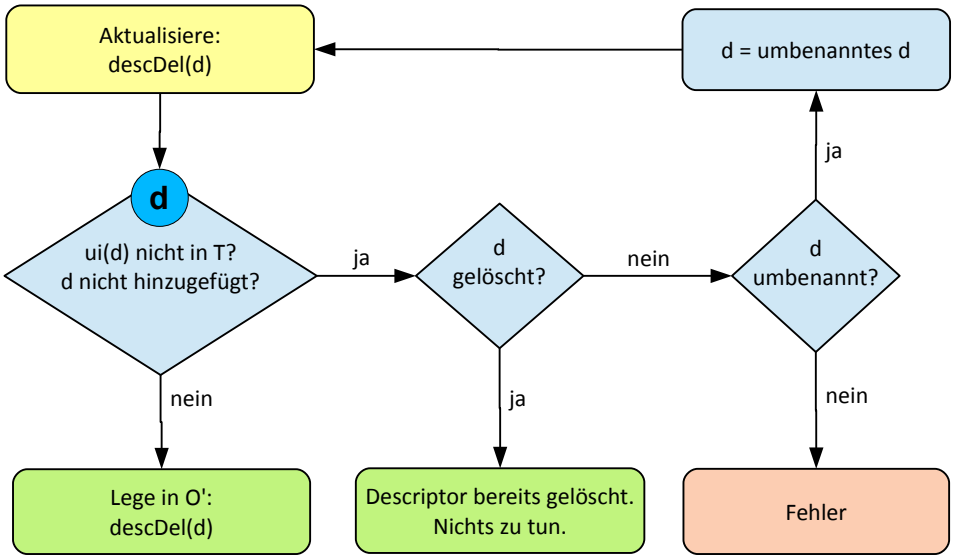

Abbildung 16: Aktualisierung von Descriptor-Löschungen

5.3.7 Descriptor-Relabellings und Vertex-Umbenennungen

Descriptor-Relabellings und Vertex-Umbenennungen treten in diesem Anwendungsfall nicht auf, da keine beim Vergleich des MeSH 2008 und des Semedico-MeSH 2008 bestimmt werden und auch nicht durch das Aktualisieren von Transformationen entstehen können. Daher wurden die beiden Transformationstypen noch nicht umgesetzt.

6 Umsetzung

Nachdem nun die Grundlagen und das prinzipielle Vorgehen zur Lösung klar sind, folgt hier eine kurze Darstellung der entwickelten Software. Zusätzlich findet sich eine ausführliche Dokumentation als Kommentare im Quelltext sowie in den Java-Doc-Pages. Bei Detailfragen sei darauf verwiesen.

6.1 Wesentlichen Eigenschaften

Folgende Richtlinien wurden bei dem Design und der Entwicklung der Software beachtet:

fail-early Die Software soll so entwickelt werden, dass die Parameter der Methoden unmittelbar auf ihren Gültigkeitsbereich überprüft werden, so dass Fehler frühzeitig erkannt und auch dem Nutzer mitgeteilt werden können.

report-early Die Software soll so entwickelt werden, dass sie den Nutzer möglichst gut Auskunft darüber gibt bzw. geben kann, was sie gerade tut.

Bezeichner Die Software soll so entwickelt werden, dass die verwendeten Bezeichner für Variablen, Methoden, Klassen und Paketen einheitlich und unmittelbar einsichtig sind, und Auskunft über die Funktion bzw. Bedeutung geben. Um lange Bezeichner zu vermeiden, werden eingängige und einheitliche Abkürzungen verwendet.

Mehr Kommentare sind besser als weniger Die Software soll möglichst umfangreich und mit Hilfe von Java-Doc-Tags kommentiert werden. Das heißt, Klassen und Methoden, die sich nicht unmittelbar selbst erklären, sollen eine Beschreibung ihrer Funktion, Parameter und Besonderheiten enthalten. Zudem soll auch die interne Struktur und Funktionsweise mit Hilfe von Kommentaren begleitend erläutert werden.

Frühes Abstrahieren und Kapselung Das Design der Software soll so geschehen, dass ein möglichst hoher Grad an Abstraktion und Kapselung erreicht wird. Das macht die Software leichter verständlich und nutzbar, erweiterbarer, wiederverwendbarer und weniger fehleranfällig.

Sprache Die Sprache des Quellcodes und der Kommentare im Quellcode ist Englisch.

Dem Design lagen folgende Entscheidungen zugrunde:

Programmiersprache Da der Großteil des Semedico Projekts in Java entwickelt wurde, fiel die Wahl auch für diese Studienarbeit automatisch auf Java.

Entwicklungsumgebung Als Entwicklungsumgebung wird Eclipse verwendet. Dies geschieht aus zwei Gründen: Erstens habe ich bereits Erfahrungen mit Eclipse und zweitens wird auch zur Entwicklung von Semedico Eclipse verwendet.

Versionsmanagement Da für das Semedico Projekt bereits ein SVN-Repository existiert, wurde diese, inklusive des Maven-Build-Managements, übernommen.

XML-Parser Die XML-Daten des MeSH haben eine ungefähre Größe von 300 MB. Daher ist es eminent wichtig einen XML-Parser zu verwenden, der erstens mit Dateien dieser Größe umgehen kann und zweitens diese auch in kurzer Zeit verarbeiten kann.

Der klassische XML-Parser JDOM hat sich bei einem kurzem Test als vollkommen unbrauchbar herausgestellt. JDOM muss jederzeit ein komplettes Abbild der XML-Daten im Speicher halten und wird damit bei diesen Datenmengen extrem langsam bzw. stürzt ab.

Die Wahl fiel daher auf die Simple API for XML (SAX) in Form des Java-SAX-Parser, der über die Pakete org.xml.sax.* zur Verfügung gestellt wird. Dieser arbeitet sequentiell und push-event-basiert. Dadurch ist SAX deutlich schneller und minimiert den Speicherbedarf.

Eine mögliche Alternative ist die Streaming API for XML (StAX). Jedoch erscheint die strikt sequentielle Verarbeitungsweise von SAX hier passender und für diese Anwendung effizienter und leichter umsetzbar.

Graphenbibliothek Als Graphenbibliothek wird die etablierte JGraphT-Bibliothek verwendet.

6.2 Design

Abbildung 17 zeigt einen Überblick zur Struktur und dem Zusammenspiel der wichtigsten Klassen und Packages. Die Darstellung ist an UML-Klassen-Diagramme angelehnt.

Im Zentrum steht die Tree-Klasse. Sie repräsentiert einen MeSH-Tree.

Der Import und Export von Modifikationen und Daten für Tree wird von Klassen im exchange-Package übernommen.

Unterschiede zwischen zwei MeSH-Trees können über eine Instanz der Klasse TreeComparator bestimmt werden. Diese implementieren die Algorithmen aus 4 Transformationsrekonstruktion bei MeSH-Graphen.

Eine Aktualisierung von Modifikationen, wie in 5 Aktualisierung von Transformationsfolgen beschrieben, kann durch eine Instanz der Klasse TreeModificationMerger berechnet werden.

Instanzen der Klasse TreeModificator erlauben es Instanzen von Tree zu verändern.

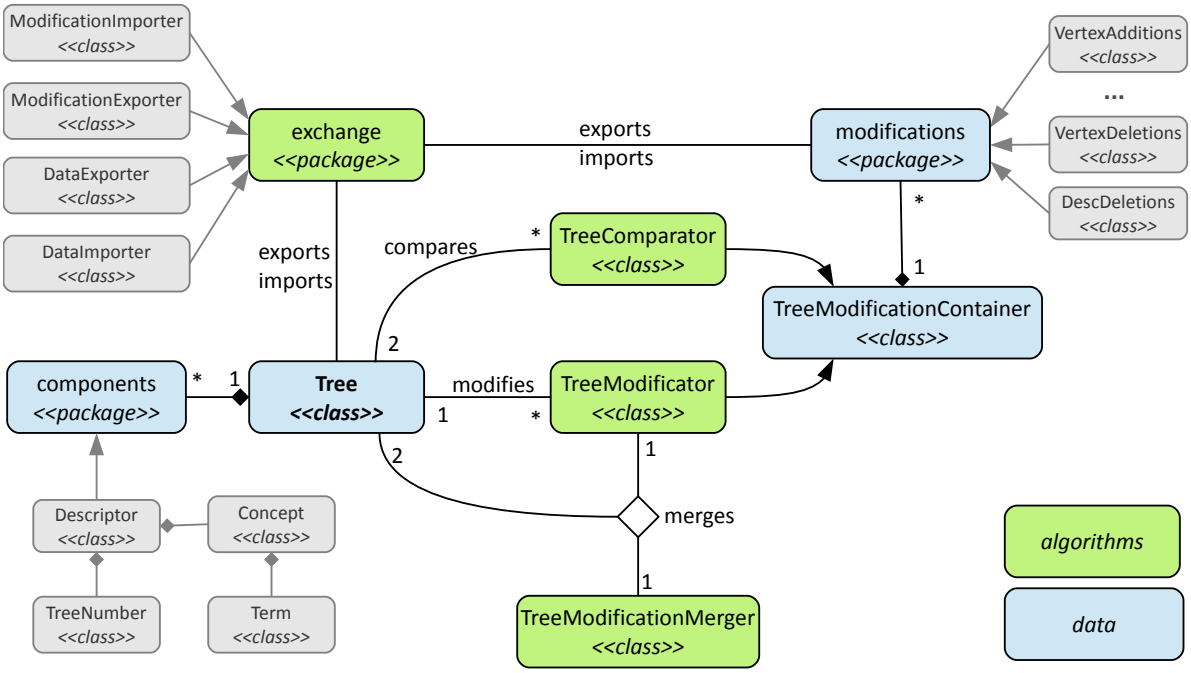

Abbildung 17: Überblick zur Struktur der Software

TreeComparator und TreeModificator leiten von der selben Basisklasse TreeModificationContainer ab, da beide ein Menge von Modifikationen verwalten. Deshalb kann das Ergebnis eines Vergleichs zweier Bäume, also ein TreeComparator-Objekt, leicht in eine Instanz von TreeModificator überführt werden, und umgekehrt.

Die möglichen Modifikationen entsprechen den in 3.3.1 Operationen und Transformationen aufgelisteten Operationen.

6.3 Package-Übersicht

Die Software ist in eine Reihe von Packages aufgeteilt, die jeweils für sich eine funktionale Einheit bilden. Eine Übersicht bietet Tabelle 1.

6.4 Klassen-Übersicht

In Tabelle 2 sind die wichtigsten Klassen zusammen mit ihrer Funktion aufgeführt.

Package-Name	Erläuterung
Basis-Package	Enthält alle andere Pakete sowie die wesentlichen Klassen der Software. Dazu gehören insbesondere die Klassen Process, Tree, TreeComparator, TreeModificator und TreeModificationMerger.
components	Beschreibt die wesentlichen MeSH-XML-Elemente, wie Descriptor, Concept oder Term, als einzelne Klassen.
exchange	Für Import bzw. Export der MeSH-Daten und Modifikationen. Verschiedene Formate werden unterstützt.
modifications	Beschreibt Modifikationen die auf MeSH-Trees angewandt werden können, wie DescAdditions oder VertexDeletions.
testing	JUnit-Testing-Klassen.
tools	Verschiedene Hilfsklassen und Methoden.

Tabelle 1: Package-Übersicht

6.5 Interessante Details

XML-Formate

Es werden im Moment drei XML-Formate unterstützt:

UD-XML Der bisher zu Erstellung des Semedico-MeSH verwandte Algorithmus speichert seine Ergebnisse in einem eigenen XML-Format. Dieses Format wird hier als UD-XML bezeichnet. Die Klasse Parser4UserDefMeSH implementiert einen SAX-XML-Handler zum Einlesen. Ein Schreiben ist nicht möglich.

MeSH-XML Dies ist das offizielle und bereits in 3.1.2 MeSH Komponenten und XML beschrieben XML-Format des MeSH. Die Klasse Parser4MeSH implementiert den entsprechenden SAX-XML-Handler zum Einlesen. Ein Schreiben ist nicht möglich. Stattdessen kann dazu das Own-XML-Format verwendet werden.

Own-XML Eigenes XML-Format, das eine leicht veränderte Teilmenge des MeSH-XML-Formats darstellt:

Nur die XML-Elemente, welche auch in Tree-Objekten repräsentiert werden, wurden übernommen.

Klassenname	Erläuterung
Process	Beispielanwendung, welche die entwickelte Bibliothek nutzt.
Tree	Repräsentation eines MeSH-Trees. Tree stellt elementare Methoden zum Modifizieren und Abfragen eines MeSH-Trees zur Verfügung. Dies umfasst beispielsweise Tree-Vertices löschen, verschieben oder hinzufügen.
TreeComparator	Abstrakte Basisklasse um zwei MeSH-Trees zu vergleichen. Bestimmt dabei die Modifikationen, die den einen MeSH-Tree in den anderen überführen.
TreeComparatorMeSH	Ableitung der Klasse TreeComparator. Vergleicht zwei beliebige Tree-Instanzen. Implementiert die Methoden aus 4 Transformationsrekonstruktion bei MeSH-Graphen.
TreeComparatorUD	Ableitung der Klasse TreeComparator. Vergleich eine beliebige Tree-Instanz mit einer Tree-Instanz zu vergleichen, welche aus UD-XML-Daten (siehe Unterabschnitt 6.5) erstellt wurde.
TreeFilter	Zum Filtern von unerwünschten Tree Vertices oder Descriptors aus einem Tree-Objekt. Existiert unabhängig von den anderen Klassen zum Modifizieren.
TreeModificationContainer	Container für Tree-Modifikationen. Basisklasse für TreeModificator und TreeComparator.
TreeModificationMerger	Zum Aktualisieren einer Instanz von TreeModificationContainer für ein verändertes Tree-Objekt. Implementierung der Methoden aus 5 Aktualisierung von Transformationsfolgen.
TreeModificator	Ableitung der Klasse TreeModificationContainer. Ermöglicht die Anwendung der enthaltenen Modifikationen auf ein Tree-Objekt.
DataExporter	Exportieren eines Tree-Objekts in verschiedene Formate. Neben dem Own-XML-Format (siehe Unterabschnitt 6.5), wird auch ein direktes Exportieren in das Semedico-Datenbankmanagementsystem (DBMS) unterstützt.
Parser4MeSH	SAX-XML-Handler um MeSH-XML-Dokumente zu parsen.
Parser4OwnMeSH	SAX-XML-Handler um Dokumente mit einer MeSH-XML-ähnlichen Syntax zu parsen. Siehe auch Unterabschnitt 6.5.
Parser4UserDefMeSH	SAX-XML-Handler um XML-Dokumente zu parsen, die mit dem alten Algorithmus zum Erstellen des Semedico-MeSH erzeugt wurden.

Tabelle 2: Klassen-Übersicht

Statt des XML-Elements <TreeNumberList> samt <TreeNumber>s, gibt es nun ein XML-Element <LocationList> mit den Kindern <Location>, welche selbst aus <ParentVertexName> und <VertexName> bestehen. Der Sinn ist es, die Struktur des MeSH explizit als Relation festzuhalten. Siehe dazu auch in 3.1.2 den Unterpunkt Tree Number.

Die Klasse Parser4OwnMesh implementiert einen SAX-XML-Handler zum Einlesen. Schreiben ist über zwei überladene Methoden toOwnXml der Klasse DataExporter möglich.

Descriptor-Additionen in Tree-Objekte

Da die MeSH-XML-Daten als Menge von Descriptors organisiert sind, geschieht auch das Einlesen der XML-Daten als Iteration über alle <DescriptorRecord>-Elemente. Und weil Descriptors jeweils eine Menge aus Tree Vertices besitzen, und diese an beliebigen Stellen im Tree-Vertex-Baum verteilt sein können, sind wir mit einem Problem konfrontiert: Es müssen ggf. Tree-Vertices eingefügt werden, deren Väter noch nicht im MeSH-Tree existieren.

Um diese Problem zu lösen, verwaltet jedes Tree-Objekt eine pendingVertices-Map: Schlüssel der Map sind die noch fehlenden Vater-Tree-Vertices. Die Werte der Map sind jeweils eine Liste der Tree Vertices, die als Kinder des Schlüssels eingefügt werden sollen. Bei jedem Einfügen einer Tree Vertex wird dann pendingVertices abgefragt, um ggf. wartende Tree Vertices tatsächlich einzufügen.

Validierung von Tree-Objekten

Die Klasse Tree stellt elementare Methoden zum Verändern zur Verfügung. Allerdings ist es damit möglich ein Tree-Objekt so zu modifizieren, dass die Struktur seiner Tree Vertices nicht länger einen Baum darstellt. Geschieht das auf unkontrollierte Art und Weise und wir dann versucht mit dem Objekt zu arbeiten, ist das Verhalten nicht determiniert.

Daher wurde die Methode validataIntegrity() implementiert, welche überprüft ob ein Tree-Objekt eine gültige Struktur hat. Dies hat sich auch während der Entwicklung als sehr hilfreich herausgestellt, da durch kontinuierliches Verwenden dieser Methode viele Fehler sehr früh entdeckt werden konnten.

6.6 Testing

Zum Testen werden in erste Linie JUnit Tests verwendet, die als Klassen im Package testing implementiert sind.

Folgende Tests werden für die Klasse TreeComparatorMeSH betrachtet:

- ein Testfall für jeden Operationstyp, der anhand eines minimalen Beispiels überprüft, ob die zu erwartenden Modifikationen bestimmt wurden.
- ein komplexer Testfall der alle Operationstypen involviert, bei dem analog überprüft wird, ob die entsprechenden Modifikationen bestimmt wurden.
- ein Testfall, der die MeSH-Versionen der Jahre 2008 und 2009 vergleicht, dann die bestimmten Modifikationen auf den MeSH 2008 anwendet und im Anschluss die Gleichheit des resultierende MeSH-Trees und des MeSHs 2009 überprüft. Sind beide gleich, bestätigt dies auch für ein außerordentlich großes Beispiel die korrekte Funktionsweise des Algorithmus'.

Zum Testen der Klasse TreeComparatorUD wird genauso vorgegangen wie beim dritten Testfall für TreeComparatorMeSH.

Wünschenswert wäre ein automatisches Testen der Klasse TreeModificationMerger. Ziel dieser Klasse ist es eine Menge von Modifikationen so anzupassen, dass sie auf einen veränderten MeSH-Tree angewandt werden können - *und dabei die Semantik möglichst wenig zu ändern*. Weil sich eine Änderung der Semantik schwer in Zahlen fassen lässt, es vor allem schwer zu sagen ist, ob eine Aktualisierung „richtig" oder „falsch" ist, ist ein automatisches Testen mit adäquatem Aufwand nur eingeschränkt möglich. Stattdessen kann zumindest automatisiert überprüft werden ob das Ergebnis syntaktisch korrekt ist.

6.7 Benutzung

Ziel war es eine Bibliothek zu entwickeln, die alle zu Beginn aufgeführten Ziele erfüllt und dabei möglichst flexibel sowie einfach nutzbar ist. Der normale User braucht nicht und sollte nicht wissen, welche Vorgänge im Detail im Inneren Software ablaufen. Er sollte nur wissen müssen, welche Klassen und Methoden sein konkretes Problem lösen. Ein Großteil der Bibliothek wird daher vom Endnutzer nicht direkt genutzt und soll hier auch nicht aufgeführt werden. Die genauen Schnittstellen aller Methoden sind darüber hinaus mit Java-Doc-Tags dokumentiert.

Als Anwendungsbeispiel wird in Listing 2 der vollständige Quellcode zum Erstellen des Semedico-MeSH 2009 dargestellt. Nur die Aufrufe getProp(...) verweisen auf ein zuvor eingelesenes Konfigurationsfile, dass die entsprechenden Pfade einstellt.

```
private static void create2009SemedicoMesh() {
    // parse original mesh (as of 2008)
    Tree mesh2008 = new Tree("MeSH-2008");
    DataImporter.fromOriginalMeshXml(getProp("mesh2008FilePath"), getProp("
        importMeshXmlDtdFilePath"), mesh2008);

    // parse semedico mesh (as of 2008)
    Tree semedico2008 = new Tree("semedico-MeSH-2008");
    DataImporter.fromUserDefinedMeshXml(getProp("semedico2008FilePath"), "
        no-dtd",semedico2008);

    // determine modifications
    TreeComparatorUD mods4semedico2008 = new TreeComparatorUD(mesh2008,
        semedico2008);
    mods4semedico2008.determineModifications();

    // parse mesh (as of 2009)
    Tree mesh2009 = new Tree("MeSH-2009");
    DataImporter.fromOriginalMeshXml(getProp("mesh2009FilePath"), getProp("
        importMeshXmlDtdFilePath"), mesh2009);

    // update modifications
    TreeModificationMerger merger = new TreeModificationMerger(mesh2008,
        mesh2009, mods4semedico2008);
    TreeModificator mods4semedico2009 = merger.merge();

    // apply modifications on mesh 2009 -> create semedico mesh 2009
    mods4semedico2009.applyAll(true);

    // export modifications for semedico mesh 2009
    mods4semedico2009.saveModificationsToFiles(getProp("
        mods4semedico2009FilePath"));

    // export mesh 2009
    DataExporter.toOwnXml(mesh2009, getProp("semedico2009FilePath"));
}
```

Listing 2: Erstellen des Semedico-MeSH-2009

7 Auswertung

Eingangs wurden in Abschnitt 2 drei zu lösende Probleme besprochen. Diese sollen nun mit den erreichten Ergebnissen verglichen und bewertet werden.

Verwaltung der Wissensbasis

Die Software ist in der Lage den MeSH entsprechend der aktuellen Anforderungen Semedicos zu importieren, zu repräsentieren, zu modifizieren und zu exportieren. Insbesondere können die offiziellen MeSH-XML-Daten importiert und Tree-Objekte direkt in das Semedico-DMBS exportiert werden.

Weil Modifikationen der Tree-Instanzen als Objekte eigener Klassen dargestellt werden, ist es ebenfalls möglich, diese Modifikationen zu exportieren und zu importieren. Dadurch ist es dem Nutzer möglich kompakte, eigene Java-Programme zu schreiben, die typische Aufgaben der Wissensbasisverwaltung, wie das kontrollierte Zusammenfügen Daten verschiedener Quellen, vollautomatisiert übernehmen.

Weiterhin ist die Tree-Klasse reicher an Information als es die aktuelle Version Semedicos benötigt, da für Semedico im Moment nicht die vollständige Struktur des MeSH-Graphen verwandt wird, sondern nur eine vereinfachte Form. Zudem lässt sich die Software durch das Hinzufügen neuer Attribute bei Descriptors oder Tree Vertices leicht erweitern, ohne dass dazu tiefe Änderungen notwendig wären. Im Wesentlichen müssten dazu nur die Methoden zum Importieren und Exportieren angepasst werden.

Es ist denkbar, die Bibliothek direkt als Datenbank für Semedico zu nutzen. Statt also wie bisher die für Semedico erstellte Wissensbasis in ein separates DBMS zu exportieren, könnten alle Datenanfragen der Web-Suchmaschine direkt an Tree-Objekte gestellt werden. Dazu wären allerdings eine Reihe von Änderungen und Optimierungen bezüglich der Performance notwendig, sowie auch eine Erweiterung der Funktionalität der Tree-Klasse.

Da der Schwerpunkt der Studienarbeit auf dem MeSH liegt, wurden bisher keine Methoden implementiert, um Daten anderer Quellen, wie beispielsweise UniProt, zu importieren. Diese Erweiterung würde aber nur das Einlesen und Abbilden auf die Tree-Struktur umfassen und wäre mit kleinem Aufwand machbar.

Vergleich von MeSH-Trees

Die Umsetzung der Methoden aus Abschnitt 4 erlauben den Vergleich *beliebiger* MeSH-Trees und finden immer eine dazugehörige Transformationsfolge. Die erfolgreichen JUnit-Tests aus Unterabschnitt 6.6 belegen dies für den Vergleich der MeSH-Trees der Jahre 2008 und 2012: Die 2012er Daten enthalten etwa 26 000 Descriptors und 54 000 Tree Vertices und der Vergleich bestimmt 1 892 Descriptor-Additionen, 81 Descriptor-Löschungen, 258 Descriptor-Umbenennungen, 11 547 Vertex-Additionen, 2 462 Vertex-Löschungen, sowie 798 Vertex-Verschiebungen. Wendet man diese Modifikationen auf

den MeSH 2008 an, so erhält man genau den MeSH 2012 (bzw. die jeweiligen Tree-Objekte).

Es ist also möglich Änderungen zwischen MeSH-Versionen zu verfolgen. Auch Daten aus anderen Quellen lassen sich grundsätzlich damit vergleichen. Allerdings wäre es vermutlich sinnvoll, die Implementierung zum Vergleich von MeSH-Daten als Grundlage zu verwenden, und darauf basierend eine optimierte, d. h auf die Charakteristiken der Daten und dessen Veränderungen angepasste, Version des Vergleichens zu entwickeln.

Aktualisierung von Transformationsfolgen

Das Aktualisieren von Transformationsfolgen ergänzt, zusammen mit dem Vergleich von Tree-Objekten, die Verwaltungsfunktionalität. Damit ist es nicht nur möglich die Datenbasis für Semedico einmalig aus einer Reihe verschiedener Datensätze zusammenzustellen, sondern auch automatisch auf eine neue Version der jeweiligen Datensätze umzustellen. Denn das Zusammenstellen der Wissensbasis entspricht gerade der Anwendung von bestimmten Transformationen. Diese Transformationen können mit den in Abschnitt 2 dargestellten Methoden für eine neue Version der Daten aktualisiert werden - ohne Eingriff des Nutzers.

Auch hier gilt allerdings, wie im vorangegangen Abschnitt, dass es für Daten aus anderen Quellen sinnvoll scheint eine angepasste Version des Algorithmus' zu schreiben. Der Aufwand dafür sollte allerdings eher gering sein, da nur die Behandlung der verschiedenen Fälle verändert werden muss, grundsätzlich aber die Struktur und Fallunterscheidung unverändert bleibt.

Beim Update des Semedico-MeSHs vom MeSH 2008 auf den MeSH 2009 werden beispielsweise folgende Anzahl an Transformationen aktualisiert: 0 von 35 Descriptor-Additionen, 0 von 35 Vertex-Additionen, 131 von 246 Vertex-Verschiebungen, 1 238 von 3 946 Vertex-Löschungen und 22 von 19 895 Descriptor-Löschungen.

Zusammenfassung

Insgesamt stellt die entwickelte Software stellt eine flexible Grundlage zur Verwaltung und Aktualisierung der Wissensbasis für Semedico dar. Die zu Beginn aufgelisteten Ziele wurden in Hinsicht auf die wichtigste Datenquelle Semedicos erreicht: Der MeSH kann importiert, verändert und exportiert werden, zudem kann der Semedico-MeSH erstellt und aktualisiert werden.

Danksagung

Ich möchte mich besonders bei meinem Betreuer Erik Fäßler bedanken, der durch unsere konstruktiven und kritischen Gespräche viel zu dem Erfolg dieser Arbeit beigetragen hat.

Zudem bedanke ich mich bei Prof. Dr. Clemens Beckstein für seine ausführlichen Anmerkungen zur Arbeit im Vorfeld der Abgabe.

Dank geht auch an Eric Bach, für sein Interesse und die langen, gemeinsamen Abende im Büro. Ohne ihn hätte es bestimmt noch länger gedauert.

Selbstständigkeitserklärung

Hiermit versichere ich, Philipp Lucas, diese Ausarbeitung sowie die zugehörige Software selbstständig verfasst zu haben und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, sowie Zitate kenntlich gemacht zu haben.

Jena, 28. Mai 2013

Literatur

[1] H. Bunke. „Error correcting graph matching: on the influence of the underlying cost function“. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 21.9 (Sep. 1999), S. 917 –922. ISSN: 0162-8828. DOI: 10.1109/34.790431.

[2] G. Chartrand, G. Kubicki und M. Schultz. „Graph similarity and distance in graphs“. In: *Aequationes Mathematicae* 55 (1 1998). 10.1007/s000100050025, S. 129–145. ISSN: 0001-9054. URL: http://dx.doi.org/10.1007/s000100050025.

[3] John Clark und Derek Allan Holton. *A first look at graph theory. (Graphentheorie. Grundlagen und Anwendungen. Aus dem Engl. von Lydia Fritz.)* German. Heidelberg: Spektrum Akademischer Verlag. xii, 363 S. DM 58.00; öS 453.00; sFr. 56.00, 1994.

[4] The UniProt Consortium. „Reorganizing the protein space at the Universal Protein Resource (UniProt)“. In: *Nucl. Acids Res.* 40 (2011), S. D71–D75. DOI: 10.1093/nar/gkr981.

[5] Reinhard Diestel. *Graphentheorie*. Springer, 2010.

[6] *Entry Terms and Other Cross-References*. U.S. National Library of Medicine. Aug. 2012. URL: http://www.nlm.nih.gov/mesh/intro_entry.html.

[7] *Java API for XML Processing*. Okt. 2012. URL: http://de.wikipedia.org/wiki/Java_API_for_XML_Processing.

[8] *JGraphT API Documentation*. Okt. 2012. URL: http://jgrapht.org/javadoc/.

[9] Salim Jouili, Salvatore Tabbone und Ernest Valveny. „Comparing graph similarity measures for graphical recognition“. In: *Proceedings of the 8th international conference on Graphics recognition: achievements, challenges, and evolution*. GREC'09. Berlin, Heidelberg: Springer-Verlag, 2010, S. 37–48. ISBN: 3-642-13727-X, 978-3-642-13727-3. URL: http://dl.acm.org/citation.cfm?id=1875532.1875536.

[10] *Medical Subjects Headings Homepage*. English. U.S. National Library of Medicine. Sep. 2012. URL: http://www.nlm.nih.gov/mesh/.

[11] *MeSH Organization Principles*. U.S. National Library of Medicine. Aug. 2012. URL: http://www.nlm.nih.gov/mesh/intro_preface.html#pref_organizing.

[12] mkyong. *Java XML Tutorial*. Okt. 2012. URL: http://www.mkyong.com/tutorials/java-xml-tutorials/.

[13] Nelson u. a. „The MeSH translation maintenance system: Structure, interface design, and implementation“. In: *Proceedings of the 11th World Congress on Medical Informatics*. IOS Press, 2004, S. 67–69.

[14] W. Douglas Johnston Stuart J. Nelson und Betsy L. Humphreys. *Relationships in Medical Subject Headings (MeSH)*. English. National Library of Medicine, Bethesda, MD, USA. 2001. URL: http://www.nlm.nih.gov/mesh/meshrels.html.

[15] Roel Wuyts. *UML Class Diagrams*. 2006. URL: http://decomp.ulb.ac.be:9090/Courses/ami0506/05-ClassDiagrams.pdf.

[16] *XML MeSH Data Elements*. U.S. National Library of Medicine. Okt. 2012. URL: http://www.nlm.nih.gov/mesh/xml_data_elements.html.

[17] Laura A. Zager und George C. Verghese. „Graph similarity scoring and matching". In: *Applied Mathematics Letters* 21.1 (2008), S. 86 –94. ISSN: 0893-9659. DOI: 10.1016/j.aml.2007.01.006. URL: http://www.sciencedirect.com/science/article/pii/S0893965907001012.