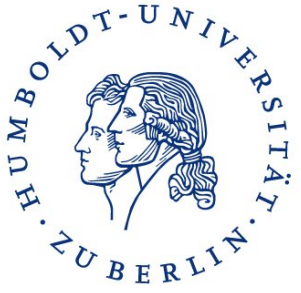


Diplomarbeit

Entwicklung eines Konverterframeworks für linguistisch annotierte Daten auf Basis eines gemeinsamen (Meta-)modells

Florian Zipser

30. November 2009

Gutachter: Prof. Dr. rer. nat. Ulf Leser
Prof. Dr. phil. Anke Lüdeling

Danksagung

An dieser Stelle möchte ich mich bei all denen bedanken, die mich während meines Studiums und meiner Diplomarbeit unterstützt haben.
Einen Dank richte ich an dieser Stelle an Guido Wachsmuth und Silvio Pohl die mir sehr bei der Erkundung der verschiedenen Techniken wie OSGi und QVT geholfen haben. Ich bedanke mich herzlich bei Julia Richling, Hagen Hirschmann und Amir Zeldes für die vielen Diskussionen, die Erklärungen über die Korpuslinguistik, sowie das ausgiebige Korrekturlesen dieser Arbeit. Weiter bedanke ich mich bei meinen Betreuern und Gutachtern Anke Lüdeling und Ulf Leser für die kontoversen Diskussionen.
Einen besonderen Dank richte ich an Konstanze Swist, die mir bei vielen Problemen mit Rat und Tat zur Seite gestanden und mich in dieser Zeit ertragen hat. Ich bedanke mich auch bei meinen Eltern, die mir dieses Studium möglich gemacht und mich die ganze Zeit über in meiner Entscheidung für dieses Studium bestärkt haben.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Die vorliegende Diplomarbeit leistet einen Beitrag zur Konvertierung von Daten im Bereich der Korpuslinguistik. Sie ist interdisziplinär und verbindet den Bereich Informatik und Korpuslinguistik.
In der Korpuslinguistik werden Vorkommen linguistischer Phänomene in empirischen Daten aufbereitet und in so genannten Korpora gespeichert. Aufgrund der Größe typischer Korpora ist es meistens notwendig, sie rechnergestützt zu analysieren.
Eine Herausforderung ist die Darstellung und Persistenzierung von Korpora. Hierzu hat sich eine Vielzahl von Werkzeugen und Formaten etabliert, die in der Regel aber nur für einen bestimmten Anwendungsfall entwickelt wurden und auch nur für diesen geeignet sind. Da ein Korpus aber oft mehr Facetten besitzt, als ein bestimmtes Werkzeug oder Format abbilden kann, ist es notwendig ein Korpus verschiedenen Werkzeugen zugänglich zu machen. Dafür müssen Korpora in verschiedene Formate konvertiert werden. Die Konvertierung von linguistischen Daten ist ein bisher wenig bearbeitetes Gebiet. Konverter werden meist für bestimmte Korpora entwickelt. Die Nachhaltigkeit der Konverter ist dadurch kaum gegeben.
An dieser Stelle setzt die vorliegende Arbeit an. Sie befasst sich mit der Frage, wie Konvertierungen linguistischer Daten aus unterschiedlichen Formaten nachhaltig und korpusübergreifend entwickelt werden können.

1.2. Zielsetzung und Vorgehensweise

Da Konverter zur Korpuskonvertierung oft nur einen Anwendungsfall haben, werden sie selten veröffentlicht. Viele der Konverter sind Teil eines Annotationswerkzeugs und bieten nur die Möglichkeit, ein Korpus in genau ein Zielformat zu konvertieren.
Aufgrund der vielen existierenden Formate ist es oft schwierig, ein Korpus aus Format A nach Format B zu konvertieren, da kein Konverter von A nach B existiert. Evtl. muss das Korpus über Zwischenschritte (weitere Formate wie C, D,...) in das Zielformat B konvertiert werden. Dieses Vorgehen erzeugt lange Konvertierungsketten und damit ggf. hohe Informationsverluste, da bspw. die Mächtigkeit der Formate sehr unterschiedlich sein kann. Die direkte Konvertierung von n Formaten ineinander erfordert (n^2-n) Konverter.
Um beide Probleme zu lösen, wähle ich einen Ansatz, der über ein gemeinsames (Meta-)Modell führt. Im Rahmen dieser Arbeit werde ich für fünf unterschiedliche Formate ein (Meta-)Modell entwickeln, das mächtig genug ist, alle Aspekte der einzelnen Formate abzudecken. Für jedes dieser Formate werde ich ein Mapping entwickeln, um es auf das gemeinsame (Meta-)Modell abzubilden. Abbildung 1.1 verdeutlicht den Ansatz.

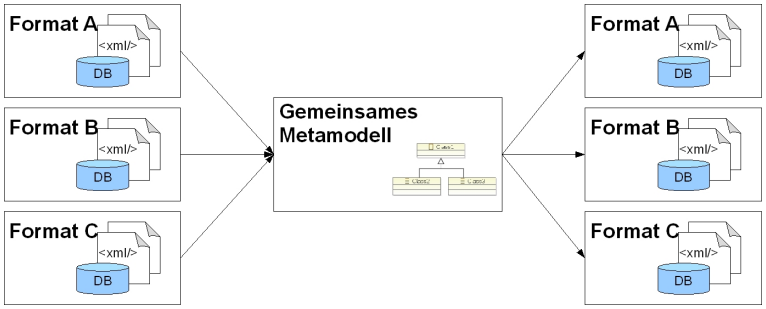


Abbildung 1.1.: gemeinsames (Meta-)Modell , über das Mappings von unterschiedlichen Formaten in unterschiedliche Formate durchgeführt werden können

Das Mapping über ein gemeinsames (Meta-)Modell reduziert die Anzahl benötigter Mappings und verkürzt ggf. die Konvertierungskette. Angenommen, es sollen n Formate aufeinander abgebildet werden, so werden statt $(n^2 - n)$ nur $2n$ Mappings benötigt. Jede Konvertierung besteht dann aus genau zwei Schritten.
Um zu gewährleisten, dass das gemeinsame (Meta-)Modell mächtig genug ist, alle Aspekte der untersuchten Formate abzudecken, muss dessen Mächtigkeit der Mächtigkeit der Vereinigung der einzelnen Aspekte der Formate entsprechen. Da die Mächtigkeit eines Formates jedoch schwer zu messen ist, werde ich einzelne Aspekte der Formate untersuchen und diese in Konzepten vereinigen. Auf der Basis der Konzepte werde ich die Mächtigkeiten der Formate miteinander vergleichen und das gemeinsame (Meta-)Modell entwickeln. Zur Umsetzung des gemeinsamen (Meta-)Modells werde ich modellbasierte Techniken verwenden und untersuchen, inwiefern sich diese dafür eignen.
Neben der Entwicklung des gemeinsamen (Meta-)Modells werde ich in dieser Arbeit einen Prototyp eines Konverterframeworks entwickeln. Die Entwicklung eines Frameworks anstelle eines Konverters, der lediglich die fünf Formate ineinander konvertiert, soll die nachhaltige Entwicklung von Konvertern ermöglichen. Das Framework wird auf dem gemeinsamen (Meta-)Modell basieren. Ein Konverter kann ein Importer, der Daten eines Formates auf das gemeinsame (Meta-)Modell abbildet, ein Exporter, der die Daten des gemeinsamen (Meta-)Modells auf ein Format abbildet, oder beides sein. Die hier untersuchten Formate sind ein kleiner Teil der im Bereich der Korpuslinguistik existierenden, daher soll das Konverterframework über einen Mechanismus verfügen, weitere Importer und Exporter hinzuzufügen. Dieser Mechanismus soll Nachhaltigkeit erzeugen, indem andere Projekte weitere Konverter entwickeln und über das Framework öffentlich zur Verfügung stellen können.
Korpora haben oft bestimmte Spezifika, die in einem Format nicht hinreichend ausgedrückt werden und bei einer allgemeinen Konvertierung verloren oder falsch gedeutet werden können. Daher werde ich eine Möglichkeit aufzeigen, wie trotz allgemeinen Im- und Exportern auf derartige Spezifika Rücksicht genommen werden kann.
Nicht zuletzt werde ich Rahmen dieser Arbeit die Performanz der Konvertierung von Korpora ansprechen und Möglichkeiten vorstellen, diese zu erhöhen. Dieses Thema wird allerdings nur am Rande und nicht ausschöpfend behandelt.

1.3. Aufbau der Arbeit

In Kapitel 2 werde ich grundlegende linguistische Begriffe einführen und die zentralen Bausteine von Korpora beschreiben. Weiter werde ich die formatbasierte Entwicklung mit der modellbasierten vergleichen und neben anderen in dieser Arbeit verwendeten Techniken die modellbasierte Technik EMF vorstellen. In Kapitel 3 werde ich fünf linguistische Annotationsformate vorstellen und Konzepte herausarbeiten, auf deren Basis ich die Mächtigkeit der Formate miteinander vergleiche. Die herausgearbeiteten Konzepte werde ich dann in Kapitel 4 benutzen, um auf deren Grundlage das gemeinsame (Meta-)Modell `Salt` zu entwickeln. Dieses (Meta-)Modell wird in Kapitel 5 als Grundlage des Konverterframeworks `Pepper` verwendet. Dort werde ich die Ansprüche an `Pepper` definieren und beschreiben, wie sie umgesetzt werden. In Kapitel 6 werde ich die verwendeten Techniken und die Vorgehensweise dieser Arbeit bewerten.
Zu dieser Arbeit habe ich im Anhang in Abschnitt A.5 ein Glossar erstellt, in dem ich bestimmte gekennzeichnete Fachbegriffe, die ich verwende, erkläre. Begriffe, die durch das Glossar erklärt werden, habe ich *kursiv* dargestellt.

1.4. verwandte Arbeiten

Auf dem Gebiet der Entwicklung eines gemeinsamen (Meta-)Modells oder Formates für linguistische Formate gibt es bisher nur wenige Arbeiten. Zwei dieser Arbeiten werde ich hier kurz erwähnen.
In [Ide et al., 2007] stellt die ISO einen Standard (GrAF) vor, um linguistische Formate auf der Grundlage eines Graphen zu entwickeln. Das **L**inguistic **a**nnotation **f**ramework LAF (siehe [Ide and Romary, 2003]) ist ein Framework basierend auf GrAF, auf dessen Basis linguistische Annotationsformate entwickelt werden können. In [Ide et al., 2007] beschreiben Ide und Suderman, wie auf der Grundlage eines solchen „pivot“-Formates die Anzahl der nötigen Mappings, um n Formate in m Formate zu konvertieren, reduziert werden können (vgl. [Ide et al., 2007] S. 2). Dieser Ansatz ist ähnlich dem hier vorgestellten. Auch das in Abschnitt 3 näher beschriebene Format PAULA (siehe 3.1.4) hat einen ähnlichen Ansatz. Der wesentliche Unterschied dieser Arbeiten zu der vorliegenden liegt darin, dass die Vorschläge der ISO und das Format PAULA formatbasiert[1] sind und nicht auf einem (Meta-)Modell aufbauen. Die Bearbeitung dieser Formate durch entsprechende Werkzeuge ist dabei nicht vorgesehen.
Ein vergleichbarer Ansatz zur Entwicklung eines Frameworks zur Konvertierung linguistischer Daten ist mir nicht bekannt. Viele Annotationswerkzeuge bieten zwar einen Import für „Fremd“-Formate an, jedoch nur einen Export in das Format, auf dem das Werkzeug basiert.

[1] Den Begriff der formatbasierten Entwicklung werde ich in Abschnitt 2.2 erklären.

2. Technische und terminologosche Grundlagen

In diesem Kapitel stelle ich die wesentlichen in dieser Arbeit verwendeten Begriffe, Methoden und Techniken vor. Zuerst erfolgt eine Beschreibung des Anwendungsgebietes Korpuslinguistik. Anschließend werde ich die Unterscheidung der Begriffe Format und Modell beschreiben und die Methoden der format- bzw. modellbasierten Entwicklung näher betrachten. Danach werde ich einige Techniken beschreiben, die ich in diesem Rahmen verwendet habe. Ich schließe dieses Kapitel mit der Darstellung einer Technologie der *ISO* zur Beschreibung der Semantik linguistischer Annotationen.

2.1. Grundlagen des Anwendungsgebietes

Viele Begriffe, die in der Korpuslinguistik verwendet werden, sind für informatische Verhältnisse relativ informal beschrieben. Deshalb werde ich in diesem Abschnitt versuchen, grundlegende Bausteine herauszuarbeiten, mit denen korpuslinguistische Analysen dargestellt werden können. Diese Bausteine werde ich semiformal als Begriffe einführen. Die Sammlung der Bausteine bildet das sprachliche Repertoire, dem ich mich im Verlauf dieser Arbeit bedienen werde, um die in 1.2 genannten Ziele umzusetzen.
Die Linguistik beschäftigt sich mit der Analyse und der Beschreibung von natürlicher Sprache. Im Gegensatz zu künstlichen Sprachen wie bspw. Programmiersprachen gibt es kein einheitliches Sprachmodell und keine exakte Beschreibung. Daher versucht die Linguistik durch Beobachtungen der Nutzung von Sprache, Regelmäßigkeiten zu identifizieren und anhand dieser die Sprache und deren Phänomene zu beschreiben.
Die Korpuslinguistik beschäftigt sich mit der quantitativen und qualitativen Analyse empirischer Daten. Um aussagekräftige Modelle zu erzeugen, die eine Sprache beschreiben, werden in der Korpuslinguistik viele Sprachbeispiele gesammelt, verarbeitet und analysiert. Die gesammelten Daten können sehr unterschiedlicher Natur sein. Dabei kann es sich bspw. um geschriebene Sprache wie Zeitungsartikel der heutigen Zeit, ein Gedicht des 12 Jhd. oder auch um gesprochene Sprache handeln. Für linguistische Analysen werden die Sprachdaten i.d.R. in unterschiedliche Bestandteile zerlegt, denen dann linguistische Kategorien zugeordnet werden. Die Kategorisierung der Bestandteile wird als Annotation bezeichnet. Daten mitsamt ihren Annotationen bilden ein Korpus. Das Korpus ist der zentrale Untersuchungsgegenstand der Korpuslinguistik, die durch Lemnitzer und Zinsmeister wie folgt beschrieben wird:

> „Als Korpuslinguistik bezeichnet man die Beschreibung von Äußerungen natürlicher Sprachen, ihrer Elemente und Strukturen, und die darauf aufbauende

word	pos	lemma
The	DT	the
TreeTagger	NP	TreeTagger
is	VBZ	be
easy	JJ	easy
to	TO	to
use	VB	use
.	SENT	.

Tabelle 2.1.: Beispiel aus einer Wortarten-Analyse. Die erste Spalte enthält das zu annotierende Wort, die zweite Spalte enthält eine part-of-speech-Annotation und die dritte Spalte eine Lemma-Annotation.

> Theoriebildung auf der Grundlage von Analysen authentischer Texte, die in Korpora zusammengefasst sind. [...] Korpusbasierte Sprachbeschreibung kann verschiedenen Zwecken dienen, zum Beispiel dem Sprachunterricht, der Sprachdokumentation, der Lexikographie oder der maschinellen Sprachverarbeitung“ ([Lothar Lemnitzer, 2006], S. 9).

Abhängig vom Zweck der „korpusbasierten Sprachbeschreibung“ ist auch die Art der Annotation eines Korpus. Damit unterscheiden sich Korpora zum einen in der Art der Sprache die sie abbilden und zum anderen in dem Forschungszweck, für den sie erzeugt wurden.
An dieser Stelle werde ich ein paar Ausschnitte aus unterschiedlichen Korpora zeigen, um ein Gefühl davon zu vermitteln, wie ein Korpus aussehen kann und wie sich die Unterschiede einzelner Korpora bemerkbar machen können.
Tabelle 2.1 zeigt eine automatische Annotation die von dem Analysewerkzeug TreeTagger (siehe [Schmid, 1994]) erstellt wurde. Der Text „The TreeTagger ...“ wurde auf Wort- und Zeichenebene zerlegt und kategorisiert, indem den Wörtern bzw. Zeichen Werte für Wortarten (auch part-of-speech-Annotation genannt) und jeweils eine Grundform (eine Lemma-Annotation) zugewiesen wurden.

Der Ausschnitt aus Abbildung 2.1 stellt einen Dialog zweier Personen dar und bezieht sich hauptsächlich auf die Darstellung z.T. parallel stattfindender Äußerungen (Diskursanalyse).

Bei dem Ausschnitt aus Abbildung 2.2 handelt es sich um einen Zeitungstext, bei dem die linguistische Analyse auf die Satzstruktur abzielt. Die syntaktische Analyse ist in einer *Baum-* bzw. *DAG*-artigen Struktur dargestellt. Das Beispiel zeigt die komplexe Annotation eines Satzes. Jedes Wort ist mit einer part-of-speech-Annotation versehen. Zusätzlich werden Wörter zu Phrasen zusammengefasst, die funktionale Bestandteile des Satzes bilden. Beispielsweise werden die Wörter „Die“ und „Tagung“ zu einer Nominalphrase (NP) und anschließend zu einem Satz (S) zusammengefasst. Auch die Verbindungen der einzelnen Einheiten sind annotiert. Die Verbindung zwischen dem Satz und der aus den Wörtern „Die“ und „Tagung“ bestehenden Phrase, trägt die Annotation „SB“. Dies zeigt an, dass es sich bei dieser Phrase um das Subjekt des gesamten Satzes handelt.

Sprecher	Diskurs	
MAX:	Du fällst mir immer ins	Wort.
TOM:		Stimmt ja gar nicht.

Abbildung 2.1.: Ausschnitt aus einem Gespräch zweier Personen. Beispiel stammt aus [Schmidt, 2002]. Beide Äußerungen finden zum Teil parallel statt, daher ist die Äußerung von MAX in zwei Ereignisse aufgeteilt. Das erste Ereignis „Du fällst mir immer ins“ steht alleine, da zur gleichen Zeit kein anderes Ereignis stattgefunden hat. Die Ereignisse „Wort“ von MAX und „Stimmt ja gar nicht“ von TOM finden im gleichen Zeitabschnitt statt.

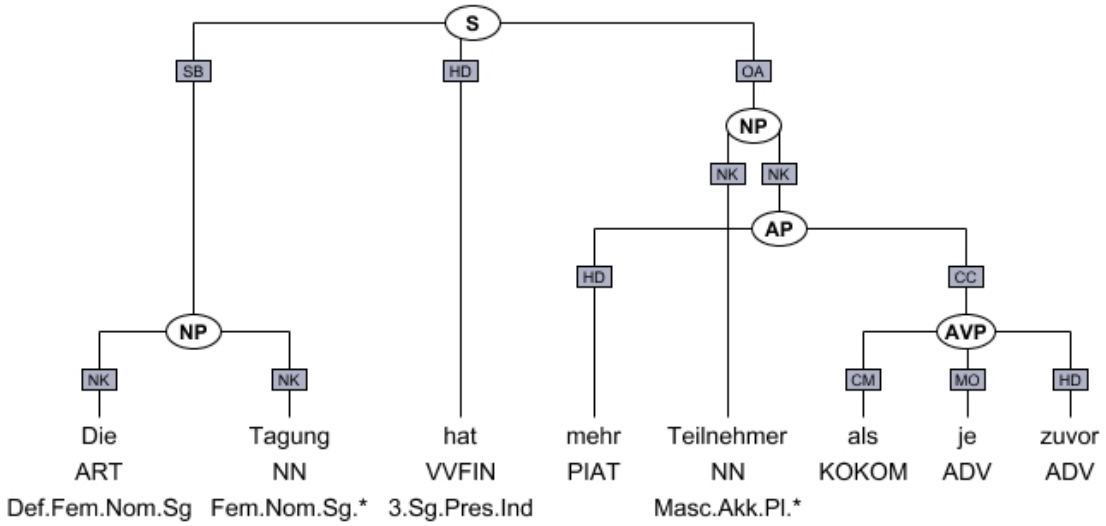


Abbildung 2.2.: Syntaktische Analyse eines Satzes aus dem Tiger-Korpus. Der Ausschnitt stammt aus dem Werkzeug TigerSearch. Quelle: Universität Stuttgart (http://www.ims.uni-stuttgart.de, zuletzt besucht am 04.08.2009)

An den gerade gezeigten Beispielen ist erkennbar, dass sich linguistische Analysen auf unterschiedliche Bestandteile von Sprache beziehen. Analysen können flach sein und sich bspw. auf Wörter (Tabelle 2.1) oder auf wortübergreifende Bestandteile (Abbildung 2.1) beziehen. Eine linguistische Analyse sprachlicher Daten kann aber auch komplexere Strukturen und Annotationen erzeugen, wie sie bspw. in Abbildung 2.2 zu sehen sind.
Trotz der Unterschiedlichkeit der existierenden Korpusdaten bestehen die in dieser Arbeit betrachteten linguistischen Analysen aus gleichen Bausteinen. Diese Bausteine werde ich nun einführen und beschreiben.

2.1.1. Grundbausteine linguistischer Analysen

Die Unterteilung linguistischen Analysen nehme ich vor, um deren Aufbau detaillierter beschreiben zu können. Eine linguistische Analyse besteht aus drei Komponenten: 1) einer Menge von Primärdaten, 2) Strukturen über den Primärdaten und 3) Annotationen über diesen Strukturen.
Für diese Arbeit muss ein Unterschied zwischen den Begriffen Primärdaten und Primärtext gemacht werden, welchen ich im Folgenden beschreibe.

Begriff 1 (Primärdaten) *Primärdaten sind der Ursprung und die Grundlage aller weiteren Bearbeitungsschritte und bezeichnen die digitalisierte Form einer Datenquelle auf die sich die Strukturen und Annotationen der linguistischen Analyse beziehen. Diese Datenquelle kann sowohl textueller, gesprochener als auch visueller Natur sein.*

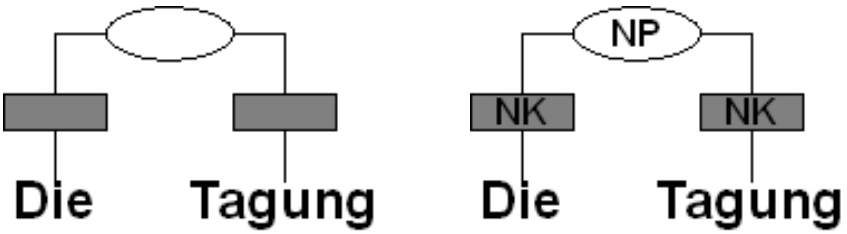
Begriff 2 (Primärtext) *Unter einem Primärtext verstehe ich eine digitalisierte, schriftliche Form der Primärdaten. Liegen die Primärdaten bereits in schriftlicher Form vor, sind Primärdaten und Primärtext identisch. Liegen die Primärdaten bspw. als gesprochene Daten vor, so bezeichne ich mit dem Primärtext eine verschriftlichte bspw. protokollierte Form der Daten.*

Jede in dieser Arbeit betrachtete Analyse enthält in ihrer Menge der Primärdaten mindestens einen Primärtext.
Abbildung 2.3 zeigt einen Ausschnitt aus Abbildung 2.2 (linker unterer Teil). Anhand dieses Ausschnittes werde ich nun den Unterschied zwischen Struktur und Annotation beschreiben.
Abbildung 2.3(a) zeigt lediglich die Struktur über einem Teil des Primärtextes. Die Struktur ist lediglich ein Gerüst, auf dem Annotationen erfolgen können. Sie besitzt selber keine interpretierbare linguistische Deutung. Einzelne Elemente der Struktur können auch als Platzhalter für Annotationen betrachtet werden. In Abbildung 2.3(a) werden die beiden Wörter „Die“ und „Tagung“ zu einer Einheit, hier symbolisiert durch ein Oval, zusammengefasst. Dieser Einheit ist noch keine Kategorie zugewiesen. Abbildung 2.3(b) zeigt die gleiche Struktur, erweitert um Annotationen. Durch eine Annotation erhält ein Element der Struktur eine Kategorie und bekommt eine linguistische Deutung. In dem Beispiel wird der strukturellen Einheit die Kategorie „NP“ zugewiesen. Dadurch bekommt diese Einheit die Bedeutung einer Nominalphrase. Neben Einheiten können auch Verbindungen zwischen Einheiten kategorisiert werden. In diesem Beispiel wird den beiden Verbindungen zwischen den Wörtern und der Einheit jeweils die Kategorie „NK“ zugewiesen.

Die Struktur einer linguistischen Analyse besteht ihrerseits aus weiteren Bausteinen. Diese werde ich nun beschreiben. Abbildung 2.4 zeigt die einzelnen Bausteine und ihre Beziehung zu einander.

Um Primärtexte linguistisch analysieren zu können, werden diese i.d.R in kleinste Einheiten unterteilt. Diese werden als Token bezeichnet.

Begriff 3 (Token) *Ein Token ist ein Platzhalter und beschreibt die kleinste Einheit, in die ein Primärdatum unterteilt wird. Bezogen auf das Primärdatum haben zwei Token eine klare Ordnung. Für zwei Token a und b gilt, wenn $a \neq b$, dann $a < b$ oder $a > b$.*



(a) Struktur (b) Struktur mit Annotationen

Abbildung 2.3.: Abbildung 2.3(a) zeigt einen Ausschnitt aus Abbildung 2.2. Dabei wird nur die Struktur abgebildet, wohingegen Abbildung 2.3(b) die gleiche Struktur mitsamt den Annotationen aus Abbildung 2.2 darstellt.

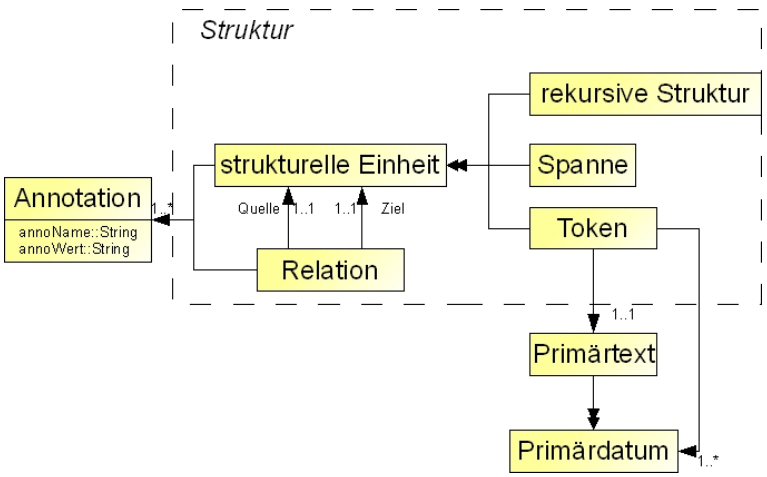


Abbildung 2.4.: Modell einer linguistischen Analyse bestehend aus den drei Hauptkomponenten Primärdaten, Struktur und Annotationen. Ebenfalls werden die Bausteine der Struktur abgebildet und zueinander in Beziehung gesetzt. Eine Beschreibung der verwendeten Notation findet sich in Abschnitt 2.4.

Der Prozess, der die Unterteilung eines Primärdatums in Token vornimmt, wird allgemein als Tokenisierung bezeichnet.
Oft werden Tokenisierungen entlang von semantischen Einheiten wie „Silbe“, „Wort“ oder auch „Satz“ vorgenommen. Dies muss aber nicht immer der Fall sein. Die Tokenisiserung kann auch die Zerlegung eines Primärtextes in Zeichen bedeuten. Umfasst ein Token gar keinen Teil des Primärdatums wird von einem leeren Token gesprochen.
Die Zerteilung eines Primärtextes in Token reicht nicht aus, um die Bestandteile eines Textes zu beschreiben. Beispielsweise kann der Satz „Ich stehe auf.“ in die drei Token „Ich“, „stehe“ und „auf“ zerlegt werden. Um die Zeichenketten „stehe“ und „auf“ bspw. mit einer Lemma-Annotation „lemma=aufstehen“ zu annotieren, müssen diese zunächst zu einem größeren Baustein zusammengefasst werden. Dieser Baustein kann entweder eine Spanne oder eine rekursive Einheit sein.

Begriff 4 (Spanne) *Eine Spanne ist ein Platzhalter und enthält eine geordnete Menge*

von Token oder ist über Relationen mit einer geordneten Menge von Token verbunden. Eine Spanne ist kontinuierlich, wenn für alle Paare aus der Menge der Token t_i, t_{i+1} mit $i \in \mathbb{N}$ gilt: $\nexists t_j (j \in \mathbb{N})$ für das gilt: $t_i < tj < t_{i+1}$. Ist eine Spanne nicht kontinuierlich, so ist sie diskontinuierlich.

Begriff 5 (rekursive Einheit) *Eine rekursive Einheit ist ein Platzhalter und kann neben einer geordneten Menge von Token und Spannen auch weitere rekursive Einheiten enthalten oder über Relationen mit ihnen verbunden sein. Rekursive Einheiten werden in der Korpuslinguistik auch als Nicht-Terminale bezeichnet und können zur Bildung von Baum-, DAG und Graph-artigen Strukturen genutzt werden.*

Die Platzhalter aus Abbildung 2.2, die die Annotationswerte „S“, „NP“, „AP“ oder „AVP“ tragen, sind bspw. rekursive Einheiten. An diesem Beispiel wird klar, wie rekursive Einheiten genutzt werden können, um mit ihnen Bestandteile eines Primärdatums, wie z.B. Token, zu größeren Strukturen zusammenzufassen. Hier werden bspw. die Wörter „Die“ und „Tagung“ zu einer Nominalphrase zusammengefasst. Die Nominalphrase ist Teil einer größeren Struktur, dem Satz.
Die gerade vorgestellten Bausteine, Token, Spanne und rekursive Einheit, werde ich im Folgenden allgemein als strukturelle Einheiten bezeichnen. Um strukturelle Einheiten miteinander in Beziehung zu setzen, werden so genannte Relationen erzeugt.

Begriff 6 (Relation) *Eine Relation ist eine gerichtete Verbindung zwischen zwei strukturellen Einheiten, die diese zueinander in Beziehung setzt. Die Bedeutung der Relation ist dabei i.d.R. durch eine Annotation gegeben.*

Relationen können explizit angegeben werden, wie in Abbildung 2.2 z.B. zwischen dem Wort oder auch Token „Die“ und der rekursiven Einheit mit dem Annotationswert „NP“’. Sie können aber auch implizit bestehen.
Strukturelle Einheiten beziehen sich immer auf Ausschnitte von Primärdaten. Diese Ausschnitte bezeichne ich als Primärfragmente.

Begriff 7 (Primärfragment) *Ein Primärfragment ist ein Ausschnitt eines Primärdatums. Ist das Primärdatum ein Primärtext, ist ein solcher Ausschnitt eine aufeinanderfolgende Zeichenkette aus dem Primärtext.*

In dem Primärtext „Die Tagung hat ...“ aus Abbildung 2.2 bezieht sich das erste Token auf die Zeichenkette „Die“. In diesem Zusammenhang spricht man in der Linguistik von einer Abdeckung. Das erste Token deckt die Zeichenkette „Die“ ab. Abdeckung und Primärfragmente beziehen sich aber nicht nur auf Token, sondern allgemein auf strukturelle Einheiten. Primärfragmente können durch eine abdeckungsvererbende Relation von einem Element an ein anderes weitergegeben werden.

Begriff 8 (Abdeckungsvererbung) *Die Abdeckungsvererbung ist eine Eigenschaft einer Relation. Relationen können in abdeckungsvererbend und nicht-abdeckungsvererbend unterteilt werden.*
Besteht zwischen zwei strukturellen Einheiten a und b eine abdeckungsvererbende Relation $r = (a, b)$ mit der Quelle a und dem Ziel b, so ist die Zeichenkette des von b abgedeckten Primärfragments ebenfalls Teil des abgedeckten Primärfragments von a.

Die Relationen aus Abbildung 2.2 sind abdeckungsvererbend. Dies bedeutet bspw., die strukturelle Einheit mit der Annotation „NP“ deckt die Primärfragmente „Die“ und „Tagung“ ab.

Begriff 9 (Annotation) *Eine Annotation ist ein Paar, bestehend aus einem Annotationsnamen und einem Annotationswert. Der Annotationsname beschreibt die Art der Kategorisierung, der Annotationswert beschreibt die genaue Kategorie. Beispielsweise legt die Annotation „part-of-speech='ART'“ fest, dass die damit annotierte strukturelle Einheit in der Kategorie „part-of-speech“ den Wert „ART“ (Artikel) hat.*

Im Folgenden wird diese Art der Annotation auch als Attribut-Wert-Paar-Annotation bezeichnet.
Linguistische Analysen können zu Dokumenten und Korpora zusammengefasst werden. Dadurch lässt sich die Datenmenge unterteilen und ist damit für Werkzeuge besser verarbeitbar.

Begriff 10 (Dokument) *Ein Dokument ist ein 4-Tupel bestehend aus einer Menge zusammengehörender Primärdaten, struktureller Einheiten, die sich auf die Primärdaten beziehen, Relationen zwischen den Primärdaten und Annotationen. Ein Dokument besitzt einen Namen und kann ebenfalls ein Platzhalter für eine Annotation sein.*

Dokumente können zu größeren Einheiten - den Korpora - zusammengefasst werden.

Begriff 11 (Korpus) *Ein Korpus ist eine Sammlung zusammengehörender Dokumente und Korpora. Korpora, die andere Korpora enthalten, werden als Superkorpora bezeichnet. Korpora, die in einem Superkorpus enthalten sind, heißen Subkorpora. Wie Dokumente besitzen auch Korpora Namen und können annotiert werden.*

Die Betrachtung unterschiedlicher Facetten von Sprachdaten hat dazu geführt, dass sich für die Verarbeitung und Analyse von Korpora eine breite Palette von Werkzeugen etabliert hat. In Tabelle 2.1 und den Abbildungen 2.2 und 2.1 werden nicht nur verschiedene Arten von Daten gezeigt (ein beschreibender Satz, ein Ausschnitt aus einem Diskurs und ein Zeitungsartikelausschnitt), sondern auch unterschiedliche Methoden der Darstellung durch unterschiedliche Werkzeuge (TreeTagger [Schmid, 1994], TigerSearch [Universität-Stuttgart et al., 2007] und EXMARaLDA [Schmidt, 2009]). Die meisten dieser Werkzeuge sind aus unterschiedlichen Forschungsprojekten hervorgegangen. Daher basieren sie auf unterschiedlichen linguistischen Analysen und verwenden unterschiedliche Teilmengen der Menge der gerade vorgestellten Bausteine. Die unterschiedliche Beschreibung der Daten bedingt unterschiedliche Formate, die die Werkzeuge zur Speicherung der Daten benutzen. So erfordern bspw. syntaktische Analysen andere Werkzeuge und damit Formate als bspw. Diskursanalysen. Handelt es sich aber um ein Korpus, das unterschiedlich analysiert werden soll, so entstehen unterschiedliche Repräsentationen ursprünglich gleicher Primärdaten. Durch die weitere Bearbeitung der Daten kann es passieren, dass sich die Daten getrennt voneinander entwickeln und nicht mehr ohne größeren technischen Aufwand vergleichbar sind. Das in dieser Arbeit vorgestellte Konverterframework `Pepper` soll eine Möglichkeit eröffnen, gemeinsam genutzte Bausteine von einem Format

ein anderes zu überführen. Damit können gleiche Daten unterschiedlicher Formate zum einen mit einander verglichen werden und zum anderen können sie auf der gleichen Basis aufbauen (gleiche Primärdaten, gleiche Token etc.), um sich durch die weitere Bearbeitung nicht zu stark voneinander zu entfernen.
Da die Konvertierung linguistischer Daten innerhalb dieser Arbeit auf Formaten basiert, werde ich mich im nächsten Abschnitt (Abschnitt 2.2) mit dem Begriff Format beschäftigen und die Lesart vorgeben, die ich in dieser Arbeit benutzen werde.

2.2. Format und formatbasierte Entwicklung

Da sich diese Arbeit mit der *Konvertierung* von verschiedenen Formaten beschäftigt, möchte ich an dieser Stelle kurz umreißen, was ich unter dem Begriff Format verstehe. Außerdem werde ich den Begriff formatbasierte Entwicklung einführen, um ihn der in Abschnitt 2.3 vorgestellten modellbasierten Entwicklung gegenüber zu stellen. Weiter werde ich einige Nachteile der formatbasierten Entwicklung nennen, die zu Schwierigkeiten sowohl bei der Analyse der untersuchten Formate als auch bei der Entwicklung von Mappings führten.
Die Auseinandersetzung mit dem Begriff Format ist notwendig, da er in unterschiedlichen Zusammenhängen leicht abweichende Bedeutungen bekommt, die hier aber missverständlich wirken können.
Eine Lesart des Begriffs Format bezeichnet eine rein technische Beschreibung von Daten. Dabei gelten beispielsweise CSV (Character Separated Values auch Comma Separated Values genannt), SGML (Standard Generalized Markup Language) und *XML* (eXtensible Markup Language) als Formate. Diese Formate zeichnen sich dadurch aus, dass sie in unterschiedlichen technischen Szenarien verwendet werden. Ein technisches Szenario für den Einsatz von CSV-Formaten ist bspw. der Einsatz in relationalen Datenbanken. Über eine Bedeutung der Daten treffen die Formate keinerlei Aussage und können daher alle Arten von Daten enthalten.
In einer zweiten Lesart werden Formate als eine inhaltsbezogene Beschreibung verstanden. In diesem Fall enthält ein Format nur Daten, die einer bestimmten Deutung unterliegen. Die später in Kapitel 3 vorgestellten Formate fallen unter diese Lesart. Als Beispiel sei hier vorweg die Speicherung des Annotationswerkzeuges EXMARaLDA [Schmidt, 2009] genannt. Diese wird vielerorts als das EXMARaLDA-Format bezeichnet. Gemeint ist damit i.d.R. eine bestimmte Struktur und Deutung linguistischer Daten.
Für diese Arbeit werde ich mich zweiter Lesart anschließen und immer, wenn ich von einem Format in einer bestimmten Technik spreche, die Technik vorweg nennen (bspw. XML-Format).
Unter der formatbasierten Entwicklung verstehe ich die Erzeugung einer Speicherung (Persistenzierung) von Daten auf Grundlage eines bestimmten technischen Formates. Dadurch entsteht eine Vermischung syntaktischer Darstellung und Deutung der Daten. Es fehlt an einer Abstraktionsebene, die es erlaubt, die Struktur und Deutung von Daten außerhalb einer bestimmten Technik zu betrachten. Diese Art der Entwicklung birgt einige Probleme in sich, die ich hier kurz skizzieren werde.

1. Mit der formatbasierten Entwicklung geht die frühzeitige Festlegung auf ein tech-

nisches Format einher. Das kann dazu führen, dass technische Einschränkungen, die das gewählte technische Format mit sich bringt, zu Einschränkungen in der Ausdruckskraft führen.

2. Bei der Deutung der Daten in einem Format ist es oft schwer zu erkennen, welche Beschränkungen der Technik und welche Beschränkungen der Semantik geschuldet sind. Bspw. ist das technische Format *XML* baumbasiert. Daten in diesem Format müssen als *Baum* vorliegen. Diese implizite Eigenschaft kann genutzt werden, um Daten miteinander in Beziehung zu setzen. Die Baumstruktur kann aber auch einfach nur technischer Natur sein und keine Bedeutung für die Daten haben. Es ist oft schwer zu erkennen, welcher der beiden Fälle im konkreten Fall vorliegt.

3. Gerade in der Welt der *XML* ist der Gedanke stark verbreitet, dass ein Format menschen- und maschinenlesbar sein sollte. Dieser Gedanke war zwar wichtig bei der Entstehung der *XML*, kann aber in der Praxis bei großen Daten kaum eingehalten werden. In der formatbasierten Entwicklung gerade im Bereich der *XML* wird überdies oft die Entwicklung einer verarbeitbaren *API* vernachlässigt. Das führt dazu, dass - wie in dieser Arbeit, bei einigen der untersuchten Formate - erst Softwarekomponenten entwickelt werden müssen, die die Verarbeitung der Daten ermöglichen.

Ein besserer Umgang mit den gerade genannten Probleme kann durch die modellbasierte Entwicklung erreicht werden. Mit Modellen, Metamodellen und der modellbasierten Entwicklung wird sich der nächste Abschnitt (siehe Abschnitt 2.3) beschäftigen.

2.3. Modell, Metamodell und modellbasierte Entwicklung

In diesem Abschnitt werde ich die Begriffe Modell, Metamodell und modellbasierte Entwicklung einführen und beschrieben. Dies ist wichtig für die vorliegende Arbeit, da das in Abschnitt 4 vorgestellte gemeinsame (Meta-)Modell modellbasiert entwickelt wurde. Außerdem habe ich die in Abschnitt 3.1 beschriebenen Formate z.T. in Metamodelle überführt. Weiter werde ich die Grundlagen und die Konsequenzen einer modellbasierten Softwareentwicklung skizzieren. Dabei wird deutlich werden, welche Vorteile dieser Ansatz gegenüber der formatbasierten Entwicklung bietet. Abschließend werde ich in einem Unterabschnitt einige Performanznachteile beschreiben, die durch die modellbasierte Entwicklung entstehen.
Um Missverständnissen vorzubeugen, sei hier darauf hingewiesen, dass in dieser Arbeit die Lesart der Begriffe Modell und Metamodell verwendet wird, wie sie in der modellbasierten Entwicklung vorherrscht und auch in der MDA [Miller and Mukerji, 2003] verwendet wird. Im Gegensatz zu der relativ verbreiteten, intuitiven Vorstellung wird in der modellbasierten Entwicklung unter einem Modell Folgendes verstanden:

Begriff 12 (Modell) *Ein Modell ist eine Abstraktion der Wirklichkeit. Durch ein Modell wird versucht, einen bestimmten Ausschnitt der Realität vereinfacht darzustellen. Welche*

Ausschnitte der Realität in einem Modell enthalten sind, hängt von dem Zweck eines Modells ab. Diese Abstraktion ist ein wichtiger Schritt, um die Realität besser zu verstehen und greifbarer zu machen. Ein Modell im Sinne der modellbasierten Entwicklung besitzt immer ein Metamodell, durch das es definiert wird.

Im Rahmen dieser Arbeit kann ein Korpus als ein Modell verstanden werden. Es bildet einen Ausschnitt natürlicher Sprache ab. Die Darstellungsmöglichkeiten der Realität hängen von den Möglichkeiten ab, die die Modellbeschreibungssprache (das Metamodell) zur Verfügung stellen.

Begriff 13 (Metamodell) *Ein Metamodell ist ebenfalls ein Modell auf einer höheren Abstraktionsebene. Ein Metamodell stellt Konzepte bereit, mit denen ein Modell Daten darstellen kann.*

Wenn ein Korpus ein Modell ist, dann ist die Beschreibung der Bestandteile eines Korpus das dazugehörige Metamodell. Bspw. können die Grundbausteine, die ich in Abschnitt 2.1.1 benannt habe, als ein Metamodell bezeichnet werden. In der modellbasierten Entwicklung werden Techniken verwendet, um Metamodelle formal zu beschreiben und aus textuellen Beschreibungen wie denen in in Abschnitt 2.1.1 verarbeitbare Metamodelle zu entwickeln. Ein Metamodell wird zwar in einer technischen Beschreibung entwickelt, setzt aber keine bestimmte Technik voraus, in der die Daten eines Modells gespeichert werden können. Das Metamodell bildet die Abstraktionsebene, die der formatbasierten Entwicklung fehlt. Ein Modell (bspw. ein Korpus) kann in unterschiedlichen Formaten gespeichert werden und behält die Konzepte, die durch das Metamodell vorgegeben sind.
Dieser Zusammenhang ist relevant für diese Arbeit, da, wie bereits erwähnt, nicht nur das gemeinsame (Meta-)Modell auf diese Weise entwickelt wurde, sondern auch einige der untersuchten Formate in Metamodelle überführt wurden. Daher werde ich an dieser Stelle am Beispiel des EXMARaLDA-Formats den Zusammenhang zwischen einem Format und einem Metamodell exemplarisch anhand des 4-Ebenen-Modells beschreiben. Das 4-Ebenen-Modell stammt aus der modellbasierten Entwicklung und beschreibt den Zusammenhang zwischen Modell- und der Metamodellebene. Abbildung 2.5 zeigt auf der rechten Seite (der Modellseite) das 4-Ebenen-Modell und die Beziehung zwischen technik-unabhängigem Modell und dessen Beschreibung durch ein Metamodell bzw. ein Meta-Metamodell. Dem gegenüber steht auf der linken Seite (der Formatseite) das entsprechende technik-abhängige Pendant gezeigt. Zwei Ebenen beider Seiten stehen insofern zueinander in Beziehung, als dass die im Bild untere Ebene durch die darüberliegende beschrieben wird. Auf der Modellseite ist die Beziehung zwischen zwei Ebenen M_i und M_{i+1} durch eine „Instanz-von“- und zwischen den Ebenen M_{i+1} und M_i durch eine „Modell-von“-Beziehung beschrieben. Das 4-Ebenen-Modell besitzt als unterste Ebene die „Instanzebene“. Mit dieser wird die reale Welt, also der Ursprung der Daten, bezeichnet. Ein Ausschnitt der Instanzebene findet sich in Form eines Modells auf der „Modellebene“ wieder. Das Modell beschreibt die Realität. In der „Metamodellebene“ werden Metamodelle definiert, die die einzelnen Konzepte eines Modells festlegen. Ein Metamodell besteht wiederum aus Konzepten, die ebenfalls auf einer höheren Ebene der „Meta-Metamodellebene“ beschrieben werden. Die Sprachen der Meta-Metamodellebene

wie bspw. ECore [Eclipse Foundation, 2009a] oder auch UML [OMG, 2009b] sind so definiert, dass sie neben der Beschreibung von Metamodellen auch ihre eigenen Beschreibungen sind.

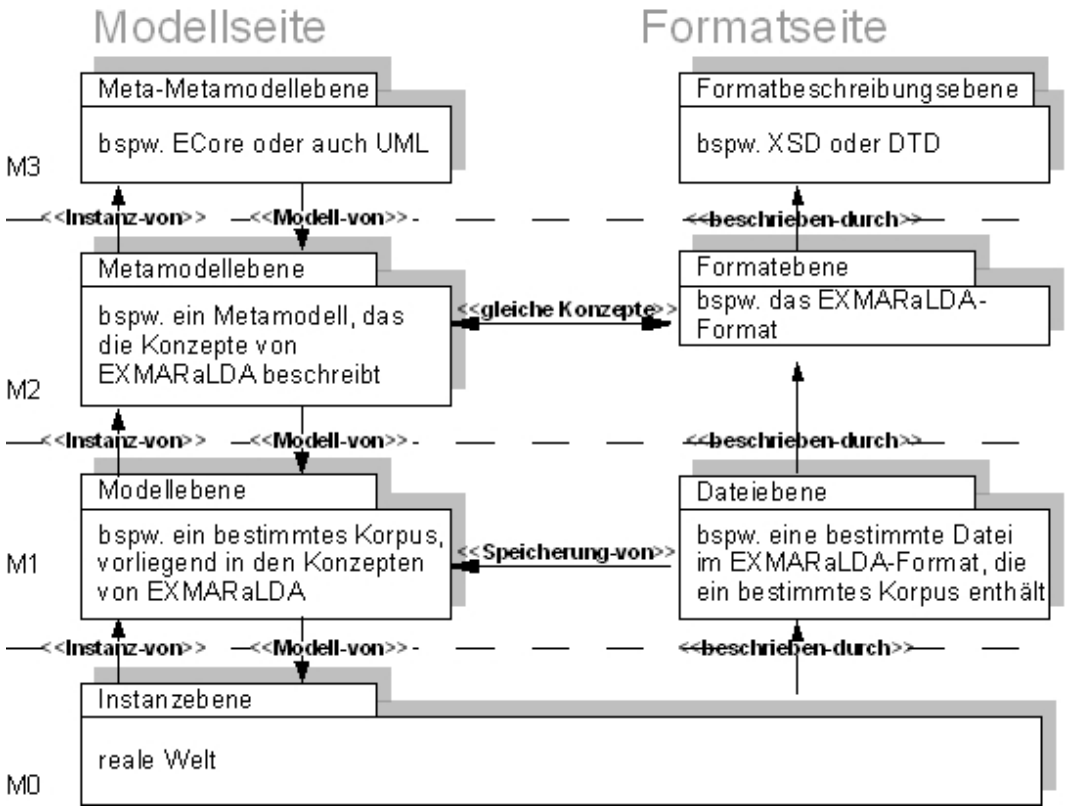


Abbildung 2.5.: 4-Ebenen-Modell bezogen auf die modellbasierte und die formatbasierte Entwicklung

Abbildung 2.5 zeigt am Beispiel des EXMARaLDA-Formats, wie dieses zu einem Metamodell in Beziehung steht. Ein Modell kann auf verschiedene Arten persistenziert werden. Im Beispiel von EXMARaLDA kann eine Persistenzierung in Form des EXMARaLDA-Formats erfolgen. Die Formatseite ist an eine bestimmte Technik, in diesem Fall XML, gebunden. Die Modellseite hingegen bezieht sich auch auf eine bestimmte Technik[1], diese dient aber nur der Beschreibung des Modells und Metamodells, erzeugt aber keine technische Abhängigkeit bezogen auf die Speicherung eines Modells. In dieser Arbeit besitzen Modelle im Gegensatz zu Formaten einen rein virtuellen Charakter. Darunter verstehe ich, dass Modelle allein im Hauptspeicher existieren, wohingegen Formate ebenfalls auf ein Speichermedium (wie eine Festplatte) geschrieben werden können. Ein Modell kann nur durch ein oder mehrere Formate persistenziert werden. Diesen Zusammenhang stellt Abbildung 2.6 grafisch dar. Neben den untersuchten Formaten lässt sich das gemeinsame (Meta-)Modell `Salt` , da es modellbasiert entwickelt wurde, in diese Ebenen einordnen und wird ebenfalls in Abbildung 2.6 dargestellt.

[1] in dieser Arbeit die Technik ECore, die ich in Abschnitt 2.4 vorstellen werde

In der modellbasierten Entwicklung werden Abbildungen zwischen unterschiedlichen Modellen als Transformationen bezeichnet.
In [Mens and Gorp, 2005] werden die Begriffe vertikale und horizontale Transformation eingeführt.
„A horizontal transformation is a transformation where the source and target models reside at the same abstraction level. ... A vertical transformation is a transformation where the source and target models reside at different abstraction levels.“ (Mens et al, 2005, S. 8 ([Mens and Gorp, 2005])).
Entsprechend des 4-Ebenen-Modells ist eine vertikale Transformation eine Abbildungen von einer Ebene M_i zu einer einer Ebene M_{i+1} oder andersherum. Eine Transformation auf gleicher Ebene ist eine horizontale Transformation. In der Literatur findet man auch die Begriffe M2M- (Model-2-Model) und M2T- (Model-2-Text) Transformationen, siehe [OMG, 2008] bzw. [Eclipse Foundation, 2009d]. Diese bezeichnen im Grunde nichts anderes, nur dass der Begriff Text bei den M2T-Transformationen oft weit ausgelegt wird. So kann eine M2T-Transformation beispielsweise aus einem Metamodell Code erstellen oder auch auf ein Persistenzformat abbilden. Im Rahmen dieser Arbeit betrachte ich auch die Ebenen, auf denen sich Format und Modell befinden, als unterschiedliche Abstraktionsebenen, daher kann die Persistenzierung eines Modells in einer Datei eines Formates als vertikale Transformation bezeichnet werden. Abbildung 2.6 zeigt die Arten der Transformationen in Bezug auf Modell und Format.

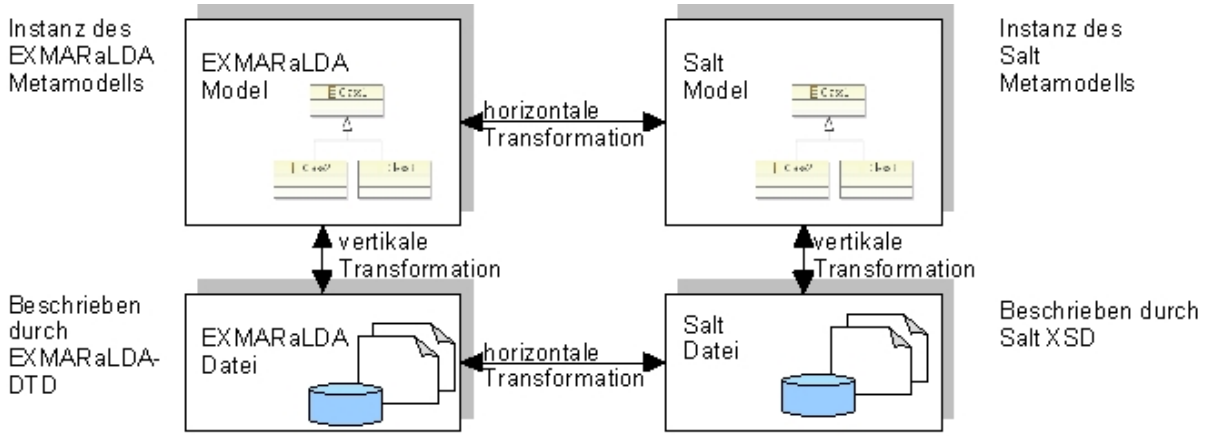


Abbildung 2.6.: Modell von EXMARaLDA und `Salt` sowie entsprechende Speicherungen der Modelle als Dateien. Zwischen den Dateien und Modellen sowie untereinander ist jeweils die Art der Transformation angegeben, um das eine in das andere zu überführen.

Bei der Vorstellung des Konverterframeworks in Kapitel 5 werde ich auf die Modell- und Formatebene, sowie auf die Transformationen zwischen diesen zurückkommen.
Als wichtiger Bereich der modellbasierten Entwicklung sei hier die **M**odel **D**riven **A**rchitecture (MDA [Miller and Mukerji, 2003]) der **O**bject **M**anagment **G**roup (OMG [OMG, 2009a]) genannt. Unter dem Sammelbegriff MDA werden viele Techniken zur modellbasierten Entwicklung vereint. So auch die in dieser Arbeit verwendete Technik QVT [OMG, 2007],

die ich in Abschnitt 2.5 vorstellen werde. Im Anhang in Abschnitt A.1 werde ich einige wesentliche Bestandteile der MDA beschreiben.

2.3.1. Vorteile der modellbasierten Entwicklung

Nachdem ich die modellbasierte Entwicklung beschrieben habe, werde ich nun explizit ein paar Vorteile nennen, die sich durch deren Nutzen im Bezug auf die vorliegende Arbeit ergeben. An einigen Stellen werde ich auf die Nachteile der formatbasierten Entwicklung eingehen, die durch den modellbasierten Ansatz gelöst werden.
Ein Metamodell dient nicht nur dem Entwickler als Dokumentation, sondern auch als Kommunikationsschnittstelle der beteiligten Akteure, wie Entwicklern und Spezialisten der betreffenden Domäne. Anhand eines Metamodells können die Akteure in mit einer formalen Beschreibung wie bspw. `Salt` überprüfen, ob ihre Ansprüche berücksichtigt wurden. Spezialisten der technischen Domäne werden in die Lage versetzt, sich bspw. mit der Verarbeitbarkeit, der Performanz und der Persistenzierung auseinander zu setzen, während die Experten der linguistischen Domäne sich auf die Frage konzentrieren können, ob alle semantischen Anforderungen in dem Metamodell dargestellt werden. Abbildung 2.7 zeigt ein Metamodell als Schnittstellenbeschreibung der unterschiedlichen Domänen.

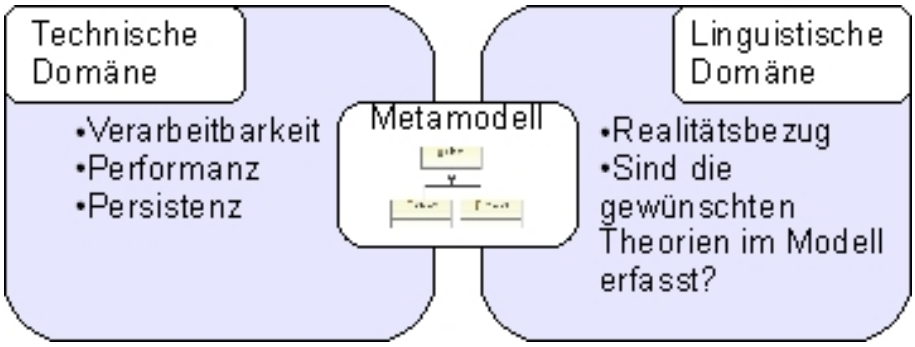


Abbildung 2.7.: Metamodell als Schnittstellenbeschreibung der unterschiedlichen Domänen

Für diesen Gedanken ist es erforderlich, dass die beteiligten Akteure die Sprache des formalen Metamodells verstehen. Die „Object Managemant Group“, kurz OMG (siehe [OMG, 2009a]), schlägt hier einige Beschreibungssprachen für unterschiedliche Perspektiven vor, die in der „Unified Modeling Language“, kurz *UML* (siehe [OMG, 2009b]), zusammengefasst sind. Der bekannteste Vertreter ist das „Klassenmodell“. Ein in dieser Arbeit verwendetes Pendant ist ECore [Steinberg et al., 2009], das ich in Abschnitt 2.4 vorstellen werde. Auch wenn die Syntax einer solchen Sprache eine Einstiegshürde darstellen kann, so ist sie dennoch geringer als bei der formatbasierten Entwicklung, bei der sich die Akteure in einer Persistenzbeschreibungssprache wie *DTD*, *XSD*, *SQL* unterhalten müssen.
Um die Lücke zwischen Metamodell und Code zu schließen, existieren Möglichkeiten, um aus einem Metamodell mehr oder weniger automatisch Code zu erstellen. Für diese Arbeit bietet die Metamodellerzeugung für die untersuchten Formate den Vorteil, dass eine *API* erzeugt werden kann, die ich für die späteren Konvertierungen genutzt habe

(auf diesen Vorteil werde ich in Abschnitt 3.1 erneut eingehen). Einige Frameworks bieten bereits Lösungen an, um eine Mischung von Programmierung und Modellierung zu erreichen. Eines dieser Frameworks ist *EMF*, das ich in Abschnitt 2.4 vorstellen werde. Dieser Ansatz fehlt weitestgehend bei der formatbasierten Entwicklung.
Die Abstraktion eines Metamodells und die damit verbundene Unabhängigkeit zwischen Modell und Persistenzierung führt dazu, dass für ein Modell verschiedene Persistenzierungsformen erstellt werden können. Deshalb ist es möglich, mehrere Persistenzierungen für ein Modell vorzunehmen. Beispielsweise kann dadurch das gemeinsame (Meta-)Modell , bspw. in *XML*, aber auch in anderen technischen Formaten, gespeichert werden. Damit entfällt - im Gegensatz zu der formatbasierten Entwicklung - die Abhängigkeit von einer bestimmten Technik. Für die in Abschnitt 5.2 vorgestellte diagonale Transformation ist dies die Grundlage.
In der modellbasierten Entwicklung kann ein Metamodell Grundlage eines weiteren Metamodells sein. Dadurch ist es möglich, dass Metamodelle aufeinander aufbauen und die Funktionen eines bereits existierenden Metamodells übernehmen. Dieses Verfahren wird als Verfeinerung oder Ableitung bezeichnet. Eine derartige Wiederverwendbarkeit und Modularisierung ist durch die formatbasierte Entwicklung nicht möglich. Durch Verfeinerung entstehen mitunter lange Verfeinerungsketten, die auf der einen Seite zwar die Lesbarkeit erschweren, auf der anderen Seite aber dazu führen, dass ein Metamodell auf der Mächtigkeit eines anderen aufbauen kann. Für die Entwicklung von `Salt` in Kapitel 4 habe ich dieses Verfahren verwendet.
Der Einsatz der modellbasierten Entwicklung führt jedoch auch zu einigen Nachteilen. Beispielsweise kann sich die Wiederverwendung von Metamodellen durch Verfeinerungen nachteilig auf die Performanz auswirken. Da die Performanz bei der später vorgestellten Konvertierung von Korpora für den praktischen Einsatz nicht unerheblich ist, werde ich den Zusammenhang zwischen modellbasierter Entwicklung und Performanz im folgenden Abschnitt 2.3.2 an einem Beispiel kurz beschreiben.

2.3.2. modellbasierte Entwicklung und Performanz

In der modellbasierten Entwicklung ist ein vorherrschendes Prinzip das von „Divide and Conquer“ (zu deutsch: teile und herrsche). Wiederverwendbarkeit spielt eine zentrale Rolle. Aus diesem Grund werden Probleme aufgespalten, nach Ähnlichkeit zu *Modulen* oder Objekten zusammengefasst und dann wieder miteinander verknüpft. An dieser Methodik ist zu erkennen, dass der Grad der Modularisierung und die nachhaltige Verwendbarkeit einzelner Komponenten zusammenhängt. Je unspezifischer ein Modul ist, desto häufiger ist es einsetzbar. Modularisierung ist auch ein wichtiger Teil der Modellierung, da Programme und Metamodelle immer komplexer werden und deren Entwicklung immer mehr Zeit in Anspruch nehmen würde, wenn man nicht ständig auf bereits existierende *Module* zurückgreifen und diese wiederverwenden könnte.
Leider lassen sich Performanz und feingranulare Modularisierung nicht immer miteinander vereinbaren. Oft sind sie sogar gegenläufig. Angenommen, für einen beliebigen

Graphen soll die minimale Färbbarkeit[2] berechnet werden. Zusätzlich soll er persistenziert werden. Weiter angenommen, zu diesem Zweck gibt es bereits ein *Modul*, das die Färbbarkeit berechnet und eines, das den Graphen persistenziert. Um das Rad nicht neu zu erfinden und davon ausgehend, dass beide *Module* schon fehlergeprüft sind, wäre es bezogen auf die Entwicklungszeit vorteilhaft, beide *Module* konkateniert einzusetzen. Aus Sicht der Performanz ergeben sich dadurch Nachteile, da der Graph sowohl zur Berechnung der Färbbarkeit als auch zur Serialisierung *traversiert* werden muss. Anstatt beides in einer *Traversierung* zu erledigen, wie es bei einer Neuentwicklung beider Module in einer Komponente der Fall wäre, werden nun zwei *Traversierungen* benötigt.
Ein weiterer Performanznachteil durch die modellbasierte Entwicklung kann in den einzelnen Frameworks zur Realisierung liegen. Bspw. erzeugt das hier genutzte EMF, auf das ich im nächsten Abschnitt (siehe 2.4) eingehen werde, bei der Codeerzeugung eine beachtenswerte Menge an Overhead. Es werden viele zusätzliche Methoden und Attribute erzeugt, um einen allgemeinen Zugriff auf Objekte zu ermöglichen. Diese sind jedoch im Einzelfall oft gar nicht notwendig. Sie dienen u.A. anderen modellbasierten Verfahren wie bspw. der QVT, auf die ich ebenfalls noch eingehen werde (siehe Abschnitt 2.5).
In EMF werden doppelte Verkettungen und Benachrichtigungen häufig genutzt. Dies führt ebenfalls zu einer längeren Verarbeitungszeit. So stößt beispielsweise das Ändern eines Attributes eine ganze Reihe von Benachrichtigungen an, um allen Objekten, die an dieser Veränderung interessiert sein könnten, mitzuteilen, dass sie stattgefunden hat. Diese Benachrichtigungskette, die bei Unachtsamkeit in der Entwicklung auch zu Zyklen führen kann, kann zu einem relevanten Zeitfaktor werden.
Derartige performanzbeeinflussende Faktoren dürfen bei der modellbasierten Entwicklung nicht außer Acht gelassen werden und spielen für `Salt` eine wichtige Rolle. Auf eine Möglichkeit, die Performanz in Metamodellen zu beeinflussen, werde ich in Abschnitt 4.1 eingehen.

2.4. EMF

Die modellbasierte Entwicklung beschreibt nur, wie Software mit Metamodellen entwickelt werden kann. Um die Konzepte der modellbasierten Entwicklung umzusetzen, werden Techniken benötigt, mit denen Metamodelle erzeugt und verarbeitet werden können. Eine Sammlung verschiedener Techniken ist das Framework EMF [Eclipse Foundation, 2009a], das ich in diesem Abschnitt vorstellen werde. Im Rahmen dieser Arbeit habe ich EMF verwendet, um zum einen das gemeinsame (Meta-)Modell `Salt` zu erzeugen und dafür eine *API* zu generieren. Zum anderen habe ich einige der in Kapitel 3 beschriebenen Formate ebenfalls in ein Metamodell überführt, um dann mit EMF eine *API* für diese Formate zu generieren. Das war für die Formate, die keine *API* besitzen, nötig, um Mappings für `Pepper` zu entwickeln.
EMF steht für **E**clipse **M**odeling **F**ramework ([Eclipse Foundation, 2009a] und [Steinberg et al., 2009]) und wurde von der Eclipse Foundation (siehe http://www.eclipse.-

[2] Jedem Knoten wird eine Farbe zugewiesen. Dabei ist zu beachten, dass kein Knoten die gleiche Farbe tragen darf wie einer seiner Nachbarknoten. Berechnet werden soll die minimale Anzahl benötigter Farben.

org/) entwickelt. Das *Framework* realisiert viele Konzepte der modellbasierten Softwareentwicklung. Es besteht aus einer Kernmodellsprache namens ECore und einer Reihe von verschiedenen *Frameworks* zur Einbettung anderer Techniken, wie z.B. Teneo zur Speicherung von Modellen in Datenbanken [Project, 2009]. Für EMF gibt es einige Eclipse-*IDE* PlugIns, die eine gute Einbettung in die Entwicklungsumgebung realisieren. Dazu gehört die grafische Erzeugung eines Metamodells und die automatische Codegenerierung aus diesem Metamodell in die Programmiersprache Java. ECore ist eine der UML sehr nahe Sprache, beschränkt sich aber auf einen kleineren Modellumfang und eine geringere Menge an Diagrammen. ECore ist kompatibel zu eMof[3], einer Sprache zur Entwicklung von Metametamodellen. Im Gegensatz zur UML, die im Grunde allen Programmiersprachen genügen soll, wurde EMF hauptsächlich für Java entwickelt. Mittlerweile existieren aber auch Generatoren, die Code in C++ bzw. C# erzeugen. EMF bietet Möglichkeiten der Einbettung von M2T [OMG, 2008], um so mit verschiedenen Techniken Generatoren zu erzeugen, die ein in ECore modelliertes Metamodell in eine andere Sprache übersetzen. Die UML und ECore verfügen über kompatible Kerne und sind über eine XMI-Persistenzierung aufeinander abbildbar. XMI ist ein Serialisierungsstandard der OMG, um sicherzustellen, dass Metamodelle von verschiedenen Werkzeugen bearbeitet werden können.
Die Philosophie von EMF ist nicht das von der Programmierung getrennte Modellieren, sondern die Verknüpfung von beidem. Modellierung und Programmierung sollen sich in einem Kreisprozess immer wieder ablösen. Diesen Prozess habe ich in Abschnitt A.2 im Anhang beschrieben.
An einigen Stellen in dieser Arbeit werde ich zur Veranschaulichung Metamodelle grafisch darstellen. Für diese Darstellungen werde ich die grafische Notation verwenden, die ECore bereitstellt. Diese Notation ist sehr ähnlich zur grafischen Darstellung von UML-Elementen und ich werde sie an dieser Stelle exemplarisch vorstellen. Der Einfachheit halber beschränke ich mich bei der Darstellung auf die Elemente, die ich auch innerhalb dieser Arbeit verwendet habe.

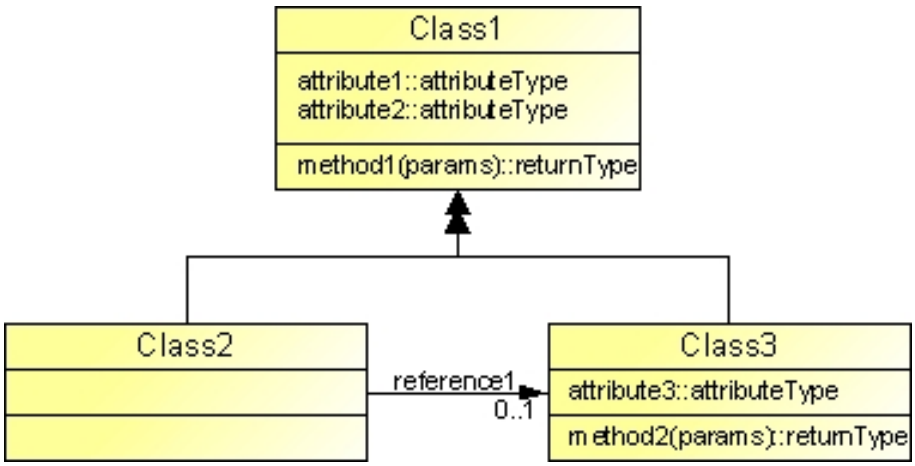


Abbildung 2.8.: Notation der verwendeten Elemente eines Metamodells.

[3]eMof ist die von der OMG entwickelte Kernsprache der MDA [OMG, 2002]

Abbildung 2.8 zeigt ein Metamodell mit den Metamodellelementen Klasse, Referenz, Vererbung, Attribut und Operation. Eine Klasse wird als Rechteck bestehend aus drei kleineren Rechtecken dargestellt. Die hier dargestellten Klassen sind „Class1“, „Class2“ und „Class3“. Eine Klasse enthält im oberen Rechteck ihren Namen, darunter die Attribute, die zu dieser Klasse gehören und zuunterst die von der Klasse bereitgestellten Methoden. Attribute besitzen ebenfalls einen Namen und einen Datentyp. Im Fall von „Class1“ gibt es zwei Attribute mit den Namen „attribute1“ und „attribute2“. Hinter den beiden Doppelpunkten folgt der Datentyp des Attributes, hier als „attributeType“ bezeichnet. Attributtypen können sehr unterschiedlich sein, sie können primitive Datentypen wie Strings (Zeichenketten), numerische Werte, Wahrheitswerte etc. oder komplexe Typen wie bspw. andere Klassen sein. Eine Operation oder auch Methode besitzt einen Namen (im Falle von „Class1“ gibt es eine Methode mit dem Namen „methode1“) eine Menge an Parametern („params“) und einen Rückgabetyp („returnType“). Die Menge der Parameter besteht aus einzelnen Attributen, der Rückgabetyp ist ein primitiver oder komplexer Datentyp. Zwischen den Klassen „Class2“ und „Class3“ existiert eine Referenz mit dem Namen „reference1“. Diese bedeutet, dass Objekte der Klasse „Class2“ über die Referenz auf Objekte der Klasse „Class2“ zugreifen können. Ein Zugriff auf das Attribut „attribute3“ aus der Klasse „Class2“ heraus würde wie folgt aussehen: „reference1.attribute3“. Am rechten Ende der Referenz ist ein Intervall („0..1“) angegeben. Diese Angabe wird als Kardinalität bezeichnet und drückt aus, dass jedes Objekt der Klasse „Class2“ entweder ein oder gar kein Objekt der Klasse „Class3“ referenzieren kann. Übliche Kardinalitäten sind: „0..1“ (keine oder eine Referenz), „1..1“ (genau eine Referenz), „0..*“ (keine oder mehrere Referenzen) und „1..*“ (mindestens eine oder mehrere Referenzen). Zwischen den Klassen „Class2“ und „Class1“ sowie zwischen „Class3“ und „Class1“ besteht eine Vererbungsbeziehung[4]. Das bedeutet, dass die Klassen „Class2“ und „Class3“ alle Eigenschaften (Attribute und Methoden) „erben“ und somit ebenfalls besitzen. Z.B. besitzt die Klasse „Class2“, obwohl sie selbst keine Attribute und Methoden definiert, über die Vererbungsbeziehung die Attribute „attribute1“ und „attribute2“ sowie die Methode „method1“. Für eine genauere Beschreibung der ECore Metamodellelemente verweise ich auf [Steinberg et al., 2009].
Ein erzeugtes Metamodell mitsamt seinem generierten und ausprogrammierten Code kann genutzt werden, um als Grundlage weiterer modellbasierter Werkzeuge zu dienen. Beispielsweise können darauf QVT-Transformationen, die ich in Abschnitt 2.5 beschreiben werde, aufbauen.

2.5. QVT

QVT (siehe [OMG, 2007]) ist eine Modelltransformationssprache. Diese habe ich im Rahmen dieser Arbeit dazu verwendet, Mappings von einem Modell auf ein anderes zu ermöglichen. QVT steht für **Q**uery **V**iew **T**ransformation und wurde im Rahmen der MDA ([Miller and Mukerji, 2003]) entwickelt. In Abbildung 2.6 aus Abschnitt 2.3 habe ich kurz angerissen, wie Transformationen in dieser Arbeit zwischen Modellen

[4]diese Notation ist weder ECore- noch UML-konform

und Formaten zu verstehen sind. Mit QVT kann eine Transformation auf Modellebene zwischen einem Quellmodell und einem Zielmodell erzeugt werden. QVT besitzt einen imperativen (QVT-Operational [Nolte, 2009a]) und einen deklarativen (QVT-Relations [Nolte, 2009b]) Teil. Deklarativ bedeutet, dass ein Mapping nur beschreibend vorgenommen wird. Es wird beschrieben, welche Elemente aus Metamodell *A* auf welche Elemente aus Metamodell *B* zu mappen sind. Wann dies innerhalb eines Mappingprozesses geschieht, kann dabei nicht bestimmt werden und wird der entsprechenden Implementierung überlassen. Ein analoger Ansatz findet sich bei der XML-Transformationssprache XSLT [Clark, 1999]. Beide unterscheiden sich dadurch, dass XSLT durch die Bindung an *XML* technik-abhängig ist, wohingegen QVT auf Modellebene arbeitet und damit technik-unabhängig ist. Eine genauere Beschreibung von QVT und eine Erklärung, wie ein deklaratives Mapping funktioniert, ist Abschnitt A.3 des Anhangs zu entnehmen.
Der imperative Teil von QVT ermöglicht neben der Beschreibung einer Transformation auch einen tieferen Eingriff in die Ausführungsreihenfolge. Dadurch kann ein Mapping bezogen auf nicht-funktionale Funktionen wie die Performanz beeinflusst werden. Diese Art der Transformation habe ich in dieser Arbeit jedoch nicht untersucht. In dieser Arbeit habe ich QVT-Relations den Vorzug gegeben, da dieser Ansatz einen leichteren Einstieg bietet und der linguistischen Domäne dadurch wahrscheinlich zugänglicher ist.
Zur Nutzung von QVT-Relations gibt es bisher nicht viele Implementationen. Ich habe Medini QVT (siehe [ikv++ technologies ag, 2009]) der Firma IKV verwendet. Medini QVT ist für nicht kommerzielle Zwecke frei verfügbar und kann im Kern in jedes Programm integriert werden. Zusätzlich gibt es für Medini eine auf Eclipse basierende Entwicklungsumgebung. Diese kann aber auch als Eclipse PlugIn in eine bestehende Eclipse-Umgebung integriert werden. Medini QVT kann Modelle in Form von ECore verarbeiten und die entsprechenden in Java erstellbaren *APIs* für die Transformation verwenden.

2.6. Technologischer Raum und Brückentechnik

In [Kurtev et al., 2002] wird der Begriff des „technological space“ (zu deutsch „Technologischer Raum“) eingeführt. Mit einem Technologischen Raum (kurz: TS) ist dabei der Kern einer bestimmten Technologie gemeint. Zu diesem Kern gehört der Sprachumfang, die Umsetzung, eine Community und Werkzeuge bzw. Techniken zur Verarbeitung. Technologische Räume sind demnach bspw. die MDA, bestimmte Programmiersprachen (wie Java, Perl etc.), XML oder auch RDBMS. Den Begriff des TS werde ich in dieser Arbeit verwenden, um verschiedene Technologien voneinander abzugrenzen.
Weiter wird in [Kurtev et al., 2002] von „bridging spaces“ gesprochen. Dies sind Techniken zwischen zwei oder mehreren TS, die zu deren Verknüpfung dienen und Technologische Räume anderen Technologischen Räumen zugänglich machen. Im Deutschen ist der Begriff Brückentechniken gebräuchlich. Beispiele für Brückentechniken sind SAX und DOM zur Brückenbildung von XML nach Java, oder auch JDBC zur Verknüpfung von RDBMS mit Java.

2.7. OSGi

In diesem Abschnitt stelle ich ein PlugIn-Framework vor, das es ermöglicht, Software nicht nur schematisch, sondern auch technisch zu modularisieren. Erweiterbarkeit durch einen PlugIn-Mechanismus ist eine wichtige Eigenschaft des in dieser Arbeit entwickelten Konverterframeworks. Wie ich später in Abschnitt 5.1.3 ausführen werde, wächst die Mächtigkeit eines Konverterframeworks mit der Anzahl der integrierten Mappings. Es ist daher wichtig, weitere Mappings durch einen PlugIn-Mechanismus in das Konverterframework zu integrieren. Zur Umsetzung habe ich das Framework OSGi verwendet.
OSGi wurde von der OSGi Alliance (siehe [OSGiAlliance, 2009]), die sich früher **O**pen **S**ervices **G**ateway **i**nitiative nannte, entwickelt. Zu ihr gehören eine Reihe namhafter Unternehmen wie Ericsson, IBM, Oracle und Sun Mircrosystems.
OSGi ist die Spezifizierung eines komponentenbasierten Frameworks zur Entwicklung von Software in Java. Es wurde entwickelt, um monolithische Software in Komponenten aufzuteilen und diese lose miteinander zu koppeln. Dadurch soll eine hohe Modularisierung der Software und eine Wiederverwendbarkeit einzelner Komponenten entstehen. Ein Softwareprodukt wird so zu einer Menge einzelner, in Beziehung gesetzter, Komponenten. Diese Komponenten können dynamisch während der Laufzeit der Software hinzugefügt oder entfernt werden, ohne dass der Ausführungsprozess dabei angehalten werden muss. OSGi definiert hierfür eine Laufzeitumgebung, die *Service Platform*, die die Verwaltung der einzelnen Komponenten übernimmt. Komponenten werden in OSGi als „Bundles“ bezeichnet.
OSGi stellt jedoch nur eine Spezifikation dar und beinhaltet keine Implementierung. Für die vorliegende Arbeit habe ich die OSGi Implementierung Equinox [Wütherich et al., 2008] verwendet. Equinox ist eine Entwicklung der Eclipse Foundation und inzwischen die Grundlage des PlugIn-Systems der Eclipse IDE.
Eine detailiertere Beschreibung von OSGi und Equinox findet sich im Anhang in Abschnitt A.4.

2.8. ISOCat

Bei der Verarbeitung von Korpora aus Formaten entsteht an einigen Stellen ein Problem bei der Deutung der Daten. Einige Formate gehen dabei implizit von bestimmten Deutungen aus wie ich in Abschnitt 3.1 anhand von TreeTagger [Schmid, 1994] oder Tiger [Stuttgart, 2003] zeigen werde. Andere besitzen diese Art der Deutung nicht. Als Beispiel sei die Annotation pos=“VVFIN“ genannt. Einige Formate enthalten das implizite Wissen, dass es sich dabei um eine morphologische part-of-speach-Annotation handelt, andere erkennen nur ein Attribut-Wert-Paar. Ein weiteres Problem ist der Definitionsbereich. Es gibt weder eine einheitliche Schreibweise, noch ein einheitliches Verständnis bestimmter Kategorien. Eine part-of-speach-Annotation kann bspw. den Namen „POS“, „pos“, „Pos“ uvm. erhalten. Ebenso könnte der Annotationsname „pos“ auch für Position stehen. In Abschnitt 3.3.13 werde ich beschrieben, wie dadurch Informationsverluste entstehen können.
Um die Deutungslücke zwischen der syntaktischen Darstellung von Kategorien als Attribut-

Wert-Paare und deren Interpretation zu schließen, wurde das Werkzeug ISOCat (siehe [ISO-TC-37, 2009]) entwickelt.
ISOCat ist ein Framework zur Spezifizierung von Kategorien (data categories).

> [...] data category specifications in the DCR contain linguistic descriptions, such as data category definitions, statements of associated value domains, and examples. Data category specifications can be associated with a variety of data element names and with language-specific versions of definitions, names, value domains and other attributes.
>
> ([ISO-TC-37, 2009], zuletzt besucht am 17.08.09)

ISOCat bietet eine Plattform zur Definition von Kategorien und Werten. Diese werden jedoch nicht als Attribut-Wert-Paare betrachtet, sondern als alleinstehende Datenpunkte. Datenpunkte können mit weiteren Informationen und Constraints, wann und wo sie einsetzbar sind, angereichert werden. Zur Identifizierung der Datenpunkte wird eine eindeutige sog. „PID“ für jeden Eintrag vergeben. Die ID erlaubt es, die Namen von Kategorien mehrfach semantisch zu belegen und dennoch eindeutig zu bleiben. Die „PID“ wird innerhalb einer URI-Referenz aufgelöst und ist so eindeutig referenzierbar.
Der Vorteil der Nutzung von ISOCat ist die Trennung semantischer Inhalte von ihrer syntaktischen Repräsentation und der Formatdefinition. Semantiken können über URI-Referenzen von außen injiziert werden, es muss nur sichergestellt werden, dass ein entsprechendes Format diese Art der Referenzierung zulässt. Somit kann die Formatentwicklung thematisch von der Datenbedeutung getrennt werden und sogar in unterschiedlichen Domänen liegen. Die Fachspezialisten der linguistischen Domäne können die Bedeutung eines Datenpunktes klären, ohne dass die Formatentwicklung davon betroffen ist. Diese kann so bspw. in die informatische Domäne verlegt werden. Außerdem ist so eine einheitliche Interpretation eines Datenpunktes durch seine zentrale Definition sichergestellt. Es können neue Kategorien hinzugefügt und Definitionen bestehender Kategorien verändert werden, ohne dass die Formatentwicklung darauf reagieren muss.
Da ISOCat allerdings noch eine sehr junge Entwicklung ist, mangelt es dieser Plattform noch an Verbreitung und Inhalt. Es gibt bisher erst relativ wenig Datenpunkte, aber die Menge wird stetig erweitert. In einem Projekt der Universität Stuttgart wird derzeit das STTS in ISOCat eingefügt.
Für das hier entwickelte gemeinsame (Meta-)Modell habe ich vorerst eine Erweiterung entwickelt, um verarbeitenden Werkzeugen eine Deutungs zu ermöglichen. Diese werde ich in Form des Metamdolls `SaltSemanticsMM` in Abschnitt 4.4 vorstellen.

3. Untersuchung linguistischer Annotationsformate

In diesem Kapitel werde ich die Merkmale einer Menge an linguistischen Annotationsformaten untersuchen und darlegen. Die untersuchten Formate sind: das Treetagger Format, das Tiger Format, das EXMARaLDA Format, das PAULA Format und das relANNIS Format. Zunächst werde ich die einzelnen Formate beschreiben und auf ihre spezifischen Eigenschaften eingehen. Ich werde aus den Eigenschaften der einzelnen Formate formatübergreifende Konzepte entwickeln. Die Konzepte sollen als Grundlage dienen, um die Formate, bezogen auf ihre Mächtigkeit, miteinander zu vergleichen. Gleichzeitig werden diese Konzepte im nächsten Kapitel als Grundlage bei der Entwicklung des gemeinsamen (Meta-)Modells dienen.

3.1. Formatvorstellung

In diesem Abschnitt werde ich fünf Formate vorstellen, die ich im Rahmen dieser Arbeit betrachtet habe um anhand von diesen das gemeinsame (Meta-)Modell zu entwickeln. Zu deren Beschreibung werde ich erklären, durch wen und zu welchem Zweck sie entwickelt wurden. Im Anschluss daran werde ich an einem Metamodell erklären, welchem Zweck die einzelnen Bestandteile der Formate dienen.

3.1.1. TreeTagger-Format

TreeTagger ist im eigentlichen Sinne weniger ein Format, als vielmehr ein Werkzeug zur automatischen Anreicherung eines Textes mit Lemma- und part-of-speech-Annotationen. TreeTagger wurde an der Universität Stuttgart von Helmut Schmidt (siehe [Schmid, 1994]) entwickelt. Als TreeTagger-Format bezeichne ich hier die Ausgabe dieses Werkzeugs. Gleichzeitig kann diese Ausgabe als Eingabe für weitere Werkzeuge wie bspw. CQP (siehe [O., 1994]) genutzt werden. Im Gegensatz zu den meisten hier untersuchten Formaten, die als Austauschformat genutzt werden können, existiert für Treetagger keine formale Schemadefinition. Ich habe dieses Format in die Menge der untersuchten Formate aufgenommen, um zu zeigen, dass es ebenfalls möglich sein soll, derartige Formate in den Konvertierungsprozess einzubinden.
Treetager kann anhand von Trainingskorpora auf verschiedene Sprachen trainiert werden. Meistens werden dabei, wie ursprünglich vorgesehen, Lemma- und part-of-speech-Annotationen als Trainingsdaten verwendet. Es können aber auch andere Arten von Annotationen erstellt werden, indem das Trainingskorpus mit entsprechenden Annotationen versehen wird. Die meisten Annotationen befinden sich auf der Tokenebene. Für das

word	pos	lemma
The	DT	the
TreeTagger	NP	TreeTagger
is	VBZ	be
easy	JJ	easy
to	TO	to
use	VB	use
.	SENT	.

Tabelle 3.1.: Beispiel des Treetagger-Formates. Die erste Zeile (word, pos, lemma) ist nicht Teil des Formates und dient nur der Erklärung.

<kongruenz numerus>		
The	DT	the
TreeTagger	NP	TreeTagger
is	VBZ	be
<verbalphrase VP>		
</kongruenz>		
easy	JJ	easy
to	TO	to
use	VB	use
</verbalphrase>		
.	SENT	.

Tabelle 3.2.: Beispiel aus Tabelle 3.1 erweitert um zwei Spannen

TreeTagger-Format wird i.d.R. angenommen, dass ein Token die semantische Einheit Wort darstellt.
Das TreeTagger-Format wurde als ein einfaches *CSV-Format* entwickelt, das Tokenannotationen in tabularischer Darstellung erlaubt. Abbildung 3.1 zeigt ein Beispiel des TreeTagger-Formats. Das Beispiel aus Abbildung 3.1 stammt aus http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/, zuletzt besucht am 28.07.2009.

Neben den einfachen Tokenannotationen, erlaubt das TreeTagger-Format außerdem *SGML*-Tags einzuführen. Diese werden durch das Format unterstützt, aber durch das Werkzeug ignoriert. Damit ist das TreeTagger-Format, kein einfaches *CSV*-Format mehr, sondern eine Mischung aus *SGML* und Tabellenstruktur. Durch die *SGML*-Tags, können Token zu Spannen zusammengefasst werden. Im Gegensatz zu einem *XML*-Dokument dürfen in einem *SGML*-Dokument Tags geschlossen werden, bevor alle später geöffneten Tags geschlossen wurden. Durch diesen Mechanismus lönnen in *SGML* konfligierende Spannen erzeugt werden. Abbildung 3.2 die *SGML*-Anreicherung im TreeTagger-Format.

Diese Daten können wie in Abbildung 3.1 als konfligierende Spannen interpretiert werden.

Aus der beispielhaften Modellbeschreibung, sowie der Dokumentation unter [Stuttgart, 2009], habe ich ein Metamodell für TreeTagger entwickelt. Mit diesem Meta-

Ebene	Inhalt						
kongruenz	numerus						
verbalphrase			VP				
Token	The	Treetagger	is	easy	to	use	.
POS	DT	NP	VBZ	JJ	TO	VB	SENT
lemma	the	Treetagger	be	easy	to	use	.

Abbildung 3.1.: Beispiel aus Tabelle 3.2 in Spannendarstellung

modell habe ich dann eine *API* erzeugt, um die Daten in diesem Format für die spätere Verarbeitung (durch das Konverterframework) leichter ansprechen zu können. Abbildung 3.2 stellt das Metamodell grafisch dar.

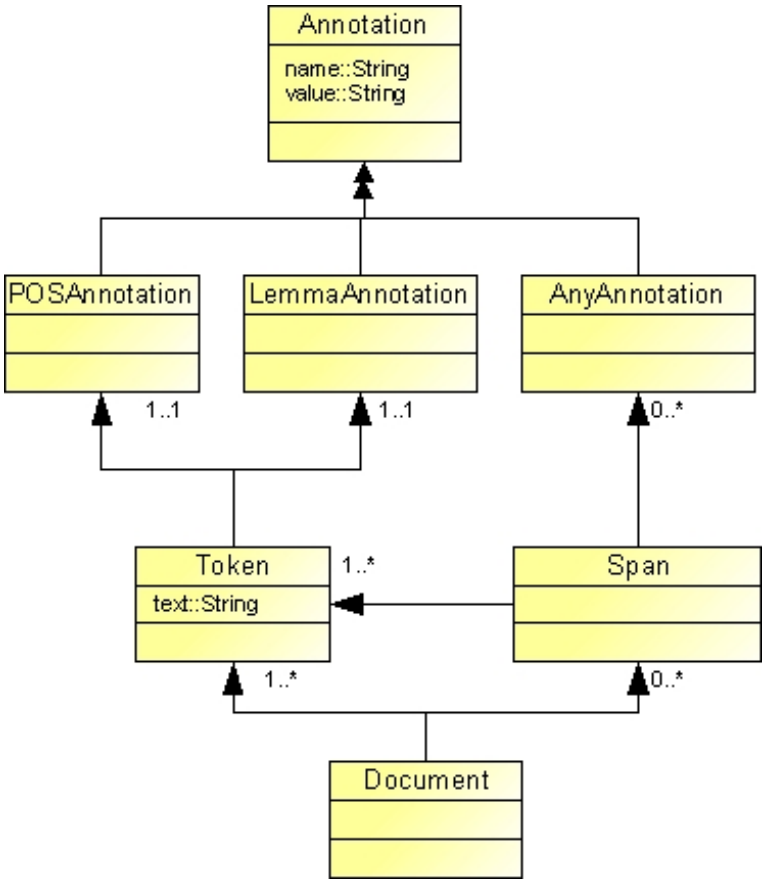


Abbildung 3.2.: Metamodell von TreeTagger

Zu sehen ist das „Dokument“-Element, dass für ein „linguistisches“ Dokument stehen soll, dieses entspricht einer Datei. Das „Dokument“-Element besitzt Referenzen zu dem „Token“- und „Span“-Element. Durch das Attribut „Token.text“ wird das Primärfragment repräsentiert. Ein Objekt des Typs „Span“ kann mehrere aufeinander folgende „To-

ken“-Objekte enthalten. Diese bilden dann eine Spanne. Das „Token“-Element besitzt Referenzen zu den Annotationen „POSAnnnotation“ und „LemmaAnnotation“. Dadurch kann ein Token part-of-speech- und Lemma-annotiert werden. Eine Spanne kann allgemein über Objekte des Typs „AnyAnnotation“ annotiert werden. Die Elemente „POSAnnotation“, „LemmaAnnotation“ und „AnyAnnotation“ sind von dem allgemeinen Typ „Annotation“ abgeleitet. Deshalb besitzt jedes Element die Attribute „name“ und „value“. Für „POSAnnotation“ und „LemmaAnnotation“ ist das „name“-Attribut schon mit den Werten „pos“ bzw. „lemma“ vorbelegt.
In den folgenden Formatvorstellungen werde ich ebenfalls grafische Darstellungen von bereits existierenden oder ebenfalls von mir entwickelten Metamodellen zeigen. An TreeTagger werde ich hier beispielhaft zeigen, wie man sich konkrete Daten in einem solchen Metamodell vorstellen kann. Abbildung 3.3 zeigt einen Teil des Beispiels aus Abbildung 3.1 in dem Metamodell für TreeTagger. In Abbildung 3.3 werden die drei „Token“-Objekte,

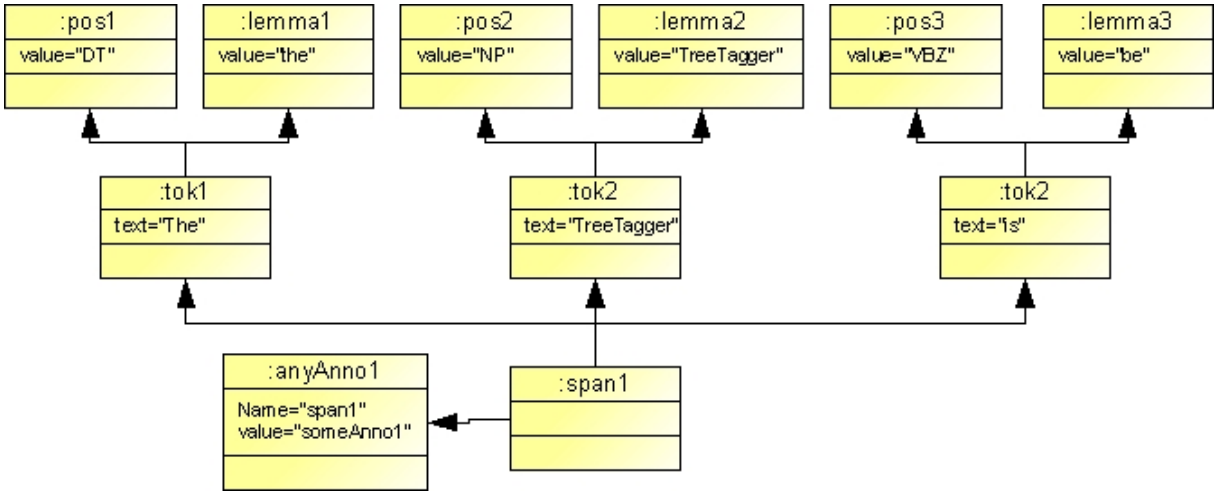


Abbildung 3.3.: Beispiel im TreeTagger Metamodell

aus Abbildung 3.2, mit den Primärfragmenten „The“, „TreeTagger“, „is“ dargestellt. Die „Token“-Objekte besitzen jeweils ein „POSAnnotation“- und ein „LemmaAnnotation“-Objekt. Weiter enthält die Abbildung ein „Span“-Objekt, dass die drei Token zu einer Spanne zusammenfasst. Diese ist mit der Annotation „span1=someAnno1“ annotiert.

3.1.2. TIGER-Format

Das Tiger-Format (siehe [Mengel and Lezius, 2000]) wurde im TiGer Projekt der Universität des Saarlandes, der Universität Saarbrücken und der Universität Potsdam entwickelt. Das TiGer Projekt (siehe [Universität-Stuttgart et al., 2007]) ist eine Art Sammelbegriff verschiedener Werkzeuge im Bereich der Korpuslinguistik. Zum TiGer Projekt gehört unter anderem das „TIGER-XML treebank encoding format“ (hier als Tiger-Format bezeichnet), eine Anfragesprache, ein Suchwerkzeug (TIGERSearch) und ein Korpus namens Tiger-Korpus.
Das Tiger-Format ist ein auf der Technik *XML* basierendes Format und wurde als „exchange format for syntactically annotated corpora“ ([Mengel and Lezius, 2000], S. 1) ent-

wickelt. Syntaktische Annotationen liegen meist als *baum-* oder *DAG*-artige Strukturen vor. Das Tiger-Format nutzt daher die durch *XML* vorgegebene Struktur zur Erzeugung von *Bäumen* und kombiniert diese mit einer nativen XML-Referenzierungsmethode (xml-id und xml-idref-Elemente (siehe [Marsh et al., 2005])). Durch diese Referenzierungsmethode lassen sich in Tiger *DAGs* erzeugen.
Für das Tiger-Format existiert eine Schemadefinition in Form mehrerer XML-Schema-Definition-Dateien (*XSD*) und eine textuelle Dokumentation unter [Stuttgart, 2003]. Da eine solche Schemabeschreibungsdatei etwas schwer zu lesen ist und Kenntnisse der Sprache *XSD* voraussetzt, habe ich versucht, das darin beschriebene Datenmodell in ein Ecore Modell zu überführen.

Bei der Modellierung des Tiger Metamodells habe ich mich weitestgehend an die Sprachsyntax des Tiger-Formates gehalten, um einen Wiedererkennungswert der einzelnen Elemente zu der *XSD* Repräsentation zu erzeugen. Funktional sind beide Darstellungen äquivalent. Abbildung 3.4 zeigt ein vereinfachtes Metamodell (Tigermetamodell) für das linguistische Modell hinter dem Tiger-Format. Ich habe bei der Modellierung technische Merkmale wie Identifizierer zur Erzeugung der Referenzen in *XML* herausgelassen, da diese durch Referenzen im Metamodell dargestellt werden.

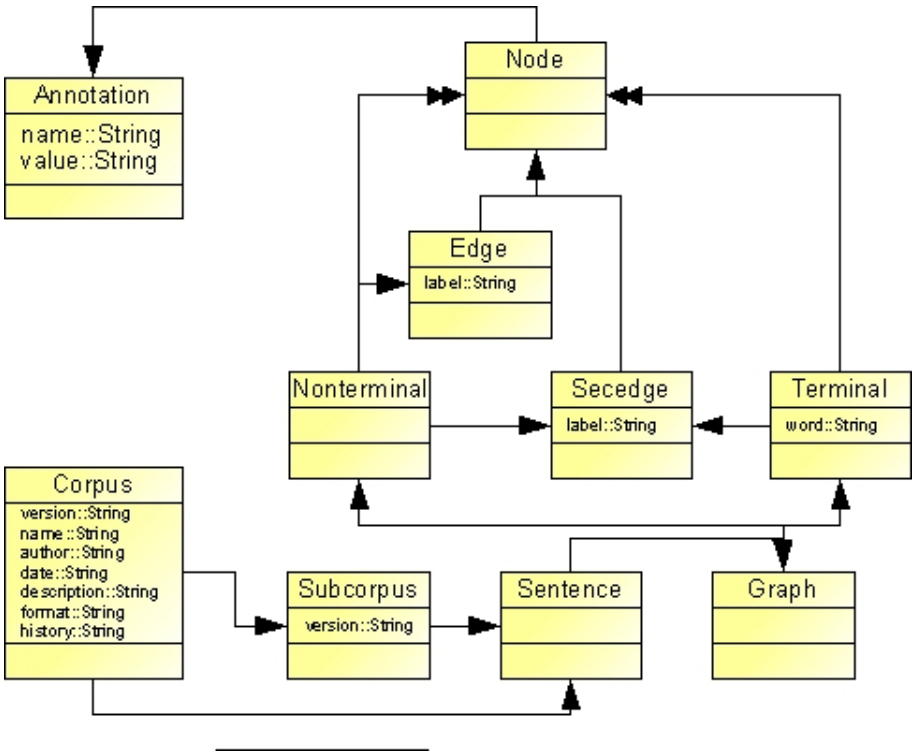


Abbildung 3.4.: Metamodell für das Tiger Format

Ein Korpus im Tigermetamodell kann in mehrere voneinander getrennte Subkorpora partitioniert werden. Dies wird in Abbildung 3.4 durch die Metamodellelemente „Corpus“, „Subcorpus“ und die Referenz zwischen beiden ausgedrückt. Das Tiger-Format ist satzbasiert, dass heißt, ein Annotationsgraph hat als Grundlage die semantische Einheit Satz. Es wird allerdings innerhalb des Formates nicht validiert, ob die Daten, die zu ei-

nem Satz gehören, wirklich dieser semantischen Einheit entsprechen. In dem Metamodell ist dieser Zusammenhang über die Beziehung der Elemente „Subkorpus“ bzw. „Korpus“, „Sentence“ und „Graph“ dargestellt. Das „Graph“-Element wiederum besitzt Referenzen zu dem „Terminal“- und dem „Nonterminal“-Element. Das „Terminal“-Element markiert einen Blattknoten innerhalb des Graphen und kann mit der strukturellen Einheit Token gleichgesetzt werden. Das „Terminal“-Element besitzt, der Dokumentation nach, das Attribut „word“, dass das überdeckte Primärfragment der Primärdaten enthält. Eine eigene Repräsentation der Primärdaten gibt es im Tiger-Format nicht. Rekursive Einheiten werden durch das „Nonterminal“-Element repräsentiert. Im Tigermetamodell werden Relationen über die beiden Kantenarten „Edge“ und „Secedge“ dargestellt. Ein „Edge“-Objekt hat im Tiger-Format als Quelle ein „Nonterminal“-Objekt und als Ziel ein „Terminal“-Objekt oder ein zweites „Nonterminal“-Objekt. Das „Edge“-Element ermöglicht die Erzeugung *baum*-artiger Strukturen im Tigermetamodell. Meist wird eine Menge von „Terminal“-Objekten zu „Nonterminal“-Objekten zusammengefasst, die eine Phrase repräsentieren. Phrasen können zu weiteren Phrasen zusammengefasst werden. Mit dem „Edge“-können im Tiger-Format *Baum*-Strukturen erzeugt werden. In manchen linguistischen Theorien und damit auch in manchen Korpora ist es erforderlich, einer strukturellen Einheit zwei Elterneinheiten zuzuweisen. Dies widerspricht jedoch der *Baum*-Struktur des „Edge“-Elements. Hierfür ist im Tiger-Format das „Secedge“-Element vorgesehen. Durch dieses können zwischen „Terminal“- und „Nonterminal“-Objekten *DAGs* erzeugt werden. Ebenfalls können diese Relationen auch zwei „Terminal“-Objekte miteinander in Beziehung setzen. Abbildung 3.5 zeigt zwei „Secedge“-Relationen.

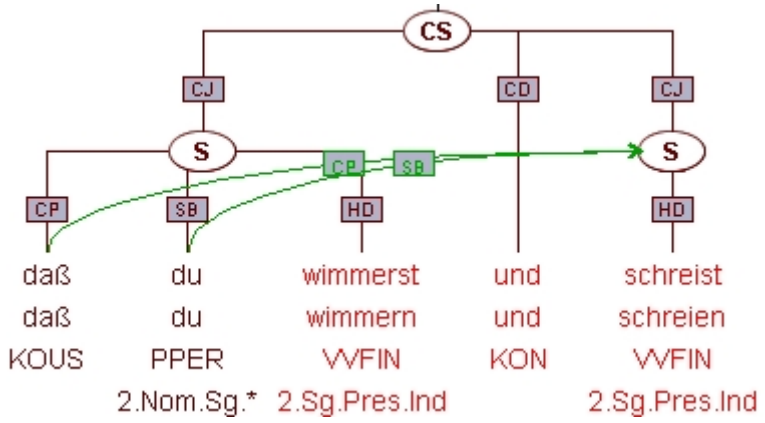


Abbildung 3.5.: Syntaxannotation aus dem Werkzeug TigerSearch. Auf der untersten Ebene stehen die „Terminal“-Objekte, darüber die „NonTerminal“-Objekte. Grüne Relationen sind vom Type „Secedge“, die restl. relationen sind vom Typ „Edge“.

Beide Relationen sind *abdeckungsvererbend*, das bedeutet, dass bspw. eine rekursive Einheit, die mit einem Token in Beziehung steht ggf. neben Anderen auch das gleiche Primärfragment beinhaltet. Das „Edge“- und das „Secedge“-Element besitzen jeweils das Attribut „label“, mit dem einer Kante eine linguistische Bedeutung gegeben werden kann. Der Wertebereich für dieses Attribut kann im Tiger-Format an anderer Stelle definiert werden. Für Token und rekursive Einheiten, können im Tiger-Format Annotationen in

Form von Attribut-Wert-Paaren vergeben werden. Deren Definitions- und Wertebereich kann ebenfalls an anderer Stelle definiert werden.
Ich habe das Tiger-Format in die Menge der betrachteten Formate mit aufgenommen, da es zum einen die Möglichkeit bietet, Hierarchien zu bilden und zum anderen eine eigene *API* besitzt. Zudem ist das Tiger-Format sehr verbreitet und es existiert eine große Anzahl an Korpora in Tiger. Andere Formate im Bereich syntaktischer Analysen sind bspw. Penn Treebank (siehe [Bies, 1995]) oder auch PML (siehe [GAAV, 2006]). Die Tiger-*API* habe ich genutzt, um das Tiger-Format in das Konverterframework zu integrieren. Daran habe ich gezeigt, dass bestehende *API*s in das *Framework* integriert werden können. Allerdings unterstützt sie nur eine Teilmenge der Möglichkeiten des Tiger-Formates. Die *API* und die unterstützte Teilmenge werde ich in Abschnitt 3.1.2.1 beschreiben.

3.1.2.1. Tiger API

Die Tiger API ist ein unter http://www.tigerapi.org/ erhältliches Open Source Projekt. Sie bietet die Möglichkeit neben anderen Formaten das Tiger-Format einzulesen und als Objektstruktur in den Hauptspeicher zu laden um ein gegebenes Modell verarbeiten zu können. Wie bereits erwähnt, unterstützt diese *API* und die dadurch gegebene Objektstruktur jedoch nicht den kompletten Sprachumfang. Ich habe auch für diese *API* ein Metamodell entwickelt, dass die Objektstruktur etwas vereinfacht darstellt. Abbildung 3.6 zeigt das Metamodell auf der Grundlage der Tiger API.

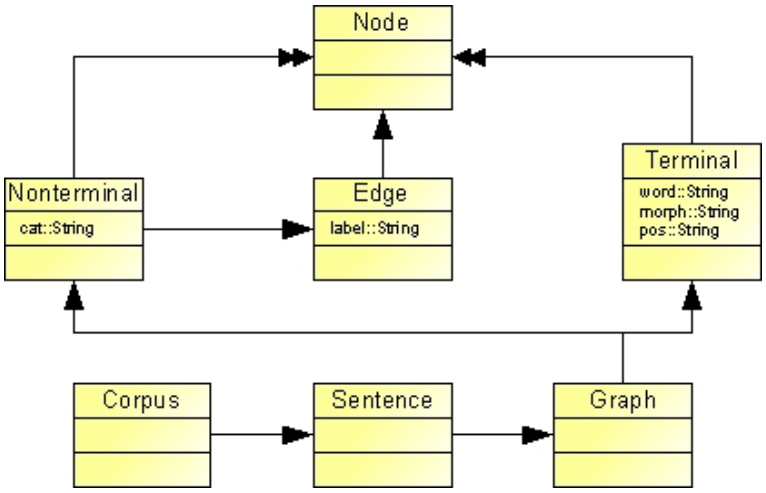


Abbildung 3.6.: Metamodell für die Tiger *API*

In diesem Metamodell fehlt generell die Möglichkeit eine Annotationen als Attribut-Wert-Paar zu speichern, es existiert nur eine vorbestimmte Menge möglicher Annotationen. Dies sind für das „Terminal“-Element die Annotationen „morph“ (für eine morphologische Annotation) und „pos“ (für eine part-of-speech Annotation) und für das „Nonterminal“-Element die Annotation „cat“ (für eine Kategorieannotation). Weiter fehlt das „Secedge“-Element für die Erzeugung von DAGs. Und es gibt keine Möglichkeit Korpora in Subkorpora zu unterteilen oder einem Korpus Metadaten zu vergeben.

3.1.3. EXMARaLDA-Format

EXMARaLDA wurde im Rahmen des SFB 538 von Thomas Schmidt (siehe [Schmidt, 2002] und http://www.exmaralda.org/) an der Universität Hamburg entwickelt. Mit EXMARaLDA wird sowohl ein Format als auch eine Sammlung von Annotationswerkzeugen wie dem Partitur-Editor, EXACT und COMA (diese Werkzeuge sind auf http://www.exmaralda.org/ zu finden) bezeichnet. Ursprünglich wurde EXMARaLDA zur Annotation von Diskursen und zur Diskursanalyse entwickelt. Allerdings bietet das Format Darstellungsmöglichkeiten an (die Spannenannotation), die in einigen Projekten für andere Zwecke genutzt wurden. Eines dieser Projekte ist FALKO (siehe [Lüdeling et al., 2008]), in dem es darum geht, geschriebene Sprache von nicht deutschen Muttersprachlern zu analysieren und u.a. auf den Fehlergehalt hin zu untersuchen. Beide Nutzungsarten sollen in dieser Arbeit berücksichtigt werden.
EXMARaLDA ist ein *XML*-Format, dass durch verschiedene *DTDs* beschrieben wird. Diese *DTDs* bilden verschiedene Aspekte des Formates ab. Es gibt *DTDs*, die die Datenintegrität sicherstellen sollen und *DTDs*, die die Anreicherung dieses Formates mit visuellen Darstellungsbeschreibungen ermöglichen. Zu Beginn dieser Arbeit habe ich bereits erwähnt, dass sie sich ausschließlich mit dem Datenmodell für linguistische Interpretationen befasst und nicht mit deren Darstellung. Daher habe ich mich hier auch nur mit einer *DTD* für die Datendarstellung („basic-transcription.dtd“) befasst. Neben der Beschreibung durch die *DTDs* findet sich unter [Schmidt, 2002] auch eine textuelle Dokumentation zu EXMARaLDA.
Wie für das Treetagger und das Tiger-Format habe ich auch für EXMARaLDA eine grafische Metamodellrepräsentation entwickelt, die ich in Abbildung 3.7 zeigen und anschließend erklären werde. Dafür habe ich mich der Sprachsyntax von EXMARaLDA bedient, um einen Wiedererkennungswert zu behalten. Im Gegensatz zu Tiger konnte ich für EXMARaLDA keine *API* finden, die das linguistische Modell von EXMARaLDA technisch ansprechbar macht. Daher habe ich aufgrund des von mir erkannten linguistischen Modells eine eigene *API* entwickelt. Diese bezieht sich auf die Abbildung 3.7.

EXMARaLDA besitzt keine Einteilung in Korpora, Subkorpora oder Dokumente. Eine EXMARaLDA-Datei kann aber als Dokument betrachtet werden. Das übergeordnete Element in EXMARaLDA ist „BasicTranscription“, dieses kann als das Element betrachtet werden, das das Dokument beschreibt. Die grundlegenden Konzepte von EXMARaLDA sind: das Spannenkonzept, das Zeitkonzept und die Tatsache, dass Spannen und Ereignisse in Form von Attribut-Wert-Paaren beliebig annotiert werden können. Ein Dokument besteht aus mehreren Schichten („Tier“-Objekte), die wiederum mehrere „Event“-Objekte beinhalten können. Ein „Event“-Objekt kann dabei entweder für eine Spanne oder für ein Token stehen. Ein „Tier“-Element kann mit vorgegebenen Annotationen, dem „Speaker“-Element, annotiert werden oder mit beliebigen Attribut-Wert-Paaren durch das „UDInformation“-Element. Eine Spanne besteht aus kleineren Einheiten, den „Event“-Elementen. Zur Realisierung des Zeitkonzeptes besitzt EXMARaLDA eine Zeitlinie („CommonTimeline“), diese beinhaltet Zeitpunkte („TLI“-Elemente), die einzelne Punkte in der Zeitlinie abbilden. Die Zeitpunkte in einer Zeitlinie sind klar geordnet, für zwei Elemente a und b gilt: $a < b$ oder $a > b$, solange $a \neq b$. Ein Ereignis in Form eines „Event“-Elements, findet in einem bestimmten Zeitabschnitt statt.

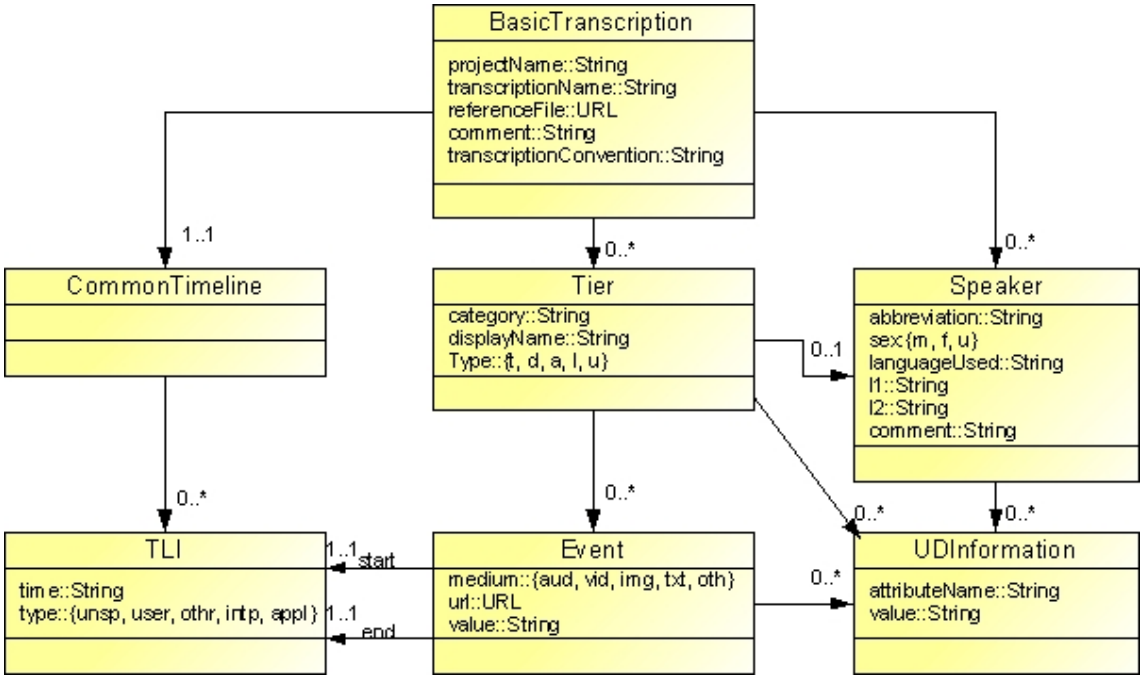


Abbildung 3.7.: EXMARaLDA Metamodell

Ein Zeitabschnitt ist ein Zeitinterval, vorgegeben durch einen Start- und einen Endzeitpunkt. Diese Verknüpfung eines Ereignisses wird durch die Referenzen „Event.start“ und „Event.end“ erreicht. Durch die klare Ordnung der „TLI“-Objekte wird durch die Referenz eines „Event“-Objekts auf zwei „TLI“-Objekte ein eindeutiges Intervall beschrieben. „Event“-Objekte können ebenfalls durch Attribut-Wert-Paare in Form von „UDInformation“-Objekten annotiert werden.
In [Schmidt, 2002] beschreibt Thomas Schmidt, dass zur Darstellung der Primärfragmente das Attribut „Event.value“ genutzt werden soll. Über das „Event“-Element wird die Zeitlinie unterteilt. Betrachtet man die Zeit parallel zu einem Text als Ressource, so kann man auch hier von einer Tokenisierung sprechen und zwar von einer Tokenisierung der Zeit. Die Spanne repräsentiert durch das „Tier“-Element ist eine Vermischung aus struktureller Einheit und Annotation, das Attribut „Tier.category“ weißt der Spanne ihre semantische Funktion zu. Nutzt man das „Tier“-Element zur Anreicherung von Zeitabschnitten mit linguistischen Informationen, so kommt dem Attribut „Event.value“ die Funktion einer Annotation zu gute. Über die gemeinsame Zeitlinie wird implizit eine Zuordnung von „Event“-Objekten erreicht, die ein Primärfragment enthalten und somit ein Token im Sinne der Texttokenisierung darstellen. In einigen Korpora, wird diese Möglichkeit auch zur Annotation genutzt. Den Annotationsnamen gibt das Attribut „Tier.category“ vor und den Annotationswert das Attribut „Event.value“. Dies ist möglicherweise nicht die Art Nutzung, die angedacht wurde, aber durchaus gebräuchlich und wird daher hier ebenfalls betrachtet. Abbildung 3.8 zeigt zwei verschiedene Nutzungsarten von EXMARaLDA, dargestellt im EXMARaLDA Partitur-Editor.

In EXMARaLDA gibt es keine Möglichkeit „Tier“-Objekte zueinander in Beziehung zu setzen. Alle Beziehungen von „Event“-Objekten sind nur implizit über die gemeinsame

	0	1
MAX	Du fällst mir immer ins	Wort.
TOM		Stimmt ja gar nicht

(a) Diskursannotation

	0	1	2	3	4	5	6	7
text	Die	Tagung	hat	mehr	Teilnehmer	als	je	zuvor
POS	ART	NN	VVFIN	PIAT	NN	KOKOM	ADV	ADV

(b) part-of-speech-Annotation

Abbildung 3.8.: oben eine Diskursannotation, dabei benennt das Attribut „Tier.category“ (die erste Spalte) den Namen der Sprecher. Die Tokenisierung ist zeitbezogen und die erste Zeile, gibt die Zeitintervalle an. Der eigentliche Tabelleninhalt ist der Primärtext bzw. die Primärtexte. Unten ist ein Beispiel aus einer part-of-spech Annotation abgebildet. Das Attribut „Tier.catgeory“ symbolisiert, dass die zweite Zeile den Primärtext angibt und die dritte Zeile die part-of-speech Annotation. Die erste Zeile gibt hier eine textbasierte Tokenisiserung an.

Zeitlinie gegeben. Es werden keine Relationen unterstützt, die „Event“- oder „Tier“-Objekte verbinden.
Ich habe EXMARaLDA in die Menge der untersuchten Formate aufgenommen, da es zum einen Spannenannotationen erlaubt und im Gegensatz zu den anderen hier untersuchten Formaten nicht einen Text als Grundlage hat, sondern die Zeit. EXMARaLDA ist ein relativ verbreitetes Format und wird derzeit von Thomas Schmidt aktiv weiterentwickelt. Ein weiteres Werkzeug bzw. Format mit ähnlichem Anwendungszweck ist bspw. ELAN (zu finden unter: http://www.lat-mpi.eu/tools/elan/).

3.1.4. PAULA-Format

PAULA steht für **P**otsdamer **Au**stauschformat für **l**inguistische **A**nnotation und wurde im Rahmen des SFB 632 an der Universität Potsdam entwickelt (siehe [Dipper, 2005]). PAULA wurde als Nachhaltigkeitsformat entwickelt und ist im Gegensatz zu den anderen hier vorgestellten Formaten nicht Teil eines linguistischen Werkzeugs. Die Intention hinter PAULA ist die Zusammenführung verschiedener Annotationsarten, wie bspw. Annotationen im Tiger-Format mit bspw. Annotationen aus dem EXMARaLDA-Format. In [Dipper, 2005] wird dies als „Multi-Level Linguistic Annotation“ (zu dt. Mehrebenenannotation) bezeichnet. PAULA hat also einen ähnlichen Ansatz wie diese Arbeit, nämlich die Erzeugung einer Struktur, die mächtig genug ist verschiedene Annotationsarten wie bspw. Spannen und rekursive Einheiten gemeinsam zu speichern. Im Gegensatz zu dieser Arbeit, ist der Ansatz von PAULA jedoch format- und nicht modellbasiert.
PAULA basiert auf der Technik *XML* und bietet zur Validierung der Datenintegrität einige *DTDs* an. Allerdings sind Möglichkeiten der Validitätsprüfung nur sehr eingeschränkt

möglich: „only very basic validation is supported. XML Schema or Relax NG would have to be used instead of DTDs to do a more fine-grained validation“ (http://www.sfb632.uni-potsdam.de/~d1/paula/doc/, zuletzt besucht am 08.10.2009). Zusätzlich zu den *DTDs* gibt es eine Dokumentation in [SFB 632, 2007]. Die Dokumentation ist an einigen Stellen ausführlicher als die *DTDs*, allerdings ist sie eher beispielbezogen als ausschöpfend beschreibend angelegt.
PAULA liegt inzwischen in verschiedenen Versionen vor, ich beziehe mich in dieser Arbeit auf die Version 1.0. Da für PAULA zum Zeitpunkt, als diese Arbeit geschrieben wurde, noch keine *API* existierte, werde ich auch hier ein Metamodell vorstellen, dass die Eigenschaften von PAULA abbilden soll. Zur Zeit wird im Rahmen des SFB 632 eine *API* entwickelt, die ein formales Metamodell für PAULA beschreibt. Dieses könnte in Zukunft als Grundlage für das PAULA-Format für das später vorgestellte Konverterframework dienen und das in Abbildung 3.9 vorgestellte proprietäre Metamodell ablösen.

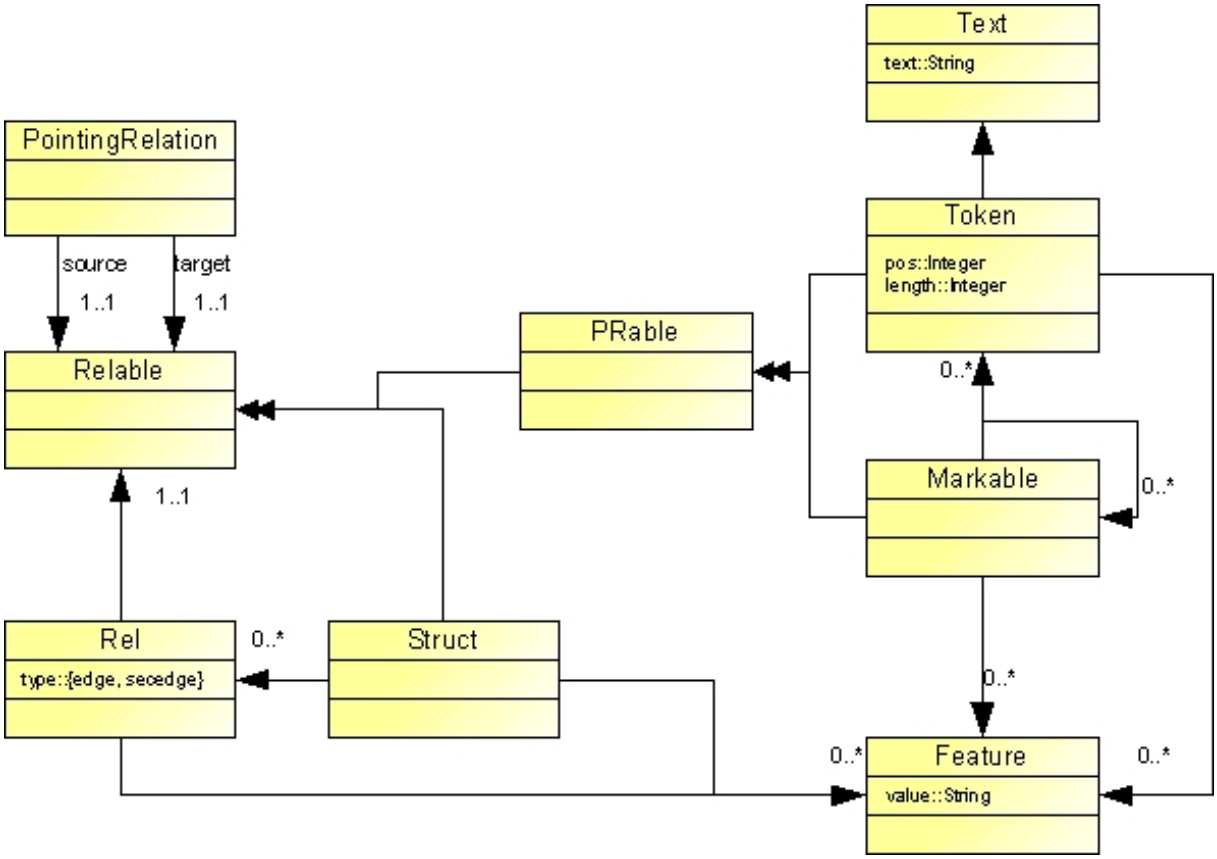


Abbildung 3.9.: Ausschnitt aus dem Metamodell des PAULA-Formats der Version 1.0. Die Elemente „Relable“ und „PRable“ gehören nicht zur PAULA-Spezifikation und dienen nur dazu, zu zeigen, welche Elemente mit Relationen („Rel“ und „PointingRelation“) in Beziehung gesetzt werden dürfen.

Der besseren Veranschaulichung halber, werde ich bei der Erläuterung des Metamodells bei den strukturellen Einheiten und Annotationen beginnen und später auf die Aufteilungsmöglichkeiten von PAULA in Korpora und Subkorpora eingehen.

PAULA besitzt mit dem Element „Text“ eine Primärtextebene, in der die digitalisierten Rohdaten gespeichert werden können. Der Primärtext wird mittels des Elements „Token“ zerlegt. Für die Zerlegung besitzt das „Token“-Element das Attribute „pos“, das die Startposition des Primärfragments im Primärtext angibt und das Attribut „length“, das die Länge des Primärfragments bestimmt. Mit dem Element „Markable“ können Spannen über den Token gebildet werden. Ein „Markable“-Objekt kann dabei mehrere „Token“-Objekte enthalten, aber auch ein weiteres „Markable“-Element. Enthält ein „Markable“-Objekt ein weiteres „Markable“-Objekt, so fungiert erstes als Platzhalter zur leichteren Annotation. Die tiefe dieser Inklusion ist maximal 1 und muss bei der Verarbeitung aufgelöst werden. Diese Darstellung ist äquivalent dazu, alle Token direkt einem übergeordneten „Markable“-Element zu unterstellen und muss daher nicht weiter berücksichtigt werden.
Rekursive Einheiten können in PAULA über die Metamodellelemente „Struct“ und „Rel“ gebildet werden. Das „Struct“-Element ist vergleichbar mit einem „Nonterminal“-Element aus dem Tiger Metamodell und das „Rel“-Element realisiert die Möglichkeit, eine Relation für rekursive Einheiten zu erzeugen. Das „Rel“-Element besitzt ein „type“-Attribut, mit dem die Art der Kante näher beschrieben werden kann. Die PAULA 1.0 Spezifikation lässt dafür die Werte „edge“ und „secedge“ zu, die ebenfalls eine parallele Semantik zum Tigermetamodell haben. Ein „Rel“-Objekt ist immer Teil eines „Struct“-Objektes, das gleichzeitig die Quelle der Relation symbolisiert. Das Ziel dieser Relation kann ein „Token“-Objekt, ein „Markable“-Objekt oder ein anderes „Struct“-Objekt sein. Zyklen dürfen bei dieser Relation nicht entstehen.
Neben dem „Rel“-Element gibt es eine weitere Art der Relation, das „PointingRelation“-Element. Diese Relation kann die Elemente „Struct“, „Token“ und „Markable“ zueinander in Beziehung setzen. Es wird typischerweise für nicht abdeckungsvererbende Beziehungen genutzt, die im Gegensatz zum „Rel“-Element keine „Teil von“-Beziehung ausdrückt. Beispielsweise können so *anaphorische Ketten* ausgedrückt werden.
Strukturelle Einheiten („Token“-, „Markable“-, „Struct“ und „Rel“-Elemente) können in PAULA allgemein in Form von Attribut-Wert-Paaren annotiert werden. Es gibt dabei keinerlei Bindung an bestimmte Semantiken. Eine part-of-speech-Annotation („POS= VVFIN“) wird genauso als Attribut-Wert-Paar behandelt wie eine syntaktische Annotation („cat=NP“). Lediglich die Relation „PointingRelation“ kann nicht annotiert werden.
PAULA unterstützt die Möglichkeit, Primärtexte, strukturelle Einheiten und Annotationen zu Korpora und Subkorpora bzw. Dokumenten zusammenzufassen. Der u.a. dafür eingesetzte Mechanismus ist allerdings etwas komplex. Abbildung 3.10 zeigt einen Ausschnitt des Metamodells der dafür zuständig ist.

Das PAULA-Format fasst Elemente wie „Token“, „Markable“, „Struct“, „Rel“ und „Feature“ durch das „MarkList“-[1], das „StructList“-[2] oder das „FeatList“-Element[3] zusammen. Das „*List“-Element[4] besitzt das Attribute „ns“ und „type“, die einen Namensraum bzw. einen Typ angeben. Im Falle der „FeatList“-Elemente gibt das „type“-Attribut den Attributnamen für Attribut-Wert-Paar-Annotationen durch das „Feat“-Element an.

[1] im Falle von „Token“- und „Markable“-Elementen
[2] im Falle von „Struct“- und „Rel“-Elementen
[3] im Falle des „Feature“-Elementes
[4] „MarkList“-, „StructList“- oder „FeatList“-Element

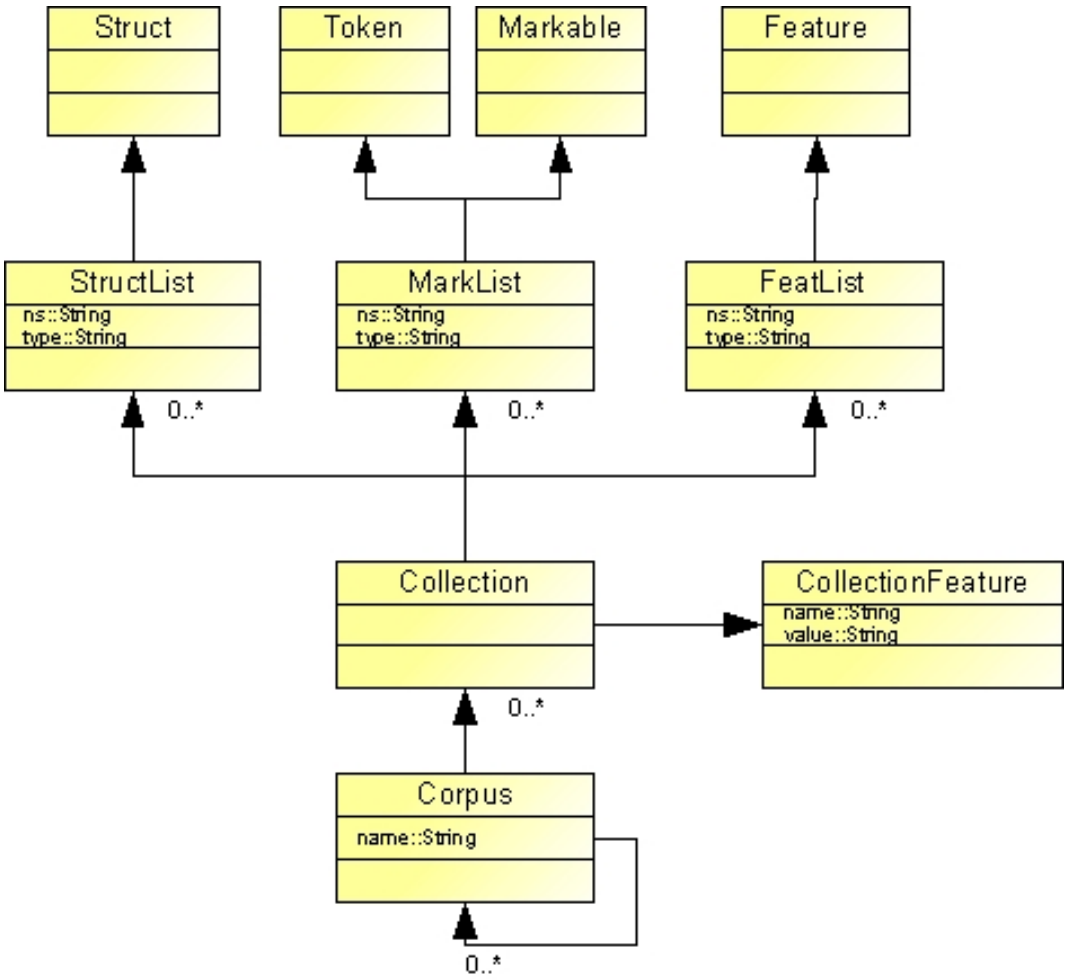


Abbildung 3.10.: Ausschnitt aus dem Metamodell des PAULA-Formats der Version 1.0. Hier wird die Strukturierung der Elemente aus Abbildung 3.9 dargestellt.

Dieser Zusammenhang mag etwas verwirrend erscheinen, bedeutet aber nur, dass mehrere „Feature“-Objekte gesammelt werden und einen gemeinsamen Annotationsnamen besitzen. Ebenso können auch mehrere „Struct“-Objekte zu einem Typ zugeordnet werden, der evtl. etwas über ihre linguistische Interpretation aussagen kann. Der Typ für „Token“-Objekte ist auf „type=tok“ festgelegt. Über die Angabe eines Namensraum können einem Typen Zusatzinformationen mitgegeben werden.
„*List“-Objekte können ebenfalls zusammengefasst werden. Dies geschieht durch das „Collection“-Element. Durch diese erneuten Zusammenfassungen können die dadurch entstehenden Gruppen mit Metaannotationen (CollectionFeature) versehen werden. Eine Zusammenführung unterschiedlicher „Collection“-Objekte kann durch das „Corpus“-Element geschehen. Dieses besitzt einen Namen und kann neben „Collection“-Objekten andere „Corpus“-Objekte enthalten. Dadurch werden Super- und Subkorpora dargestellt.
An dieser Stelle möchte ich erwähnen, dass die Elemente „Collection“, „CollectionFeature“ und „Corpus“ sowie „Token“, „PRable“ und „Relable“ nicht zum Sprachumfang des

PAULA-Formates gehören. Sie dienen nur der vereinfachten Darstellung. In PAULA werden diese Beziehungen etwas komplizierter durch *XML*-Techniken (wie bspw. die implizite Baumstruktur) o.Ä. ausgedrückt. Z.B. kann die Unterteilung in Sub- und Superkorpora durch die Verzeichnisstruktur, in der sich die einzelnen *XML*-Dokumente befinden ausgedrückt werden.
Ich habe das PAULA-Format in die Liste der zu untersuchenden Formate aufgenommen, da es zu den wenigen Formaten gehört, die nicht nur einen Aspekt linguistischer Datenverarbeitung, sondern linguistische Interpretationen allgemein darstellen wollen. PAULA ist zudem ein für die Datennachhaltigkeit relativ verbreitetes Format und relativ ausdrucksmächtig. Andere derartige Formate mit einem vergleichbarem Darstellungsumfang sind mir neben dem hier ebenfalls vorgestellten Format relANNIS, nicht bekannt.

3.1.5. relANNIS-Format

ANNIS2 ist ein Werkzeug zur Durchsuchung und Visualisierung linguistisch annotierter Korpora (siehe [Zeldes et al., 2009]). Das System wurde ebenfalls im Rahmen des SFB 632 von der Universität Potsdam und der Humboldt Universität zu Berlin entwickelt. ANNIS2 ist kein Annotationswerkzeug wie bspw. TreeTagger oder EXMARaLDA, es ermöglicht nur den Zugriff auf bereits annotierte Daten. Das Werkzeug basiert auf einer relationalen Datenbank und nutzt das relationale Datenformat relANNIS, das auf *CSV*-Dateien aufbaut. ANNIS2 wurde entwickelt, um Daten unterschiedlicher Herkunft anfrag- und visualisierbar zu machen. Es unterstützt Annotationen basierend auf Spannen, Hierarchien und verschiedene Arten von Relationen zwischen den strukturellen Einheiten.
Zur Validierung eines relANNIS Modells existiert eine relationale Schemabeschreibung in Form einer SQL Datei. Abbildung 3.11 zeigt eine Vereinfachung des Metamodells von relANNIS, das die grundlegenden Konzepte hinter diesem Format verdeutlicht.

Das relANNIS Metamodell basiert allgemein auf der Struktur eines *Baumes*. Strukturelle Einheiten werden auf Knoten abgebildet, Relationen auf Kanten. Das „Text“-Element dient der Speicherung des Primärtextes, dieser kann neben den Rohdaten („Text.text“) einen Namen („Text.name“) erhalten. Jedes „Node“-objekt stellt eine strukturelle Einheit (Token, Spanne oder rekursive Einheit) dar und hat einen eindeutigen Bezug zu genau einem Primärtext. Eine Beziehung der einzelnen „Node“-Objekte zueinander wird über Objekte des Typs „Rank“ erzeugt. Vereinfacht gesagt ist das „Rank“-Element gleichbedeutend mit einer Relation. Ein „Rank“-objekt besitzt über die Referenz „Rank.node“ einen Knoten, der als Ziel der Relation bezeichnet werden kann. Über die Referenz „Rank.parent“ kann dem „Rank“-Objekt ein Vorfahre zugeordnet werden. Damit kann einem „Rank“-objekt ein Quellknoten über die Referenz „Rank.parent.node“ zugeordnet werden. Mit diesen beiden Elementen werden Strukturen von linguistischen Interpretationen erzeugt. Ist ein Knoten bezogen auf die Baumstruktur ein Blattknoten, so ist er gleichbedeutend mit einem Token. Alle anderen Knoten sind rekursive Einheiten oder Spannen. Einem Knoten kann zusätzlich ein Name „Node.name“ und ein Namensraum „Node.namespace“ zugeordnet werden. Das Element „Rank“ verfügt ebenfalls über die Attribute „name“ „namespace“ und zusätzlich dem Attribut „type“. Über diese drei Attribute kann einer Kante eine Bedeutung gegeben werden, die wichtig für die Interpretation bei der Anfragebearbeitung durch ANNIS2 ist. Unterteilt wird hier in drei Relationsarten

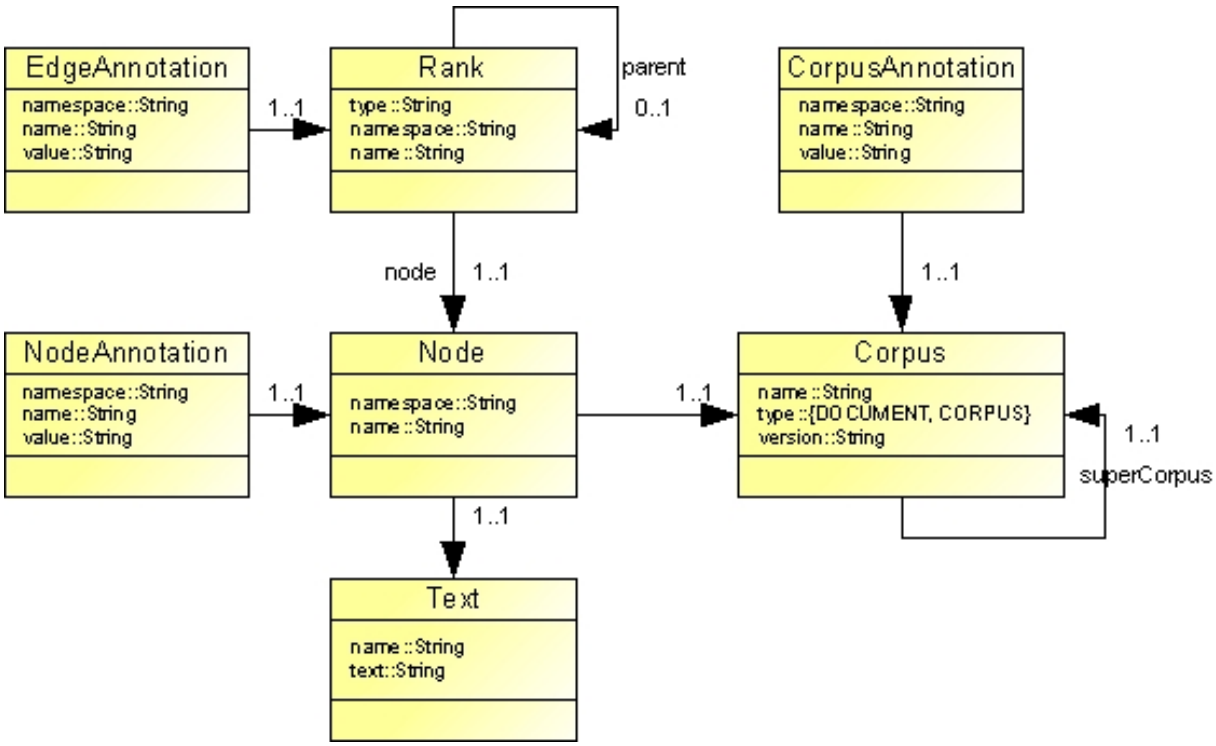


Abbildung 3.11.: vereinfachte Darstellung des Metamodells hinter dem Format relANNIS

1) „coverage relation“, 2) „dominance relation“ und 3) „pointing relation“. Mit 1) können Blattknoten und deren Vorfahren verbunden sein und haben dann die Bedeutung einer Spanne inne. 2) dient der Bildung rekursiver Einheiten (die Quelle einer solchen Relation muss immer ein innerer Knoten sein) und mit 3) können nicht abdeckungsvererbende Beziehungen zwischen beliebigen Knoten definiert werden. Für diese drei Relationen gibt es in AQL (siehe [SFB 632, 2009]), der Anfragesprache von ANNIS2, verschiedene Operatoren.
Linguistische Annotationen können in relANNIS für „Rank“- und „Node“-Objekte vergeben werden. Hierzu dienen die Elemente „EdgeAnnotation“ bzw. „NodeAnnotation“. Beide Elemente verfügen über die Attribute „namespace“, „name“ und „value“, die eine Attribut-Wert-Paar-Zuordnung mit einem erweiterten Attribut-Identifizierer („namespace“) darstellen.
Strukturelle Einheiten, Primärtexte, Relationen und Annotationen können in relANNIS zu Korpora, Subkorpora und Dokumente zusammengefasst werden. Hierzu existiert das Element „Corpus“. Jedes „Node“-Objekt besitzt genau eine Referenz zu einem „Corpus“-Objekt. Einem Objekt des Elements „Corpus“ kann über die Referenz „Corpus.superCorpus“ ein Superkorpus zugeordnet werden. Ein „Corpus“-Objekt auf unterster Ebene ist ein Dokument und enthält im Attribut „Corpus.type“ den Wert „Document“. Ein „Corpus“-Objekt kann über das Element „CorpusAnnotation“ mit zusätzlichen Informationen sog. Metaannotationen versehen werden. Dies könnten bspw. Informationen zum Textautor oder zum Annotator sein.

Ich habe geschrieben, dass die Struktur in relANNIS der eines *Baumes* gleichen muss. Diese Aussage ist allerdings nicht ganz zutreffend. An dieser Stelle vermischt das relANNIS-Format die Speicherung der Daten und die Erzeugung eines Index für den schnelleren Datenzugriff durch ANNIS2. Hierfür wird die Pre- und Post- Order (siehe [Grust et al., 2004]) verwendet und dafür muss die Struktur einem *Baum* entsprechen. Vitt hat in [Vitt, 2005] [5] jedoch gezeigt, wie ein *DAG* ebenfalls auf die Struktur eines *Baumes* abgebildet werden kann. Damit können in relANNIS *DAGs* gespeichert werden.
Ich habe das relANNIS-Format in die Liste der zu untersuchenden Formate aufgenommen, da es zu den wenigen Formaten gehört, die auf einem relationalen Datenbankschema aufbauen. relANNIS hat zudem eine ähnliche Mächtigkeit wie PAULA.

3.2. Konzepte

An mehreren Stellen habe ich bereits erwähnt, dass die gerade vorgestellten Formate zu sehr unterschiedlichen Zwecken entwickelt wurden und daher sehr unterschiedliche linguistische Analysen und Interpretationen ermöglichen. Ich habe die Darstellungskraft eines Formates bisher nur isoliert betrachtet vorgestellt. Das gemeinsame Metamodell, das ich im Rahmen dieser Arbeit entwickelt habe, soll jedoch in der Lage sein, möglichst alle Aspekte der hier untersuchten Formate zu unterstützen. Für die Schaffung dieser Struktur kann jedoch nicht jeder Aspekt einzeln betrachtet werden. Schließlich soll das gemeinsame Metamodell eine Abstraktion der Formate sein und gleiche Aspekte auch gleich darstellen. Bei der Analyse und dem Vergleich der Aspekte müssen diese abstrahiert von ihrer Darstellung in einem bestimmten Format betrachtet werden. Durch diese Abstraktion ist es möglich, Gemeinsamkeiten in der Darstellungskraft der Formate zu identifizieren. Diese Abstraktion habe ich in Form von Konzepten vorgenommen.

Begriff 14 (Konzept) *Ein Konzept beschreibt formatunabhänig eine bestimmte linguistische Struktur oder eine linguistische Deutung. Ein Konzept ist i.d.R. in mehreren Formaten durch eine konkrete Modellierung repräsentiert.*

Im Folgenden werde ich eine Reihe von Konzepten vorstellen, die ich in Bezug auf die untersuchten Formate identifiziert habe. Anschließend werde ich zeigen, in welchen Formaten diese Konzepte wie repräsentiert werden. Die Konzepte dienen im nächsten Abschnitt (siehe Abschnitt 3.3) als Grundlage für einen Vergleich der Formate und in Kapitel 4 als Grundlage zur Entwicklung des gemeinsamen Metamodells.

Konzept 1 (Primärtextkonzept) *Das Primärtextkonzept bezeichnet die Möglichkeit in einem Metamodell, den Primärtext darstellen zu können. Damit ist nicht gemeint, die Aneinanderreihung von Primärfragmenten extrahieren zu können, sondern die Möglichkeit den (digitalisieren,) originalen Text (mitsamt evtl. Leerzeichen, Zeilenumbrüchen etc.) speichern zu können.*

[5] diese Arbeit stammt aus dem DDD-Projekt, dem Vorgänger von ANNIS2

Konzept 2 (Multitextkonzept) *Das Multitextkonzept schließt an das Primärtextkonzept an und bezeichnet die Möglichkeit, mehrere Primärtexte als Grundlage einer gemeinsamen Annotationsstruktur speichern zu können. Dieses Konzept ist bspw. dann wichtig, wenn Parallelkorpora oder Diskurse betrachtet werden. Ein Parallelkorpus enthält zwei oder mehrere Fassungen eines Textes. Diese Fassungen liegen meist in unterschiedlichen Sprachen vor und werden durch zwei Primärtexte repräseniert. Anschließend werden strukturelle Einheiten beider Texte miteinander in Beziehung gesetzt (aligniert). Meist stellen solche Beziehungen semantische Identitäten also Übersetzungen dar. In Diskursen liegt ein Primärtext pro Diskursteilnehmer vor.*

Konzept 3 (Tokenkonzept) *Durch das Tokenkonzept kann ein Primärtext in Primärfragmente unterteilt werden. Das Tokenkonzept beinhaltet nicht die Deutung eines Tokens als bspw. eine Silbe, ein Wort, einen Satz etc.. Es beinhaltet nur die Schaffung einer kleinsten Einheit, die Teil weiterer Strukturen oder Annotationen ist.*

Konzept 4 (Zeitkonzept) *Als Zeitkonzept bezeichne ich hier die Möglichkeit, eine zeitliche Ebene in die Daten zu integrieren. Darunter ist zu verstehen, dass strukturelle Einheiten einem linearen zeitlichen Ablauf zuzuordnen sind. Dieses Konzept ist vor allem dann wichtig, wenn das Multitextkonzept unterstützt werden soll. Mit dem Zeitkonzept können strukturelle Einheiten in eine Ordnung außerhalb der durch den Primärtext Vorgegebenen gebracht werden.*

Konzept 5 (Spannenkonzept) *Das Spannenkonzept bezeichnet die Möglichkeit, Token zu Spannen zusammenzufassen. Zwischen einer Spanne und einem Token besteht eine abdeckungsvererbende Beziehung. Eine Spanne kann kontinuierlich oder diskontinuierlich sein. Kontinuierlich bedeutet, dass die Token, die zu einer Spanne gehören, aufeinanderfolgend[6] sind. Diskontinuierlich bedeutet, dass die Token nicht aufeinanderfolgend sein müssen.*

Konzept 6 (Hierarchiekonzept) *Unter dem Hierarchiekonzept verstehe ich die Möglichkeit, rekursive Einheiten bilden zu können. Es beinhaltet die Möglichkeit, dafür notwendige Elemente zur Verfügung zu stellen, die mit sich selber, aber auch Spannen und Token in Beziehung gesetzt werden können. Eine Beziehung zwischen diesen Einheiten ist eine abdeckungsvererbende Relation. Die Art der entstehenden Struktur, ob Baum, DAG oder allgemeiner Graph wird hier nicht betrachtet.*

Konzept 7 (Nicht-AVR-Konzept) *Das Nicht-AVR-Konzept (Nicht-Abdeckungsvererbende-Relationen-Konzept) bezeichnet die Möglichkeit nicht-abdeckungsvererbende Relationen repräsentieren zu können. Nicht-abdeckungsvererbend bedeutet: wenn zwischen den Elementen a und b eine solche Beziehung besteht, dann wird das von a abgedeckte Primärfragment nicht automatisch von b abgedeckt und andersherum. Eine nicht-abdeckungsvererbende Relation ist gerichtet.*

[6] im Bezug auf eine Datenquelle, meist der Primärtext.

Konzept 8 (Schichtenkonzept) *Das Schichtenkonzept bietet die Möglichkeit, strukturelle Einheiten, Relationen und Annotationen zu einer Schicht zusammenzufassen. Einer Schicht können dann bspw. alle strukturellen Einheiten, Relationen und Annotationen zugeordnet werden, die sich auf eine linguistische Syntaxanalyse beziehen.*

Konzept 9 (Dokument- und Korpuskonzept) *Das Dokument- und Korpuskonzept bezeichnet die Möglichkeit, in einem linguistischen Modell Primärdaten, Strukturdaten und Annotationen zu einem Dokument zusammenzufassen und anschließend in Korpora zu organisieren.*

Konzept 10 (Attribut-Wert-Paar-Konzept) *Dieses Konzept bezeichnet die Möglichkeit, strukturelle Einheiten und Relationen sowie Dokumente und Korpora mit linguistischen Annotationen in Form von Attribut-Wert-Paaren*[7] *zu annotieren. Da nicht alle untersuchten Formate eine Annotation mit Attribut-Wert-Paaren auf allen Einheiten zulassen, wird es in die folgenden Subkonzepte unterteilt:*

- *Token - ein Token kann mit einem Attribut-Wert-Paar annotiert werden.*
- *Spanne - eine Spanne kann mit einem Attribut-Wert-Paar annotiert werden.*
- *rekursive Einheit - eine rekursive Einheit kann mit einem Attribut-Wert-Paar annotiert werden.*
- *Relation - eine Relation kann mit einem Attribut-Wert-Paar annotiert werden.*

Konzept 11 (Metaannotationskonzept) *Das Metaannotationskonzept bezeichnet wie auch das Attribut-Wert-Paar-Konzept die Möglichkeit, Einheiten mit Attribut-Wert-Paaren zu belegen. Im Gegensatz zu den linguistischen Annotationen sind Meta-Annotationen nicht unbedingt linguistischer Natur. Mit Meta-Annotationen werden Zusatzinformationen wie bspw. der Name des Autors eines Textes, der Name des Annotators, die Korpusversion oder das Erstellungsdatum gespeichert. Wie auch bei dem Attribut-Wert-Paar-Konzept unterteile ich das Metaannotationskonzept in die folgenden Subkonzepte:*

- *Token - ein Token kann mit einer Meta-Annotation belegt werden.*
- *Spanne - eine Spanne kann mit einer Meta-Annotation belegt werden.*
- *rekursive Einheit - eine rekursive Einheit kann mit einer Meta-Annotation belegt werden.*
- *Relation - eine Relation kann mit einer Meta-Annotation belegt werden.*
- *Dokument - ein Dokument kann mit einer Meta-Annotation belegt werden.*
- *Korpus - ein Korpus kann mit einer Meta-Annotation belegt werden.*

[7] Also die nicht eingeschränkte Vergabe beliebiger Attributnamen und beliebiger Attributwerte.

Konzept 12 (Annotations-Semantik-Konzept) *Das Annotations-Semantik-Konzept bezeichnet die Möglichkeit, einer strukturellen Einheit, einer Relation oder einer Annotation eine bestimmte Deutung zuzuweisen. Wird bspw. in einem Format eine linguistische Struktur als Satz gedeutet, so kann diese Deutung allgemein nicht durch die strukturelle Einheit dargestellt werden. Dieses Konzept ist eine Abstraktion und wird durch die Subkonzepte Wort-Semantik-Konzept , Satz-Semantik-Konzept , POS-Semantik-Konzept , Lemma-Semantik-Konzept und Cat-Semantik-Konzept verfeinert. Die ersten beiden Konzepte beziehen sich auf strukturelle Einheiten und die letzten drei Konzepte auf Annotationen.*

In Abschnitt 3.3 werde ich aufgrund der hier vorgestellten Konzepte einen Vergleich der Formate vornehmen. Dabei werde ich zeigen, welche Formate in der Lage sind, welche Konzepte zu unterstützen. Außerdem werde ich auf Verluste eingehen, die durch die fehlende Unterstützung mancher Konzepte in einigen Formaten entstehen.

3.3. Vergleich der linguistischen Annotationsformate

Die Abbildungen 3.12 und 3.13 zeigen, welche Konzepte von welchen Formaten unterstützt werden. Hierbei unterscheide ich drei Arten der Unterstützung. Erstens die native Unterstützung (gekennzeichnet durch 'x'). Sie bedeutet, dass ein Format ein Konzept mit Sprachmitteln realisiseren kann, die nach der entsprechenden Formatbeschreibung dafür vorgesehen sind. Zweitens die bedingte Unterstützung (gekennzeichnet durch '-'). Bedingt bedeutet, dass ein Format ein Konzept nur teilweise unterstützt. Und drittens die simulierte Unterstützung (gekennzeichnet durch '/'). Dabei kann ein Konzept zwar durch die Sprachmittel des Formates erzeugt werden, aber diese Unterstützung ist entweder nicht vorgesehen oder in der Formatbeschreibung nicht enthalten.

3.3.1. Primärtextkonzept

Die Formate PAULA und relANNIS bilden das Primärtextkonzept nativ ab. Dies geschieht durch das Metamodellelement „Text“ des jeweiligen Metamodells. Die Formate Treetagger und Tiger besitzen keine Möglichkeit den Primärtext zu speichern. Für das Format EXMARaLDA ist eine simulierte Unterstützung möglich. Dafür können die Elemente „Tier“ und „Event“ verwendet werden. Bspw. kann das Attribut „Tier.category“ eines „Tier“-Objektes mit dem Wert „text“ belegt werden. Diesem „Tier“-Objekt wird dann genau ein „Event“-objekt zugeordnet, dass die gesamte Zeitlinie umfasst. So kann das Attribut „Event.value“ den Primärtext repräsentieren.

3.3.2. Multitextkonzept

Das PAULA-Format und das relANNIS-Format sind in der Lage für ein Dokument mehrere Primärtexte zu speichern. Die Formate Treetagger und Tiger müssen hierfür mehrere Dokumente erzeugen. In dem Format EXMARaLDA können, wie für das Primärtextkonzept gezeigt, weitere Primärtextebenen simuliert werden. Dafür können weitere „Tier“-

strukturelle Konzept\Format	Treetagger	EXMARaLDA	TIGER	PAULA	relANNIS
Primärtextkonzept		/		x	x
Multitextkonzept		/		x	x
Tokenkonzept	-	/	-	x	x
Zeitkonzept		x			
Spannenkonzept					
- kontinuierlich	x	x	/	x	x
- diskontinuierlich			/	x	x
Hierarchiekonzept			x	x	x
Nicht-AVR-Konzept			x	x	x
Schichtkonzept		x		x	x
Dokument- und Subkorpuskonzept	/	/	x	x	x
Attribut-Wert-Paar-konzept					
-Token	-	x	x	x	x
-Spanne	x	x	x	x	x
-rekursive Struktur			x	x	x
-Relation			-	x	x
Metaannotationskonzept					
-Token		x		x	
-Spanne		x		x	
-rekursive Struktur		x		x	
-Relationen				x	
-Dokument		x	-	x	x
-Korpus			-	x	x

Abbildung 3.12.: Unterstützung struktureller Konzepte

Objekte erzeugt werden, die jeweils ein „Event“-objekt zugewiesen bekommen. Das „Event“-Objekt ist dann nicht notwendigerweise der ganzen Zeitlinie zuzuordnen. Es wäre nur zu beachten, dass all derartige „Event“-Objekte zusammen die Zeitlinie überdecken.

3.3.3. Tokenkonzept

Ein Token kann im PAULA-Format durch das Element „Token“, im relANNIS-Format durch das Element „Node“ dargestellt werden. Beim EXMARaLDA-Format kann ebenfalls von einer Tokenisiserung, bezogen auf die Zeitlinie gesprochen werden. Ein Tokenisiserung bezogen auf einen Primärtext, kann allerdings nur simuliert unterstützt werden, da zwar eine Zerteilung eines Textes stattfinden kann, ein Primärtext aber nur simuliert unterstützt wird. Außerdem kann nicht eindeutig klar gemacht werden, welche der „Tier“-

semantische Konzept\Format	Treetagger	EXMARaLDA	TIGER	PAULA	relANNIS
Annotations-Semantik-Konzept					
-Wort-Semantik-Konzept	x	/	x	/	/
-Satz-Semantik-Konzept	x	/	x	/	/
-POS-Semantik-Konzept	x	/	x	/	/
-Lemma-Semantik-Konzept	x	/	x	/	/
-Cat-Semantik-Konzept	/	/	x	/	/

Abbildung 3.13.: Unterstützung semantischer Konzepte

Objekte für eine Tokenisierung stehen. Das Format Treetagger geht i.d.R. von einer Tokenisierung auf Wortbasis aus, das Format Tiger geht ausschließlich von einer Tokenisierung auf Wortbasis aus. Damit wird das Tokenkonzept allgemein nicht unterstützt.

3.3.4. Zeitkonzept

Das einzige Format, das das Konzept einer Zeitlinie nativ unterstützt, ist das EXMARaLDA-Format. Hier wird die Zeit durch die Elemente „CommonTimeline“ und „TLI“ umgesetzt. Die Formate PAULA, relANNIS, Tiger und TreeTagger können dieses Konzept über eine Attribut-Wert-Paar-Annotation simulieren. Dies kann auf verschiedenen Ebenen geschehen (siehe Abschnitt 3.3.10).

3.3.5. Spannenkonzept

Eine Repräsentation von kontinuierlichen Spannen wird in den Formaten Treetagger, EXMARaLDA, PAULA und relANNIS nativ unterstützt. In Treetagger geschieht dies über das Metaelement „Span“, bzw. über die SGML-Tags. EXMARaLDA repräsentiert eine Spanne durch das „Event“-Element, PAULA durch das „Markable“-Element und in relANNIS kann dafür ein „Node“-Objekt verbunden mit einem „Rank“-Objekt mit der Attributbelegung „Rank.type=c“ verwendet werden. Für das Tiger-Format ist es möglich, eine Spanne über das „Nonterminal“-Element zu simulieren. Jede Spanne kann durch ein solches Objekt repräsentiert werden. Über Relationen (einem „Edge“- oder „Secedge“-Objekt) kann die Spanne eine Menge von Worten bzw. „Terminal“-Objekten referenzieren.
Diskontinuirliche Spannen können durch die Formate PAULA, relANNIS und bezogen auf die simulierte Unterstützung durch Tiger dargestellt werden. In den Formaten EXMARaLDA und TreeTagger können nur kontinuierliche Spannen dargestellt werden.

3.3.6. Hierarchiekonzept

Die Formate Tiger, PAULA und relANNIS können rekursive Einheiten darstellen. Dafür sind die Elemente „Nonterminal“, „Struct“ bzw. „Node“ vorgesehen. Diese Formate verfügen außerdem über Möglichkeiten Relationen zu erzeugen, die rekursive Einheiten miteinander verbinden. Die Realisierung erfolgt über die Elemente „Edge“ bzw. „Secedge“ in Tiger, „Rel“ in PAULA und „Rank“ mit „Rank.type=d“ in relANNIS. In Treetagger und EXMARaLDA können rekursive Einheiten nicht erzeugt werden. Zwar könnten Spannen dafür eingesetzt werden, beide Formate verfügen aber über keine Möglichkeit diese über Relationen miteinander in Beziehung zu setzen.

3.3.7. Nicht-AVR-Konzept

Die Formate PAULA und relANNIS bieten explizit eine Möglichkeit an, ein solches Konzept zu unterstützen. In PAULA wird dies über das Metamodellelement „PointingRelation“ realisiert. relANNIS bietet ebenfalls die Möglichkeit eine Relation bzw. ein „Rank“-Objekt als „pointing relation“ zu markieren. In Tiger können hierfür Objekte des Typs „SecEdge“ genutzt werden. Für die Darstellung des Nicht-AVR-Konzepts ist es erforderlich, dass ein Format explizit Relationen zwischen strukturellen Einheiten darstellen kann. Damit gehören die Formate EXMARaLDA und Treetagger nicht zu der Menge der, dieses Konzept unterstützenden, Formate.

3.3.8. Schichtenkonzept

Schichten werden durch die Formate relANNIS und PAULA unterstützt. In beiden können strukturelle Einheiten über die Nutzung eines Namensraumes einer Schicht zugeorndet werden. In dem Format EXMARaLDA gibt es ebenfalls die Möglichkeit „Event“-Objekte (Spannen und Token) zu einem „Tier“-Objekt zu zuordnen. Ein „Tier“-Objekt steht dann für eine Schicht. Die Unterstützung durch EXMARaLDA ist allerdings dadurch beschränkt, dass einer Schicht pro Zeitabschnitt nur genau ein „Event“-Objekt zugeordnet werden kann. In den Formaten TreeTagger und Tiger fehlt eine Unterstützung dieses Konzeptes.

3.3.9. Dokument- und Korpuskonzept

Dieses Konzept kann prinzipiell als von allen untersuchten Formaten unterstützt betrachtet werden. Einige Formate wie Tiger, PAULA und relANNIS modellieren diese Organistaionsstruktur explizit durch die Elemente „Corpus“ bzw. „Subcorpus“ und „Graph“ in Tiger sowie „Corpus“ in PAULA und relANNIS. Für andere linguistische Modelle wie Treetagger und EXMARaLDA kann diese Struktur implizit über die Verzeichnisstruktur erzeugt bzw. als gegeben betrachtet werden. Dabei gibt ein Verzeichnis immer ein Korpus an. Verzeichnishierarchien werden direkt auf Korpushierarchien abgebildet und die Datei im entsprechenden Format wird auf ein Dokument abgebildet.

3.3.10. Attribut-Wert-Paar-Konzept

PAULA und relANNIS unterstützen nativ die Annotation mit Attribibut-Wert aller strukturellen Einheiten und Relationen. In Tiger kann zwar innerhalb eines bestimmten Korpus eine Einschränkung des Definitions- und Wertebereichs vorgenommen werden, diese ist aber frei bestimmbar und damit ebenfalls offen. Nur Relationen können nicht beliebig annotiert werden. Hierfür gibt es in Tiger lediglich die Möglichkeit, einer Relation genau einen beliebigen Wert in Form des „label"-Attributes zuzuweisen. Simulierte Spannen können beliebig annotiert werden.
In der Dokumentation zu Treetagger steht zwar: „TreeTagger is a tool for annotating text with part-of-speech and lemma information" ([Schmid, 1994], zuletzt besucht am 11.11.2009), dies gilt aber erstens für das Werkzeug Treetagger und zweitens können auch andere Analysemodule in Treetagger integriert werden. Das Format trifft wenig Aussage darüber, welcher Art die Annotationen sein dürfen. Allerdings bietet Treetagger keine Möglichkeit an, für Annotationen von Token den Definitionsbereich der Annotationen zu benennen. Da sowohl TreeTagger als auch EXMARaLDA keine rekursiven Einheiten darstellen können, können diese auch nicht annotiert werden. In EXMARaLDA unterstützt das Attribut-Wert-Paar-Konzept bezogen auf Spannen nativ. Dies gilt ebenfalls für simulierte Token.

3.3.11. Metaannotationskonzept

PAULA ist das einzige Format der hier untersuchten, dass alle Subkonzepte des Metaannotationskonzepts nativ unterstützt. In relANNIS können Meta-Annotationen für Korpora und Dokumente erzeugt werden, nicht aber für Token, Spannen, rekursive Einheiten oder Relationen. Tiger erlaubt eine bedingte Unterstützung des Konzepts bezogen auf Dokumente und Korpora. Bedingt deshalb, weil die Attributnamen nicht frei wählbar sind. In EXMARaLDA sind Meta-Annotationen über den Einheiten Token, Spanne und rekursive Einheit, so diese denn simuliert wird, nativ möglich. Ein Dokument kann sowohl mit vorgegebenen Attributnamen als auch frei annotiert werden. In Treetagger können Meta-Annotationen gar nicht explizit vergeben werden. Die Subkonzepte des Metaannotationskonzepts können zwar von jedem Format simuliert werden, das das Attribut-Wert-Paar-Konzept unterstützt, allerdings würde die Trennung von linguistischer Annotation und Meta-Annotation verloren gehen.

3.3.12. Annotations-Semantik-Konzept

Allgemein kann das Annotations-Semantik-Konzept von keinem der untersuchten Formate dargestellt werden. Lediglich manche Verfeinerungen werden durch unterschiedliche Formate dargestellt. Das Wort-Semantik-Konzept - und Satz-Semantik-Konzept wird von den Formaten TreeTagger und Tiger unterstützt. Das POS-Semantik-Konzept und das Lemma-Semantik-Konzept wird ebenfalls durch das Format Treetagger und durch das Modell der Tiger *API* unterstützt. Das Cat-Semantik-Konzept wird nur durch die Tiger *API* unterstützt. Die Formate relANNIS, PAULA und EXMARaLDA besitzen nur eine simulierte Unterstützung.

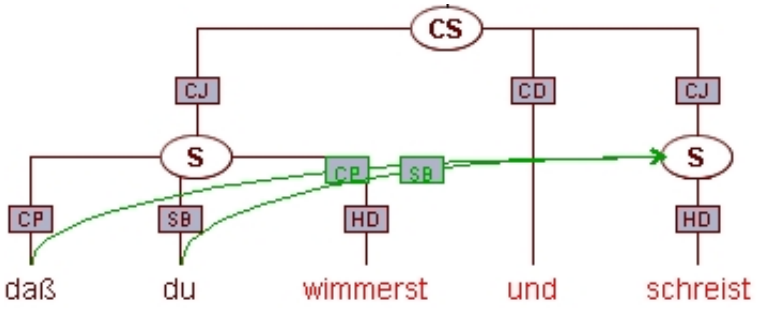
3.3.13. Informationsverluste einer Konvertierung

Aufgrund der gerade vorgestellten Konzepte, kann ein Eindruck über die Mächtigkeit eines Formates gewonnen werden. Diese ist maßgebend bei der Konvertierung von einem Format in ein anderes, da sich hieran die vorbestimmten Informationsverluste der Konvertierung bestimmen lassen.

Begriff 15 *Unter vorbestimmten Informationsverlusten verstehe ich Verluste, die bei der Konvertierung zweier Formate A und B entstehen, wenn diese nicht die gleiche Mächtigkeit besitzen. Angenommen A soll nach B konvertiert werden, B kann aber bestimmte Konzepte von A nicht darstellen, so gehen zwangsläufig alle Daten dieser Konzepte bei der Konvertierung verloren.*

Bei der Betrachtung der Mächtigkeit eines Formates muss anhand von nativen und simulierten Konzepten unterschieden werden. Da nativ unterstützte Konzepte sich dadurch ausdrücken, dass es in einem Format Sprachkonstrukte gibt, die diese repräsentieren, sind sie eineindeutig identifizierbar. Wird das Konzept K von den Formaten A und B nativ unterstützt, so kann davon ausgegangen werden, dass eine Transformation von A nach B, bezogen auf Konzept K, eine isomorphe Abbildung darstellt. Isomorph bezogen auf ein Konzept bedeutet hier, dass Daten eines Konzept in A und B eindeutig identifizierbar sind. Somit können sie, wenn sie von A auf B abgebildet wurden, auch wieder von B nach A abgebildet werden.
Für die simulierte Unterstützung von Konzepten gilt dies im Allgemeinen nicht. Da diese Konzepte nicht im Sprachumfang des Formates enthalten sind, kann auch nicht davon ausgegangen werden, dass alle das Format interpretierenden Werkzeuge von dieser Simulation Kenntnis haben. Da sie nicht eindeutig identifizierbar sind, kann ein Informationsverlust bei einer Rücktransformation generell nicht ausgeschlossen werden. Es liegt also nur ein Homomorphismus vor.
Vorbestimmte Informationsverluste können aufgrund fehlender oder simulierter Unterstützung eines Konzeptes durch das Zielformat entstehen. Generell gilt, wenn Daten eines Formates in ein anderes Format transformiert werden sollen und das Zielformat ein Konzept nicht unterstützt, kommt es zu einem Verlust. An manchen Stellen ist diese Betrachtung allerdings etwas zu simpel, da wie bei den simulierten Konzepten Informationen u.U. transformiert werden, im Zielformat aber nicht mehr gedeutet werden können.
Informationsverluste können struktureller oder semantischer Natur sein. Struktureller Natur sind Verluste dann, wenn das Zielformat nicht die strukturellen Mittel besitzt um ein Konzept darzustellen. In diesem Fall taucht das Datum im Zielformat nicht mehr auf. Diese Verluste sind relativ gut planbar. Schwieriger zu untersuchen sind semantische Verluste. Diese entstehen wenn bspw. ein Format ein Datum strukturell darstellt und diesem gleichzeitig noch eine Bedeutung, die über diese Struktur hinausgeht, gibt. Kann das Zielformat das Datum nur strukturell darstellen und besitzt jedoch nicht die Möglichkeit die gleiche Bedeutung auszudrücken, entsteht ein semantischer Verlust.
Ein struktureller Informationsverlust entsteht wenn z.B. Daten des Hierarchiekonzepts aus bspw. dem Tiger-Format auf das EXMARaLDA-Format abgebildet werden. Die rekursiven Einheiten können zwar in Spannen transformiert werden, in EXMARaLDA fehlt jedoch die Möglichkeit die Relationen der Hierarchie darzustellen. Diese Relationen sind

ein struktureller Verlust. Abbildung 3.14 zeigt ein Beispiel, das aus dem Tigerformat in das EXMARaLDA-Format transformiert wurde. Hier ist zu erkennen, dass die Relationen zwischen den rekursiven Einheiten bei der Überführung in Spannen verlorengingen.



(a) Beispiel in Tiger

	0	1	2	3	4
X [v]	CS				
X [v]	S				S
word	daß	du	wimmerst	und	schreist

(b) Beispiel in EXMARaLDA

Abbildung 3.14.: Beispiel aus dem Tiger-Format (oben, visualisiert mit TigerSearch) transformaiert in das EXMARaLDA-Format (unten, visualisiert durch den Partitur Editor)

Ein semantischer Verlust entsteht bspw. dadurch, dass ein Token im Tiger-Format immer einem Wort entspricht. Wird dieses Token in das EXMARaLDA-Format transformiert geht diese Information nicht verloren, da bspw. ein entsprechendes „Tier“-Objekt in EXMARaLDA mit dem „Tier.category“-Wert „word“ belegt werden könnte. Abbildung 3.14(b) zeigt dieses vorgehen. Die Deutung der Zeichenkette „word“ ist aber durch das Format nicht definiert. Einem das EXMARaLDA-Format interpretierenden Werkzeug steht es frei diese Information zu verarbeiten oder nicht. Wenn also die Daten an sich erhalten bleiben, ihre Deutung aber verloren geht (oder nicht spezifiziert ist) entsteht ein semantischer Verlust. Diese entstehen i.d.R. bei der nicht-Unterstüzung oder einer simulierten Unterstützung des Annotations-Semantik-Konzepts .

4. Entwicklung des gemeinsamen (Meta-)Modells

In diesem Kapitel werde ich das gemeinsame (Meta-)Modell `Salt` vorstellen. Das gemeinsame (Meta-)Modell ist der zentrale Kern zur Reduzierung der Anzahl benötigter Mappings bei der Konvertierung (siehe Abbildung 1.1 aus Abschnitt 1). Mappings können nur dann verlustfrei sein, wenn das Zielmodell mindestens die Mächtigkeit des Quellmodells besitzt. Um auszuschließen, dass es zu Verlusten bei der Abbildung der Daten aus den Formaten auf `Salt` kommt, habe ich die in Abschnitt 3.2 beschriebenen Konzepte in `Salt` umgesetzt. Ich werde in diesem Kapitel, während ich die einzelnen Komponenten von `Salt` beschreibe, deutlich machen wie und in welcher Komponente das jeweilige Konzept umgesetzt wurde. Am Ende dieses Kapitels werde ich zeigen, dass jedes Konzept in `Salt` umgesetzt wurde.
Linguistische Interpretationen bestehen im Wesentlichen aus Objekten und Relationen zwischen diesen. Hinzukommt, dass Objekte und Relationen eine bestimmte linguistische Semantik ausdrücken können. Diese Semantik wird in Form von Annotationen auf Objekten oder Relationen beschrieben. Ein Primärtext mit einer linguistischen Struktur und dazugehörigen Annotationen kann daher als eine graphartige Struktur beschrieben werden. An manchen Stellen wird auch von Annotationsgraphen gesprochen (siehe [Schmidt, 2005] S. 1). Die Graphentheorie ist ein gut erforschtes Gebiet mit einer breiten Palette existierender Algorithmen und gut untersuchten Traversierungsmöglichkeiten für Graphen. Ein Graph kann daher auch für linguistisch annotierte Daten verwendet werden. Dieser Gedanke findet sich auch in dem durch die ISO entwickelten linguistischen Metamodell LAF (siehe [Ide and Romary, 2003]) wieder. Dieses basiert auf einem Graphenmodell namens GrAF (siehe [Ide et al., 2007]). Auch ich betrachte linguistische Strukturen als einen Graphen. Daher bin ich zunächst von einem allgemeinen Graphmodell ausgegangen, und habe das gemeinsame (Meta-)Modell namens `Salt` anschließend auf dessen Basis entwickelt.
`Salt` besteht aus drei voneinander abgeleiteten Metamodellen. Abbildung 4.1 soll diesen Zusammenhang und den Bezug von `Salt` zu dem allgemeinen Graphen deutlich machen.

Das Metamodell von `SaltCoreMM` (`SaltCoreMM`) ist eine direkte Ableitung aus dem Metamodell eines allgemeinen Graphen, hier als `GraphMM` bezeichnet. Es stellt lediglich eine Anpassung des allgemeinen Graphmodells auf linguistische Anforderungen im Bezug auf unterschiedliche Annotationsmöglichkeiten dar. `SaltCoreMM` ist die Grundlage des eigentlichen gemeinsamen (Meta-)Modells `SaltCommonMM` , kann aber auch als Metamodell für andere graphartige linguistische Datenmodelle dienen. Das Metamodell `SaltCommonMM` beinhaltet die in Abschnitt 3.2 herausgearbeiteten strukturellen Konzepte. Da das (Meta-)Modell möglichst viele unterschiedliche Daten aus verschiedenen linguistischen

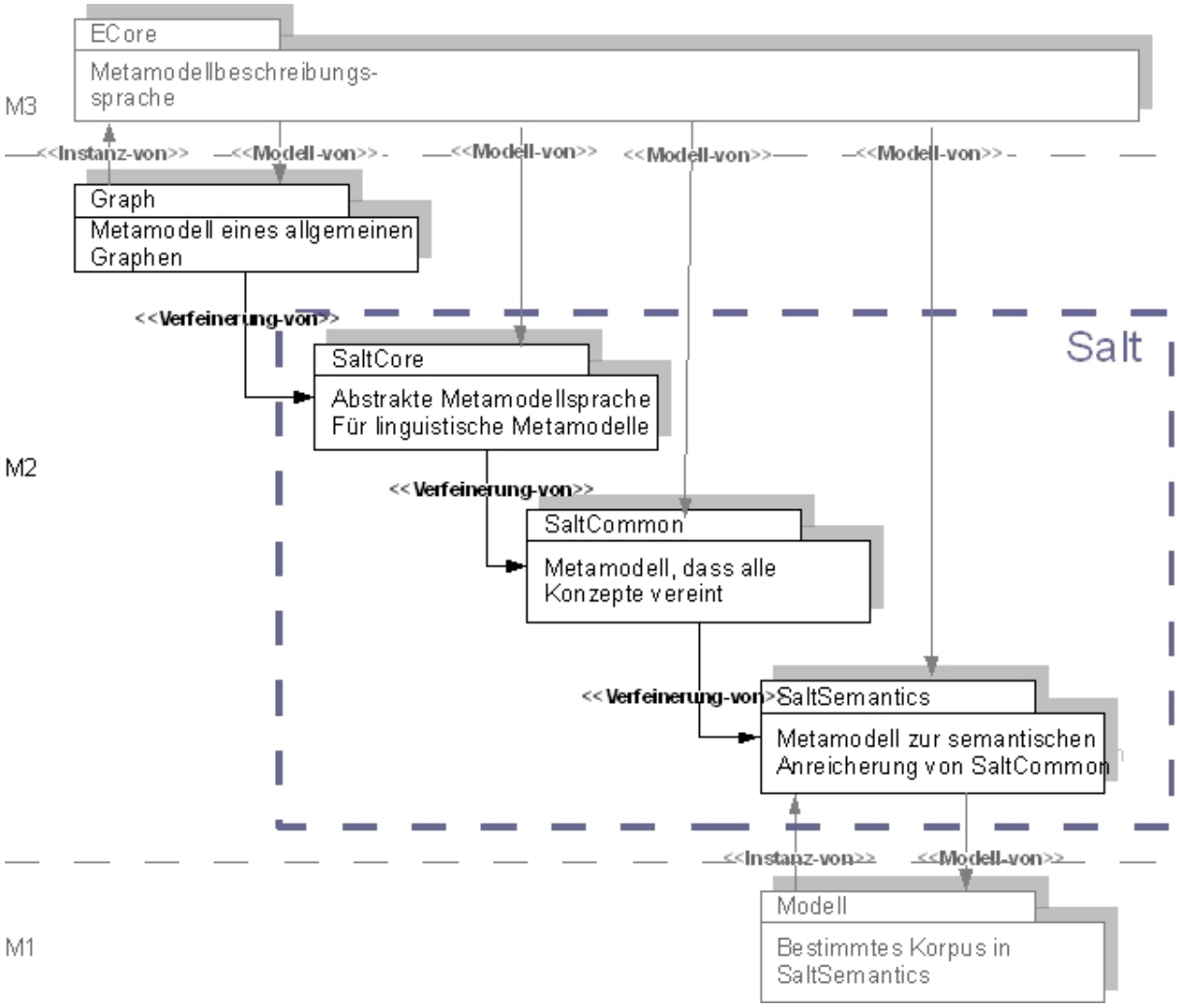


Abbildung 4.1.: Ableitungshierarchie von `Salt` aus dem allgemeinen Graphen und den einzelnen Metamodellen von `Salt`

Theorien abdecken soll, habe ich versucht an dieser Stelle eine Trennung von Struktur und Semantik vorzunehmen. `SaltCommonMM` ist daher nur der strukturelle Teil des gemeinsamen (Meta-)Modells und beinhaltet die Möglichkeit der Darstellung struktureller Einheiten und eine generelle Annotierbarkeit. Eine völlige Ignoranz der Semantik der Annotationen führt jedoch an verschiedenen Stellen zu großen Problemen. Bei der späteren Erzeugung von Konvertierungen kann die Semantik nicht mehr vollkommen außer acht gelassen werden. Dies begründet sich darin, dass manche der untersuchten Formate eine Annotationssemantik beinhalten. Einige der vorgestellten Konzepte sind daher semantischer Natur (siehe Wort-Semantik-Konzept). Um dennoch eine Trennung auf Modellebene zu ermöglichen, habe ich den annotationssemantischen Teil von `SaltSemanticsMM` , von dem strukturellen Teil getrennt. `SaltSemanticsMM` ist eine Verfeinerung des `SaltCommonMM` .
Aufgrund der entstandenen Ableitungshierarchie entspricht ein Modell eines der Metamodelle bspw. (`SaltSemanticsMM`) immer noch der allgemeinen Graphstruktur. Deshalb

können Algorithmen, die auf allgemeinen Graphen arbeiten, ebenfalls auf ein solches Modell angewendet werden.
Ich habe sowohl den allgemeinen Graphen als auch `Salt` modellbasiert mit EMF entwickelt. In den folgenden vier Abschnitten werde ich den Graphen (`GraphMM`) und die einzelnen Metamodelle von `Salt` : `SaltCoreMM` , `SaltCommonMM` und `SaltSemanticsMM` vorstellen.

4.1. `GraphMM`

In dieser Arbeit werde ich linguistische Strukturen als Knoten und Kanten repräsentieren. Damit können diese Strukturen durch das Modell eines allgemeinen *Graphen* dargestellt werden. Linguistische Annotationen könnnen dann als Labels eines Knotens oder einer Kante repräsentiert werden. Hierfür ist eine Anpassung des allgemeinen *Graphen* nötig, daher habe ich ihn wie folgt erweitert.

Definition 1 (erweiterter Graph)
Sei $G = (V, E, L, LAYER)$ *ein Graph mit:*

- *einer Menge* V *von Knoten,*
- *einer Menge* $E \subseteq V \times V$ *von gerichteten Kanten (*$e = (v_1, v_2)$*)*[1]*,*
- *einer Menge* $LAYER$ *von Schichten und*
- *einer Menge* L *von vergebenen Labels.*

Sei $layer : (V \bigcap E) \rightarrow LAYER$ *eine Abbildung, die einem Knoten* $v \in V$ *und einer Kante* $e \in E$ *eine Schicht* $layer \in LAYER$ *zuordnet.*
Weiter sei $l : (V \bigcap E) \rightarrow L$ *eine Abbildung, die einem Knoten* $v \in V$ *und einer Kante* $e \in E$ *ein Label* $l \in L$ *zuordnet.*

Wie aus der Motiavation (Abschnitt 1.1) hervorgeht, ist ein zentraler Punkt dieser Arbeit, das gemeinsame (Meta-)Modell modellbasiert zu entwickeln. Das macht es notwndig, dass auch die Grundbausteine des gemeinsamen (Meta-)Modells modellbasiert nutzbar sind. Die Technik, die ich für die Realisiserung des modellbasierten Ansatzes verwendet habe, ist EMF (siehe Abschnitt 2.4). Daher ist es erforderlich, dass der erweiterte Graph in dieser Technik modelliert ist. Um diesen Ansprüchen zu genügen, habe ich hierfür ein Metamodell entwickelt, dessen vereinfachte grafische Darstellung ich in Abbildung 4.2 zeige.

Das Metamodell `GraphMM` besteht im Kern aus den drei Modellelementen „Graph“, „Node“ und „Edge“. Das Element „Graph“ ist die Sammlung aus Knoten und Kanten. Das „Node“-Element steht für Knoten, das Element „Edge“ steht für Kanten und verfügt über die Attribute „source“ und „target“ über einen Quell- und einen Zielknoten. Jedem

[1] v_1 wird als Quellknoten und v_2 als Zielknotenbezeichntt

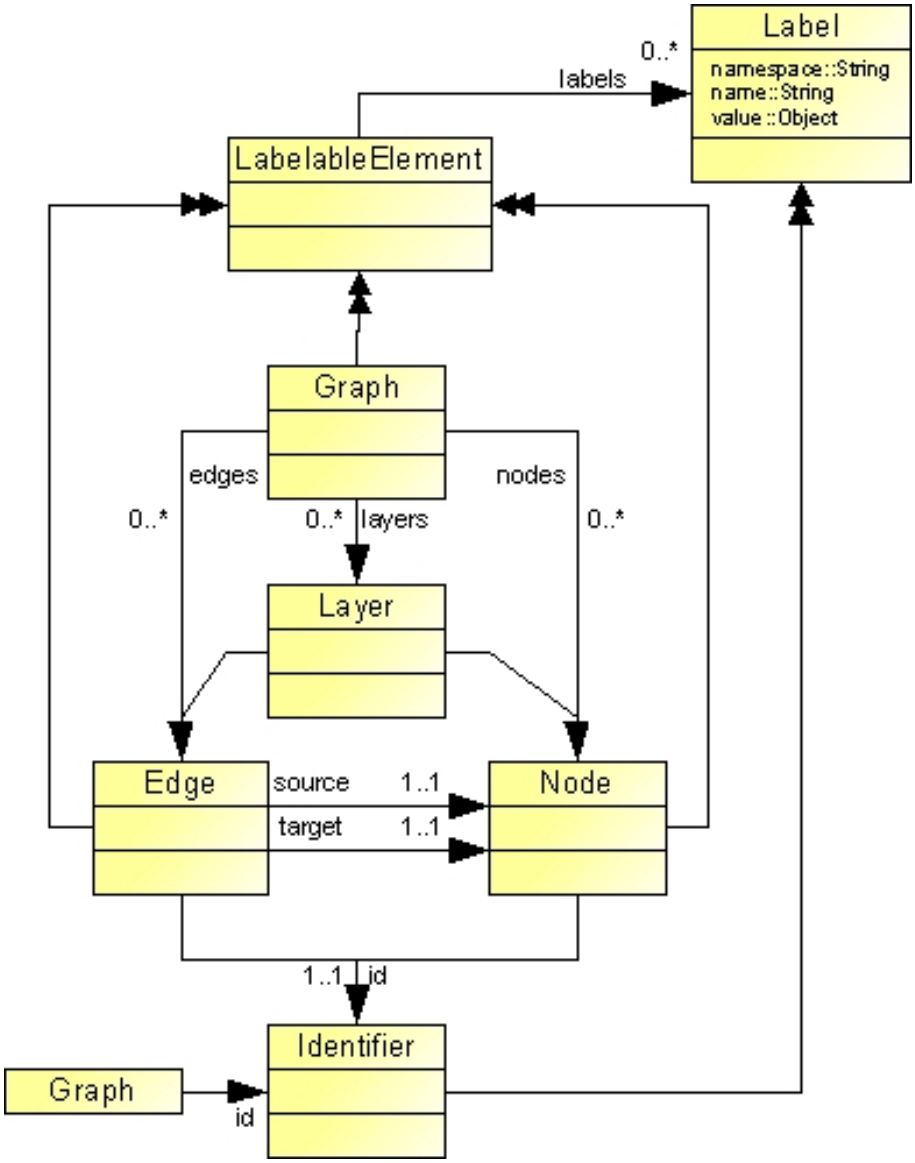


Abbildung 4.2.: Metamodell für den erweiterten Graphen. Das Element „Graph“ ist der besseren Veranschaulichung halber zweimal dargestellt, steht aber für das selbe Element.

der drei Elemente können beliebige Labels in Form des „Label“-Elementes und ein eindeutiger Identifizierer in Form des „Identifier“-Elementes zugeordnet werden. Das Element „Label“ besitzt die Attribute „namespace“, „name“ und „value“. Ein „Label“-Objekt ist somit ein Attribut-Wert-Paar mit einem Namensraum, über den Attribute näher spezifiziert werden können. So kann bspw. ein „Knoten“-Objekt zwei „Label“-Objekte mit dem gleichen Namen, aber einem anderen Namensraum besitzen. Mit dem „Layer“-Element können Knoten und Kanten zu Schichten gruppiert werden. Dadurch können Teile des Graphen als zusammengehörend bzgl. einer bestimmten Deutung klassifiziert werden. Ein Knoten oder eine Kante kann zu mehreren Schichten gehören. Eine Schicht kann über ein „Identifier“-Objekt eindeutig identifiziert werden.

Durch die Modellierung in EMF entsteht neben einer *API* auch eine native *XML*-Persistenzierung eines Graphmodells. Abbildung 4.3 zeigt einen Graphen und die entsprechende Persistenzierung durch EMF.

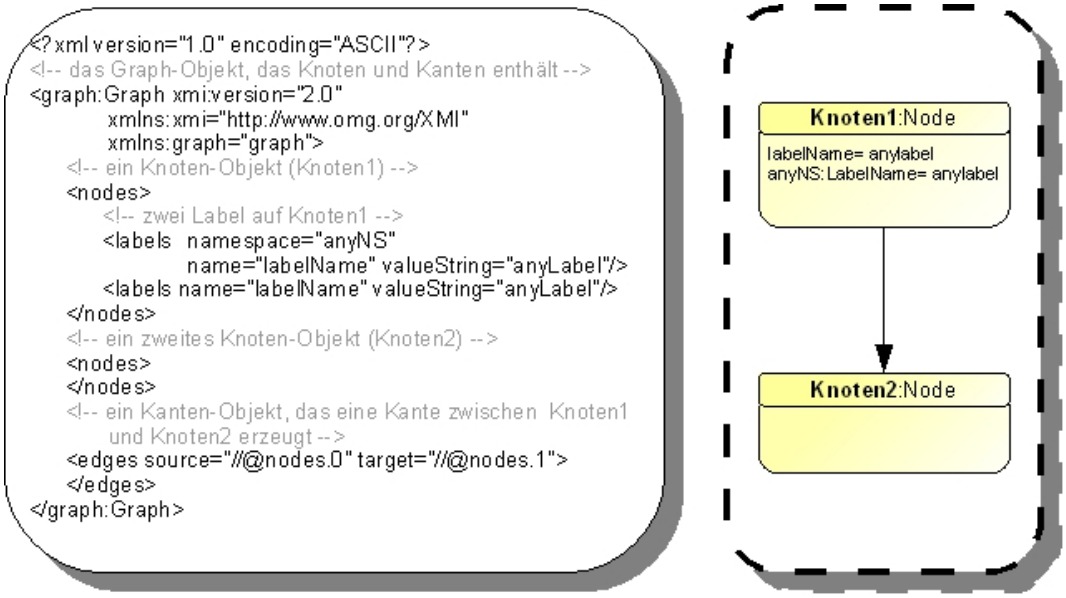
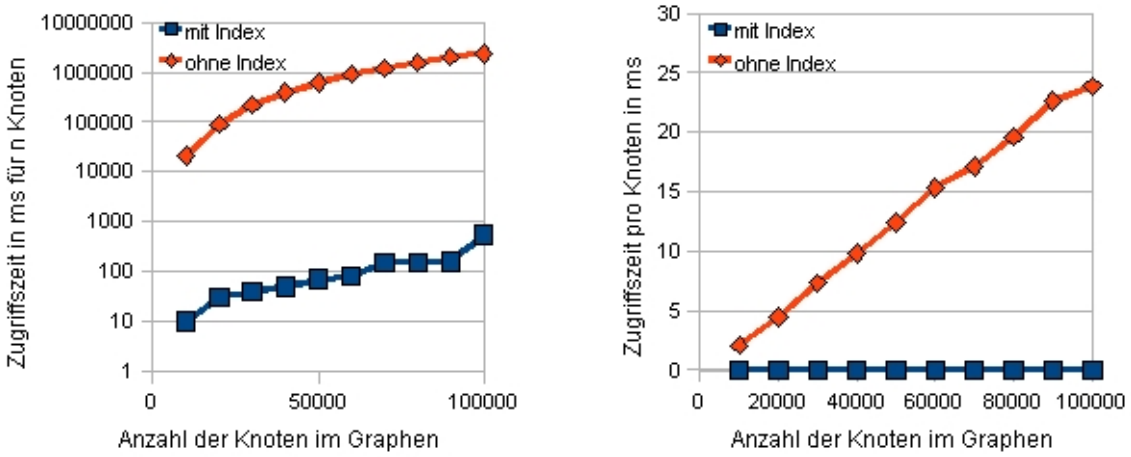


Abbildung 4.3.: links die native Persistenzierung durch EMF und rechts der dadurch abgebildete Graph, bestehend aus zwei Knoten, zwei Labels auf einem Knoten und einer Kante zwischen den Knoten

Bei der nativen Persistenzierung in *XML* durch EMF wird eine Inklusionsbeziehung[2] durch die *Baum*-Hierarchie von *XML* ausgedrückt. Eine „lose“ Referenz wird hingegen durch einen durch EMF definierten proprietären Adressierungsmechanismus erzeugt. In Abbildung 4.3 ist dieser Mechanismus bspw. durch das Attribut „source='//@nodes.0'“ aufgeführt. Diese Notation ist eine Referenz auf das erste „nodes“-Element des aktuellen *XML*-Dokuments.
Die native Persistenzierung durch EMF gehört unter anderem aufgrund dieser proprietären Art der Adressierung nicht zu den standardkonformsten oder „schönsten“ Möglichkeiten einer *XML*-Speicherung. Daher ist es ein Vorteil, dass das Eclipse Modeling Framework die Perstistenzierung von dem eigentlichen Metamodell abkapselt. Daher lässt sich die Persistenzierung austauschen, ohne dass das Metamodell davon betroffen ist. Die Speicherung könnte bspw. in einem standardisierten Graphformat (das die gleiche Mächtigkeit besitzt) vorgenommen werden. Über diese Technik oder über einen Adapter könnte das hier entwickelte proprietäre Metamodell `GraphMM` gegen ein standardisiertes ausgetauscht werden. Ein Beispiel auf diesem Gebiet ist das Graphenframework JUNG (siehe [JUNG, 2009]).
An dieser Stelle möchte ich noch einmal kurz auf die modellbasierte Entwicklung mit EMF und die Performanz eingehen. Grundsätzlich werden durch EMF für Referenzen mit der Kardinalität „0..*“ oder „1..*“ Listen erzeugt, die die entsprechenden Objekte

[2]Inklusionsbeziehung, auch „Containment“ genannt, bedeutet, dass ein Element Teil eines anderen Elementes ist.

enthalten. Ein Zugriff auf die Objekte dieser Listen kann mit zunhemender Listengröße ein ernstzunehmender Faktor für die Performanz werden. Um ein bestimmtes Objekt aus einer Liste zu identifizieren, muss im schlechtesten Fall die gesamte Liste durchsucht werden. Deshalb habe ich das Metamodell `GraphMM` mit einer Möglichkeit versehen, verschiedene Indizes einzufügen. Im Rahmen dieser Arbeit habe ich einen simplen Index auf die Identifizierer der „Node“ und „Edge“-Objekte in Form einer *Hashtable* erzeugt[3]. Dieser wird immer dann genutzt, wenn ein Knoten oder eine Kante mit dem entsprechenden Identifizierer gesucht wird. Ebenfalls wird darüber die Menge der eingehenden bzw. ausgehenden Kanten zu einem Knoten bestimmt. Abbildung 4.4(a) zeigt den Zeitbedarf für den Zugriff auf n Knoten mit und ohne die Nutzung des Index. In dieser Abbildung bedeutet bspw. $n = 5000$, dass die Liste 5000 Knoten enthält. Auf der y-Achse ist die Zeit abzulesen, die benötigt wird, um alle n Knoten in der Liste zu identifizieren. Abbildung 4.4(b) zeigt die Zeit, die benötigt wird, um einen Knoten zu identifizieren, bei steigender Anzahl an Knoten in der Liste. Anhand dieser Abbildung ist abzulesen, dass der Zeitbedarf ohne Indexnutzung linear zu der Anzahl an Knoten in der Liste steigt, während der Zeitbedarf mit Nutzung eines Index relativ konstant bleibt. Der Zeitbedarf beider Zugriffsmethoden wurde auf einem Laptop (Dell Inspiron 510m) mit einer CPU mit 1,6 GHz (Intel Pentium M) und 2 GB RAM gemessen.



(a) total (b) pro Knoten

Abbildung 4.4.: Abbildung 4.4(a) zeigt den Zeitbedarf (logarithmisch dargestellt) für die Identifikation von n Knoten in einer Liste der Länge n. Abbildung 4.4(b) zeigt den durchschnittlichen Zeitbedarf für den Zugriff auf einen Knoten bei steigender Listenlänge.

Um ein performanteres Verhalten mit EMF zu ermöglichen, ist es an einigen Stellen sinnvoll statt der automatisch erzeugten Listen andere Möglichkeiten wie bspw. *Hashtables* zu verwenden.

[3]Ich habe in dem Metamodell `GraphMM` die Möglichkeit vorgesehen, beliebige Indexstrukturen zu integrieren, daher ist ein Index basierend auf einer *Hashtable* nicht zwingend.

4.2. SaltCoreMM

SaltCoreMM ist eine Verfeinerung des Metamodells GraphMM . Es ist eine erste Annäherung an die Anforderungen eines linguistischen Metamodells. SaltCoreMM ist so allgemein gehalten, dass es für viele unterschiedliche linguistische Metamodelle genutzt werden kann. Bspw. könnte neben SaltCommonMM auch das Tiger Metamodell als eine Verfeinerung von SaltCoreMM modelliert werden. Dies hätte den Vorteil, dass alle Algorithmen, die auf SaltCoreMM angewendet werden können auch für das Tiger Metamodell nutzbar wären. Abbildung 4.5 zeigt eine vereinfachte Darstellung des Metamodells SaltCoreMM .

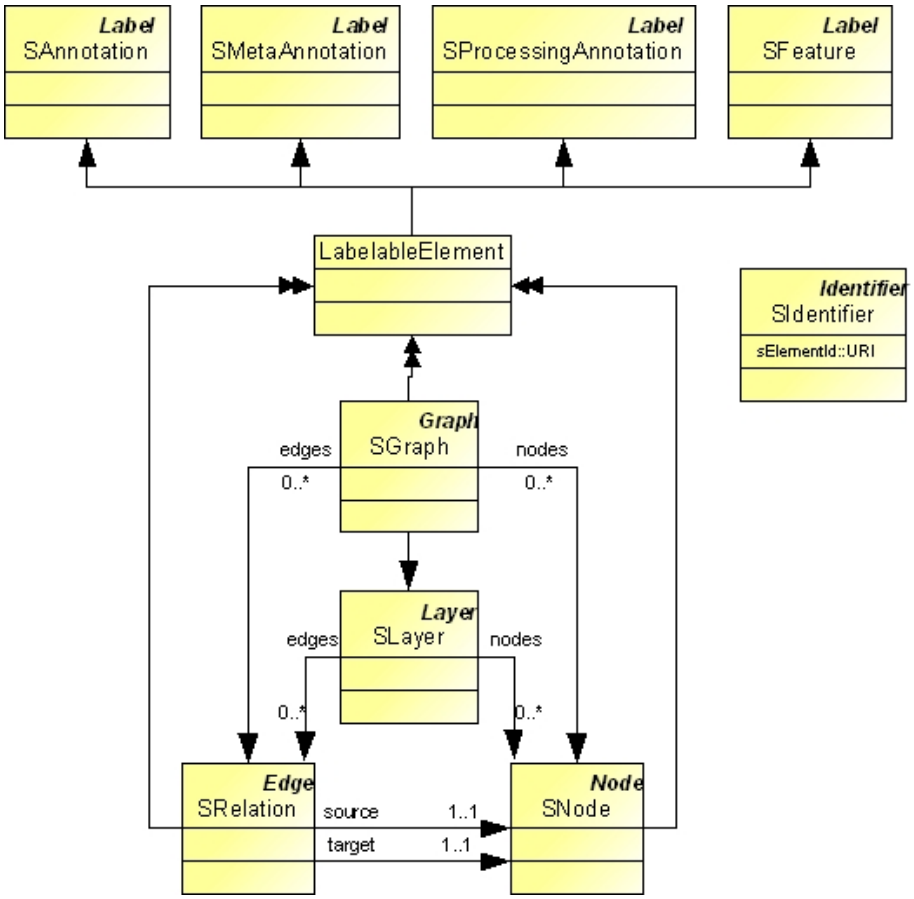


Abbildung 4.5.: Metamodell SaltCoreMM . Hier habe ich eine nicht EMF-konforme, aber ebenfalls gebräuchliche Notation für eine Vererbungsbeziehung benutzt. Dabei wird das vererbte Element rechts über den Namen des erbenden Elementes geschrieben. Bspw. erbt das Element „SGraph“ von dem Element „Graph“ aus GraphMM . Diese Notation dient der einfacheren Darstellung.

Die Elemente „SGraph“, „SNode“, „SRelation“ und „SLayer“ sind Verfeinerungen der entsprechenden Elemente[4] aus dem Metamodell GraphMM . Die Verfeinerung wird durch den Elementnamen des verfeinerten Elementes rechts über dem Namen des verfeinern-

[4]diese sind „Graph“, „Node“, „Edge“ und „Layer“

den Elementes angezeigt (siehe Bildunterschrift von Abbildung 4.5). Mit den Elementen „SGraph“, „SNode“ und „SRelation“ können strukturelle Einheiten und Relationen zwischen diesen repräsentiert werden. Bspw. kann ein „SGraph“-Objekt für ein Dokument, ein „SNode“-Objekt für ein Token, eine Spanne oder eine rekursive Einheit und ein „SRelation“-Objekt für eine beliebige Relation zwischen den strukturellen Einheiten stehen. Die Elemente „SAnnotation“, „SMetaAnnotation“, „SProcessingAnnotation“ und „SFeature“ sind Verfeinerungen des Elementes „Label“ aus dem Metamodell `GraphMM` . Ich habe das „Label“-Element verfeinert, um eine unterschiedliche Aussagekraft der verfeinerten Elemente zu erzielen.

Umsetzung des Attribut-Wert-Paar-Konzept

> Das Element „SAnnotation“ steht allgemein für linguistische Annotationen und bietet mit den aus dem Element „Label“ abgeleiteten Attributen „namespace“, „name“ und „value“ die Möglichkeit, eine beliebige strukturelle Einheit oder Relation mit einem Attribut-Wert-Paar zu annotieren. Da jedes Element des Typs „SGraph“, „SNode“ oder „SRelation“ mit einer linguistischen Annotation in Form eines Attribut-Wert-Paars annotiert werden kann, wird so das Attribut-Wert-Paar-Konzept und dessen Subkonzepte realisiert.

Umsetzung des Metaannotationskonzept

> Das Metaannotationskonzept wird durch „SMetaAnnotation“-Element dargestellt. Wie auch bei dem „SAnnotation“-Element kann jedes Element des Typs „SGraph“, „SNode“ oder „SRelation“ mit einem „SMetaAnnotation“-Element versehen werden. Daher wird auch das Metaannotationskonzept und dessen Subkonzepte durch `SaltCoreMM` realisisiert.

Die beiden Elemente „SProcessingAnnotation“ und „SFeature“ dienen keinem linguistischen Zweck. Mit „SProcessingAnnotation“-Objekten können die später beschriebenen Konverter während des Konvertierungsprozesses eine strukturelle Einheit mit speziellen Informationen versehen. Diese sind nicht relevant für das Modell, sondern nur für den Konvertierungsprozess. Das Element „SFeature“ bietet die Möglichkeit, einem Element Attribute zu geben. Bspw. besitzt, wie in Abschnitt 4.3 gezeigt, das Element „STextualDS“ (das für die Repräsentation des Primärtextes steht) ein Attribut „sText“. Dieses enthält den Primärtext. Damit dennoch die Graphstruktur bewahrt bleibt, habe ich solche Attribute über ein „SFeature“-Element modelliert. So werden sie im Endeffekt als Labels eines Knoten, einer Kante oder eines Graphen abgelegt. Damit bekommen allgemeine Graphalgorithmen einen Zugriff auf diese Attribute.

Umsetzung des Schichtenkonzept

> Durch das „SLayer“-Element können Knoten und Kanten zu Schichten gruppiert werden. Diese können dafür genutzt werden, einzelne Aspekte einer

linguistischen Korpusanalyse zusammenzufassen. Bspw. kann ein „SLayer“-Objekt alle „SNode“- und „SEdge“-Objekte gruppieren, die zu einer linguistischen Syntaxanalyse gehören. Durch das „SLayer“-Element wird das Schichtenkonzept realisiert.

In Abschnitt 4.3 werde ich die Verfeinerung `SaltCommonMM` des Metamodells `SaltCoreMM` vorstellen.

4.3. `SaltCommonMM`

Das Metamodell `SaltCommonMM` ist aufgrund der vielen unterstützten Konzepte recht umfangreich und schwierig als Ganzes darszustellen. Zum leichteren Verständnis werde ich `SaltCommonMM` im Folgenden ausschnittsweise darstellen. Ich werde dabei die in Abschnitt 3.2 vorgestellten strukturellen Konzepte, die durch `SaltCommonMM` unterstützt werden, als strukturelle Grundlage nehmen und zeigen, wie die Konzepte umgesetzt wurden. An einigen Stellen werde ich das Metamodell etwas ausführlicher darstellen als es das Konzept verlangt. Mit diesen Darstellungen möchte ich versuchen, einen tieferen Einblick in die Mächtigkeit von `SaltCommonMM` zu geben. Auf eine Darstellung des gesamten Metamodells möchte ich hier verzichten, da eine ausführliche Darstellung mit hoher Detailtiefe den Rahmen dieser Arbeit sprengen würde.

Umsetzung des Primärtextkonzepts und des Zeitkonzepts

Für diese Arbeit betrachte ich einen Primärtext als eine sequentielle Datenquelle. Ein hat eine Ordnung auf Basis der enthaltenen Zeichen. Zwei Zeichen a und b stehen entweder vor oder nacheinander, solange $a \neq b$. Ich betrachte die Zeit als kontinuierlich und linear, damit existiert hier ebenfalls eine klare Ordnung zwischen zwei Zeitpunkten. Außerdem stellt die Zeit analog zu einem Primärtext eine Datenquelle dar. Diese enthält keine Zeichen sondern Zeitpunkte.
In `SaltCommonMM` habe ich das Element „SSequentialDS“ (DS steht für data source) für sequentielle Datenquellen vorgesehen. Dieses besteht aus einem Attribut „sData“, in dem das sequentielle Datum abgelegt wird. Dieses kann allgemein jede sequentielle Art von Datum sein, hier jedoch meist ein Text oder ein Zeitabschnitt. Das Element „SSequentialDS“ ist eine Verfeinerung des Elements „SNode“ aus `SaltCoreMM` und damit als Knoten im Graphen erreichbar. Das Attribut „sData“ ist im Endeffekt ein Label dieses Knotens.
Die beiden Elemente „STextualDS“ und „STimeline“ sind Ableitungen der sequentiellen Datenquelle und dienen der Speicherung eines Primärtextes bzw. einer gemeinsamen Zeitlinie. Ein Primärtext wird durch das Attribut „sText“ gespeichert, das eine Ableitung des Attributes „sData“ aus dem Element „SSequentialDS“ ist. Die gemeinsame Zeitline wird durch das Attribut „sPointsOfTime“ auf gleiche Weise gespeichert. „STextualDS“ und „STimeline“ sind aufgrund der Verfeinerung auch vom Typ „SNode“. Abbildung 4.6 zeigt einen

Ausschnitt aus SaltCommonMM , bezogen auf die Modellierung der Datenquellen.

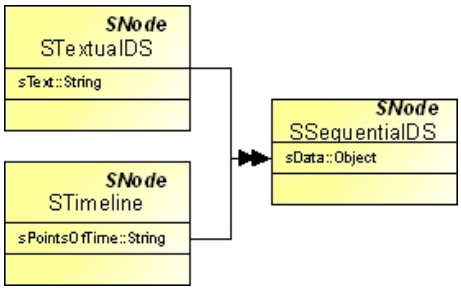


Abbildung 4.6.: Realisierung des Primärtextkonzepts und des Zeitkonzepts in SaltCommonMM

Umsetzung des Tokenkonzepts

Ein Token dient in SaltCommonMM , nicht nur der Aufteilung eines Primärtextes, sondern allgemein der Aufteilung einer Datenquelle in kleinste Einheiten. Das Element „SToken“ ist eine Verfeinerung des Elementes „SNode“ und besitzt keine Attribute. Über eine Relation kann es mit einer Datenquelle verbunden werden. Eine solche Relation stellt einen Ankerpunkt in einer Datenquelle dar, indem sie angibt, auf welchen Bereich der Datenquelle sich das Token bezieht. Für die Verknüpfung mit einer sequentiellen Datenquelle existiert in SaltCommonMM das Element „SSequentialRelation“. Dieses Element ist eine Verfeinerung des Typen „SRelation“ aus SaltCoreMM . Es markiert über die Attribute „sStart“ und „sEnd“ einen addressierten Bereich in der Datenquelle. Durch die Verknüpfung eines Tokens mit einer Datenquelle über eine Relation und der Adressierung des Bereiches in dieser Relation, kann ein Token mit beliebig vielen Datenquellen unterschiedlichen Typs verknüpft werden. Abbildung 4.7 stellt die Verknüpfung zwischen „SToken“, „STextualDS“ und „STimeline“ dar.

Umsetzung des Multitextkonzepts

Die Anzahl der „STextualDS“-Objekte pro Dokument ist in SaltCommonMM nicht beschränkt. Dadurch können einem Dokument beliebig viele Primärtexte zugewiesen werden. Annotationsstrukturen können sich dann auf einen oder auf mehrere Primärtexte beziehen. Abbildung 4.8 zeigt die Referenz eines Dokumentes zu einem Primärtext. Die Beziehung beider Elemente ist implizit über die Verfeinerung aus SaltCoreMM gegeben.

Umsetzung des Spannenkonzepts

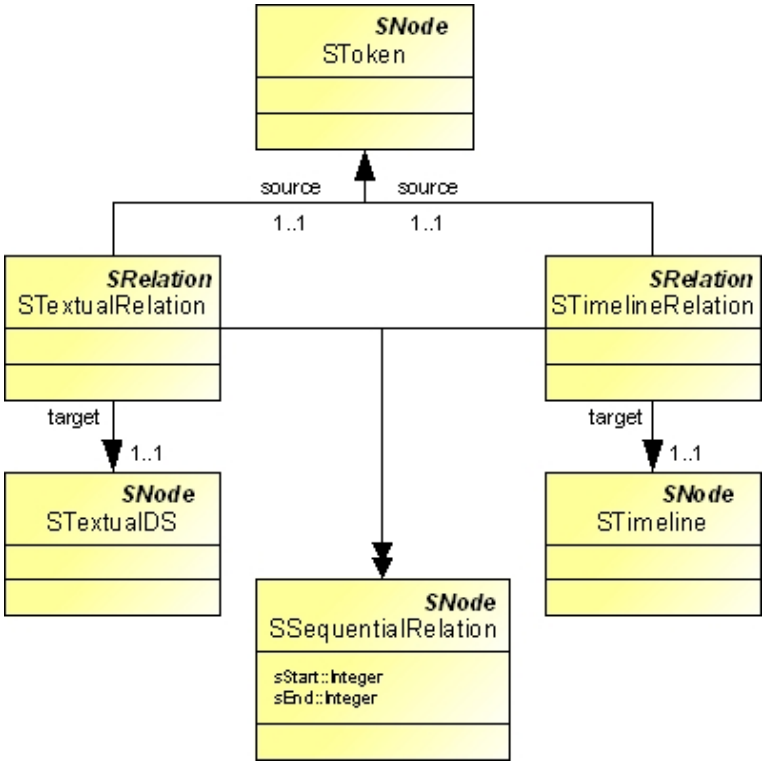


Abbildung 4.7.: Realisierung des Tokenkonzepts in `SaltCommonMM`

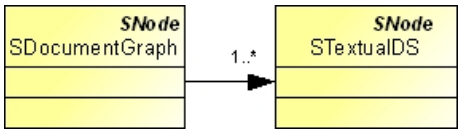


Abbildung 4.8.: Realisierung des Multitextkonzepts in `SaltCommonMM`

In `SaltCommonMM` steht das Element „SSpan“ für die strukturelle Einheit Spanne. Token können dadurch zu Spannen zusammengefasst werden. Da `SaltCommonMM` auf einem Graphen basiert und somit aus Knoten und Kanten besteht, können Token nicht, wie in PAULA oder EXMARaLDA, als Teile einer Spanne, z.B. durch Attribute, modelliert werden. Das „SToken“- und „SSpan“-Element sind von dem Typen „SNode“ abgeleitet und müssen durch eine Relation verbunden werden. Hierfür habe ich den Typ „SSpanningRelation“ erzeugt, der die Beziehung einer Spanne zu einem Knoten ausdrückt. Das Element „SSpanningRelation“ ist eine Verfeinerung des Typen „SRelation“ aus `SaltCoreMM` . Es ist nicht erforderlich, dass die referenzierten Token bezogen auf eine Datenquelle aufeinanderfolgend sein müssen. Daher werden diskontinuierliche Spannen ebenfalls unterstützt. Abbildung 4.9 zeigt die Umsetzung des Spannenkonzepts in einem vereinfachten Ausschnitt des Metamodells `SaltCommonMM` .

Umsetzung des Hierarchiekonzepts

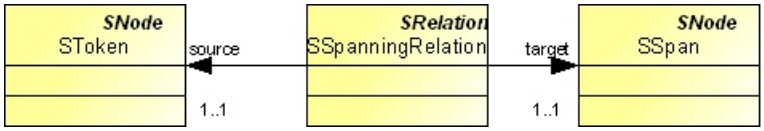


Abbildung 4.9.: Realisierung des Spannenkonzepts in `SaltCommonMM`

In `SaltCommonMM` wird die Möglichkeit Hierarchien zu erzeugen, und damit das Hierarchiekonzept , durch die Relation „SDominanceRelation“ realisiert. Ein „SDominanceRelation“-Objekt besitzt als Quellknoten ein Objekt des Typs „SStructure“. Dieses Element steht in `SaltCommonMM` für eine rekursive Einheit. Als Zielknoten der Relation „SDominanceRelation“ kann entweder ein Token („SToken“-Objekt), eine Spanne („SSpan“-Objekt) oder eine rekursive Einheit („SStructure“-Objekt) dienen. Durch ein „SStructure“-Objekt als Zielknoten kann eine rekursive Einheit erzeugt werden. Um diesen Zusammenhang in dem Metamodell `SaltCommonMM` darzustellen, besteht zwischen den Elementen „SToken“, „SSpan“ und „SStructure“ und dem Element „SStructuredNode“ eine Vererbungsbeziehung. Dadurch wird gewährleistet, dass ein „SDominanceRelation“-Objekt immer nur einen Zielknoten haben kann.[5] Abbildung 4.10 zeigt die Umsetzung des Hierarchiekonzepts .

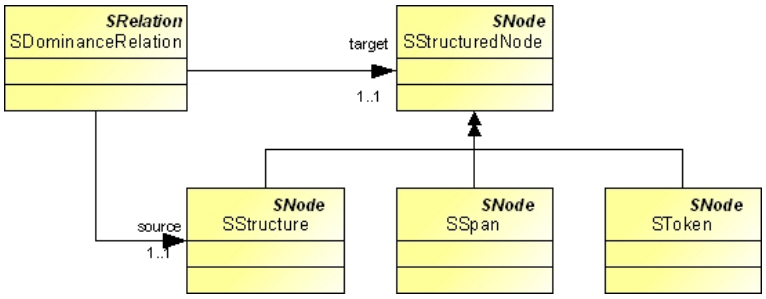


Abbildung 4.10.: Realisierung des Hierarchiekonzepts in `SaltCommonMM`

Umsetzung des Nicht-AVR-Konzepts

Ähnlich wie in den Formaten relANNIS und PAULA gibt es auch in `SaltCommonMM` eine Relation, die das Konzept Nicht-AVR-Konzept realisiert. Diese Relation wird durch das Element „SPointingRelation“ dargestellt. Ein „SPointingRelation“-Objekt kann sowohl als Quelle als auch als Ziel jede strukturelle Einheit zugewiesen bekommen, die dem Typ „SStructuredNode“ entspricht. Wie Abbildung 4.10 zeigt, sind dies die Elemente „SToken“, „SSpan“ und „SStructure“. Abbildung 4.11 zeigt die Modellierung des „SPointingRelation“-Elements.

[5] Dies wäre nicht gewährleistet, wenn die drei möglichen Zielknoten-Elemente direkt mit dem Element „SDominanceRelation“ in Beziehung gesetzt würden.

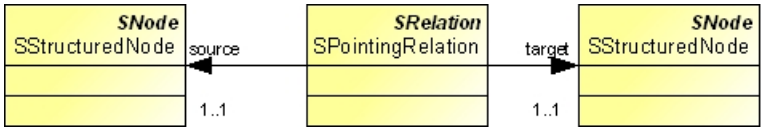


Abbildung 4.11.: Realisierung des Nicht-AVR-Konzepts in `SaltCommonMM`

In dem Metamodell `SaltCommonMM` habe ich die, in dem Konzept Nicht-AVR-Konzept beschriebene, Möglichkeit der Primärfragment- und Zeitintervall-Vererbung explizit modelliert. Wenn ein Algorithmus den Graphen traversiert, um die von einem Element enthaltenen Primärfragmente oder Zeitabschnitte zu ermitteln, kann er hierfür die Relationen „STextOverlapping“ bzw. „STimeOverlapping“ nutzen, die genau diese Semantik verkörpern. Relationen, die diese Semantik ausdrücken sollen, werden mit der entsprechenden Relation („STextualRelation“ bzw. „STimelineRelation“) in eine Vererbungs-Beziehung gesetzt. Dadurch ist es möglich, weitere Relationen in dem Metamodell `SaltCommonMM` zu erzeugen, die evtl. ebenfalls diese Semantik verkörpern ohne den Algorithmus anzupassen. Abbildung 4.12 zeigt den Zusammenhang der abstrakten Relationen „STextOverlapping“ bzw. „STimeOverlapping“ und der konkreten Relationen „SDominanceRelation“ und „SSpanningRelation“.

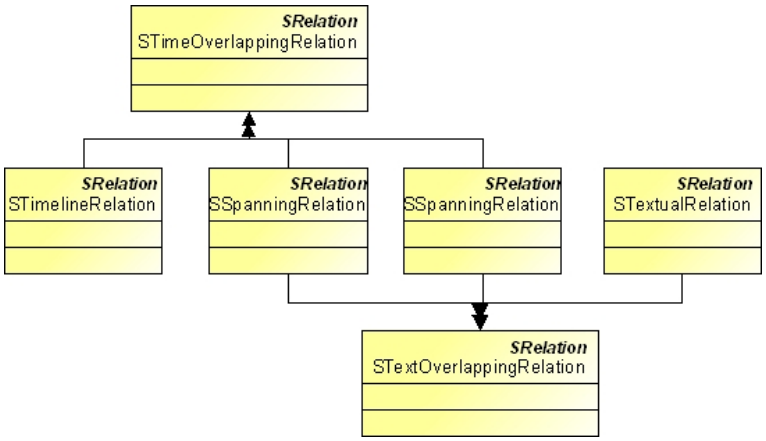


Abbildung 4.12.: Realisierung des Einfügens struktureller Semantiken, wie der Text- bzw. Zeitüberdeckung durch Vererbung.

Umsetzung des Dokument- und Korpuskonzepts

Das Metamodell `SaltCommonMM` unterstützt das Dokument- und Korpuskonzept durch die Elemente „SCorpus“ und “SDocument“. Ein „SCorpus“-Objekt repräsentiert ein Korpus und kann über die Relation „SCorpRelation“ mit einem anderen „SCorpus“-Objekt in Beziehung gesetzt werden. Diese Relation drückt eine Super- bzw. Subkorpus-Beziehung aus. Dadurch können Korpora in beliebiger Schachtelungstiefe erzeugt werden. Das Element „SDocument“ steht für ein Dokument und einen Graphen, der sämtliche Primärtexte, strukturelle Einheiten und Relationen beinhaltet. Ein Dokument kann über

die Relation „SDocCorpRelation“ mit einem Korpus in Beziehung gesetzt werden und ist dadurch Teil eines Korpus. Abbildung 4.13 zeigt die Realisierung des Konzepts Dokument- und Korpuskonzept .

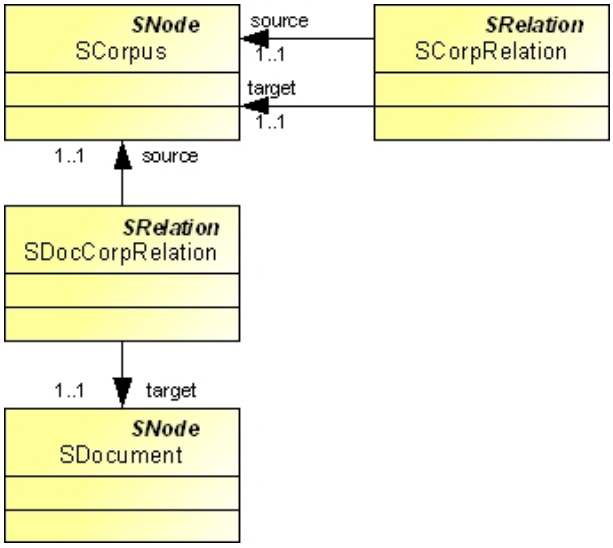


Abbildung 4.13.: Realisierung des Dokument- und Korpuskonzepts in `SaltCommonMM`

Sowohl die Korpusstruktur, als auch die Dokumentstruktur (in Form von Primärtexten, strukturellen Einheiten und Annotationen) sind in einem Graphen angeordnet. `SaltCommonMM` unterscheidet hier in einen Korpus-Graph und einen Dokument-Graph. Abbildung 4.14 zeigt die Realisierung des Korpus-Graphen durch das Element „SCorpusGraph“.

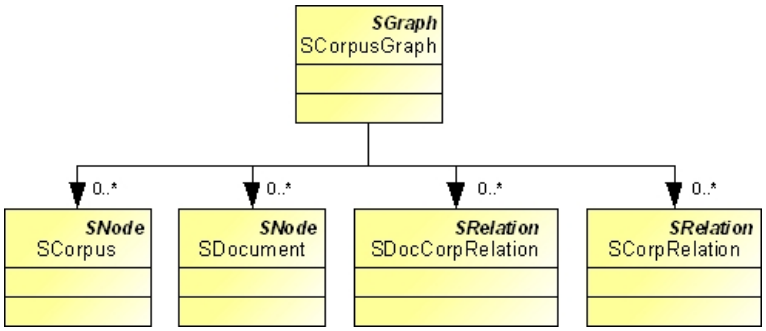


Abbildung 4.14.: Korpusstrukur in `SaltCommonMM`

Abbildung 4.15 zeigt die Realsierung des Inhalts eines Dokuments als Dokument-Graph durch das Element „SDocumentGraph“. Einem „SDocument“-Objekt ist ein „SDocumentGraph“-Objekt zugeordnet.

Die hier vorgestellte Modellierung von `SaltCommonMM` sorgt aufgrund der Vererbungshierarchie dafür, dass jedes Modell des Metamodells `SaltCommonMM` immer noch die Struktur eines Graphen besitzt und somit auch ein Modell des Metamodells `GraphMM` ist. Dieser Zusammenhang wird durch Abbildung 4.16 demonstriert. Das Beispiel zeigt

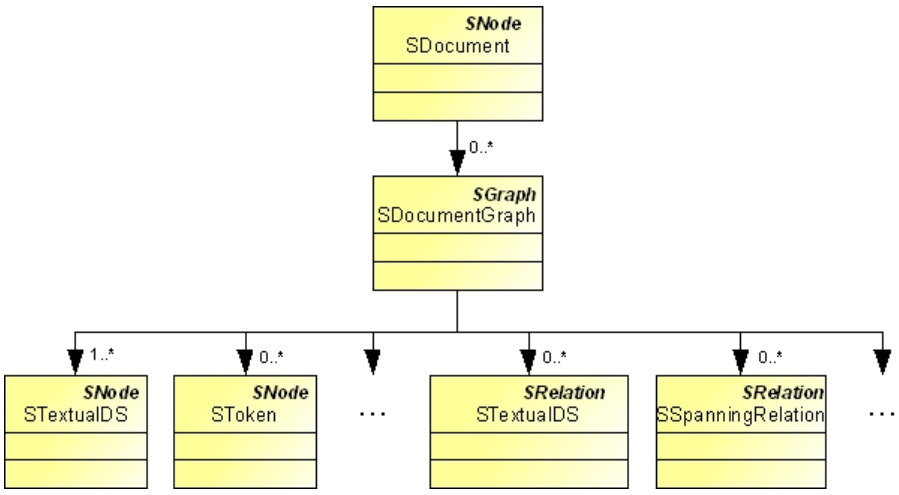


Abbildung 4.15.: Dokumentstrukur in `SaltCommonMM`

den Primärtext „Ich ging los“, der in drei Token „tok1“, „tok2“, „tok3“ zerteilt ist. Jedes dieser Token besitzt eine part-of-speech-Annotation. Zwei dieser Token („tok2“ und „tok3“) sind (über „SSpanningRelation“-Objekte) Teil der Spanne „span1“. Diese besitzt eine Lemma-Annotation. Die rekursive Einheit „struct1“ ist mit den Objekten „tok1“ und „span1“ durch „SDominanceRelation“-Objekte in Beziehung gesetzt. Die Spanne, sowie eine der Relationen sind ebenfalls annotiert. Dieses Beispiel veranschaulicht, dass ein Modell des Metamodells `SaltCommonMM` immer noch einem Graphen entspricht. Ein solches Modell besteht nach wie vor nur aus Knoten, Kanten und Labeln auf diesen.

In diesem Abschnitt habe ich die Umsetzung der strukturellen Konzepte durch das Metamodell `SaltCommonMM` gezeigt. Dieses Metamodell bildet die Grundlage des in Kapitel 5 vorgestellten Konverterframeworks. Obwohl das Metamodell `SaltCommonMM` mächtig genug ist, strukturell alle vorgestellten Konzepte zu unterstützen, kann es bei einer Transformation eines Modells aus einem Format, das bestimmte semantische Deutungen über die Strukturen beinhaltet, auf `SaltCommonMM` zu semantischen Informationsverlusten kommen. Um diese Lücke zu schließen und die fehlenden semantsichen Konzepte zu unterstützen, habe ich für diese Arbeit ein Metamodell entwickelt, das eine semantische Erweiterung für `Salt` ermöglicht. Das Metamodell `SaltSemanticsMM` kann optional bei der Konvertierung von Daten genutzt werden. Im folgenden Abschnitt werde ich den Gedanken hinter `SaltSemanticsMM` und dessen Vorteile vorstellen.

4.4. `SaltSemanticsMM`

Die bisher vorgestellten Metamodelle von `Salt` beinhalten nur eine strukturelle Sicht auf die Daten. Dadurch entsteht eine Unabhängigkeit von bestimmten linguistischen Theorien oder Interpretationsansätzen.
Einige der betrachteten Formate, nutzen eine linguistische Deutung der dargestellten Strukturen. Ein paar dieser Deutungen, die ein fester Bestandteil eines Formates sind, habe ich durch die Verfeinerung des Annotations-Semantik-Konzepts aus Abschnitt 3.2

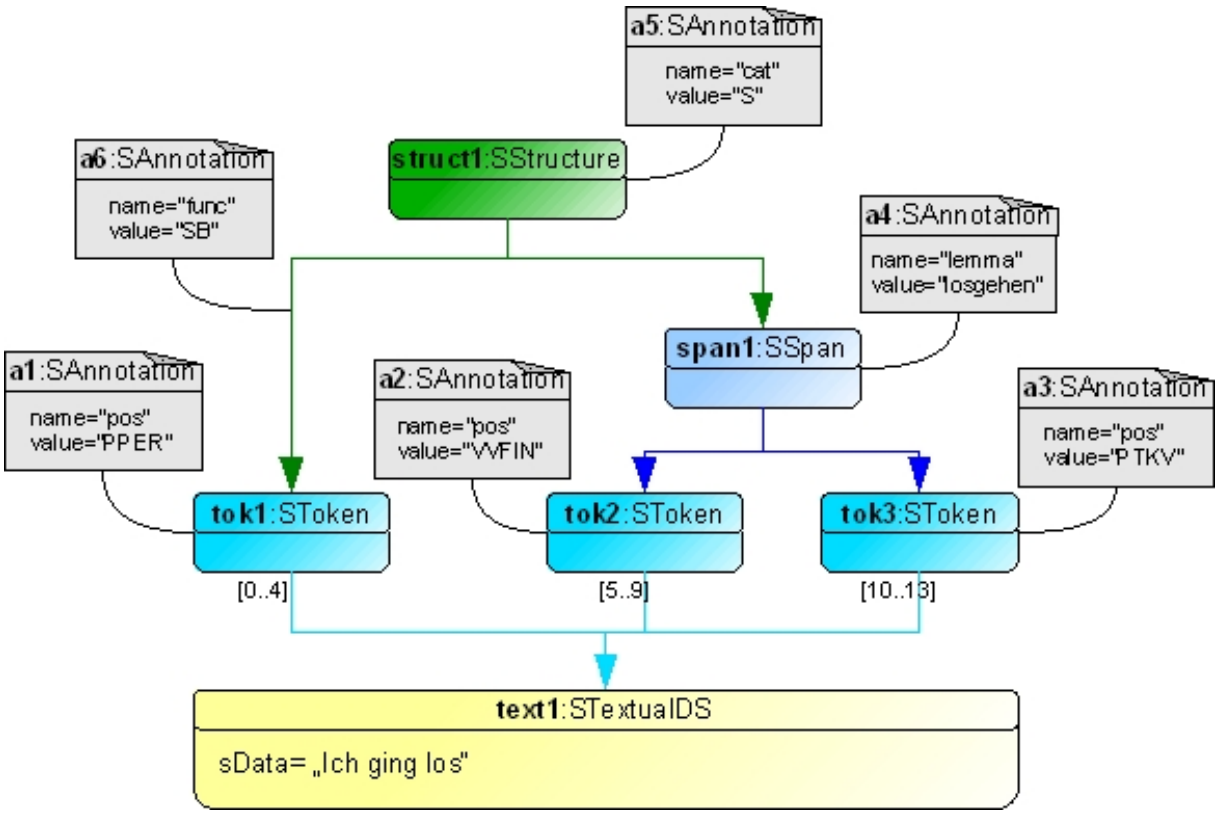


Abbildung 4.16.: Graphstruktur von Modellen im Metamodell `SaltCommonMM` , verdeutlicht durch ein Beispielkorpus

erfasst. Ein Beispiel ist das Wort-Semantik-Konzept , das davon ausgeht, dass einer bestimmten strukturellen Einheit die Bedeutung eines Wortes zukommt. Ein weiteres Beispiel ist die Annotation mit einer part-of-speech-Annotation. Beide Deutungen sind ein Bestandteil des TreeTagger-Formates und implizit in dem Schema des Formates enthalten (die erste Spalte im TreeTagger-Format beschreibt immer ein Wort, die zweite Spalte gibt die dazugehörige part-of-speech-Annotation an). Bei der Transformation eines Korpus aus dem TreeTagger-Format in das `SaltCommonMM` entsteht ein semantischer Verlust, da die Daten zwar dargestellt werden können, aber nicht explizit als Wort bzw. als part-of-speech-Annotation gedeutet werden. Bei der Rücktransformation entsteht dadurch ein Problem, das zu einem strukturellen Verlust führen kann. Angenommen TreeTagger-Daten enthalten den Eintrag „Ich PPER“, so bedeutet dies, dass das Wort „Ich“ mit einer part-of-speech-Annotation „PPER“ versehen ist. Bei der Transformation nach `SaltCommonMM` würde aus der Annotation bspw. das Attribut-Wert-Paar „part-of-speech=’PPER’“ werden. Genauso könnte der Annotationsname aber auch „POS“, „pos“ oder „p-o-s“ lauten. Bei der Rücktransformation ist daher nicht mehr eindeutig klar, dass es sich dabei ursprünglich um eine part-of-speech-Annotation handelte.
In Abschnitt 2.8 habe ich zur Lösung dieses Problems die Nutzung des Systems ISOCat vorgeschlagen. Dadurch könnte anstatt eines mehr oder weniger proprietären Annotationsnamens ein eindeutiger Identifizierer in Form einer *URI*-Referenz vergeben werden. Dieser Ansatz bietet zwei Vorteile: zum einen wird dadurch eine Trennung von strukturel-

lem Metamodell und der Deutung der Daten erreicht und zum anderen kann die Deutung der Daten in die linguistische Domäne verlegt werden. Im Rahmen dieser Arbeit kann ich ISOCat noch nicht verwenden, da es zur Zeit dieser Arbeit noch nicht über den nötigen Umfang an Datenpunkten verfügt. Dennoch besteht innerhalb des Metamodells `SaltCoreMM` die Möglichkeit anstelle von Annotationen in Form von Attribut-Wert-Paaren auf Zeichenkettenbasis eine *URI*-Referenz anzugeben.
Um dennoch eine deutbare Semantik bestimmter Annotationen zu erhalten, habe ich ein weiteres Metamodell entwickelt, das diese Aufgabe übernimmt. Das Metamodell `SaltSemanticsMM` verfeinert das Element „SAnnotation“, indem es den Annotationsnamen (das Attribut „SAnnotation.name“) festlegt. Abbildung 4.17 zeigt einen Ausschnitt aus `SaltSemanticsMM` .

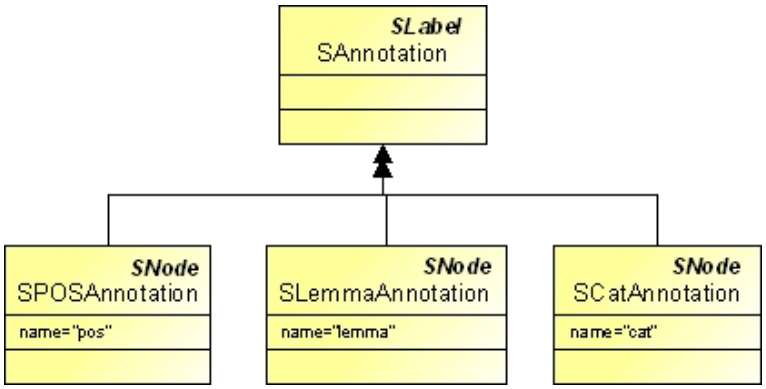


Abbildung 4.17.: Verfeinerung des „SAnnotation“-Elements, bspw. um eine eindeutige Deutung für part-of-speech-, Lemma- und Cat-Annotationen zu ermöglichen

Das Beispiel aus Abbildung 4.16 kann in `SaltSemanticsMM` wie in Abbildung 4.18 dargestellt werden.

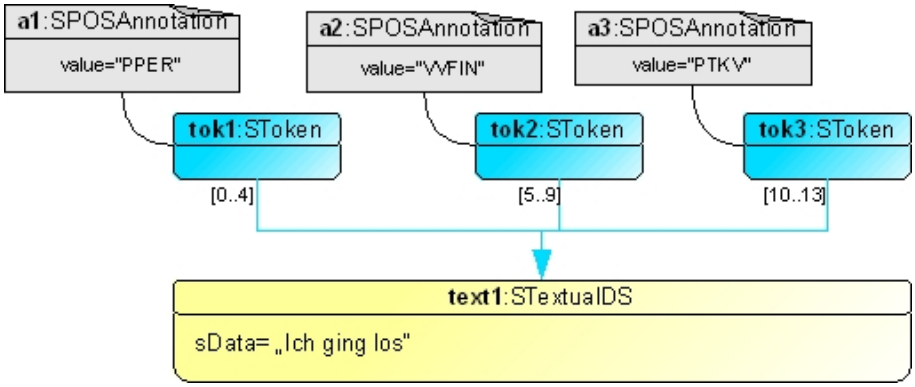


Abbildung 4.18.: Ausschnitt aus einem Beispielkorpus in `SaltSemanticsMM`

Durch diese Erweiterung und die damit verbundene Umsetzung des Annotations-Semantik-Konzepts durch das Metamodell bekommen Konvertierungswerkzeuge eine Möglichkeit, semantische Deutungen zu identifizieren. Bspw. bei der Transformation der part-of-speech-Annotation „PPER“ des Wortes „Ich“ kann eine Annotation des Typs „SPOSAnnotation“

genutzt werden. Damit kann die Rücktransformation ebenfalls auf der Grundlage dieses Typs erfolgen und ist damit eindeutig als part-of-speech-Annotation deutbar.
`SaltSemanticsMM` kann nicht als fertig betrachtet werden, da nur ein Bruchteil an spezifischen Annotationen hier aufgenommen ist. Dies sind genau die Deutungen, die aus den untersuchten Formaten hervorgehen und als Verfeinerungen des Annotations-Semantik-Konzepts in Abschnitt 3.2[6] beschrieben sind.
Mit der vorgestellten Trennung von Deutung und Struktur, wird der Vorteil, den ISOCat bietet: die Trennung von strukturellem Modell und der Deutung der Daten erreicht. Der zweite Vorteil, die Verlagerung der Deutung in die linguistische Domäne, wird hierdurch nicht erfüllt. Das Erzeugen einer neuen Annotation mit fester Deutung hat eine Änderung des Metamodells `SaltSemanticsMM` zur Folge.

In diesem Kapitel habe ich das gemeinsame (Meta-)Modell `Salt` vorgestellt. Dieses basiert auf der Grundlage eines allgemeinen Graphen, dan ich zunächst um Label und Schichten erweitert habe. In Abschnitt 4.2 habe ich ein abstraktes linguistisches Metamodell (`SaltCoreMM`) vorgestellt, das die strukturellen Konzepte Attribut-Wert-Paar-Konzept , Metaannotationskonzept und Schichtenkonzept realisiert. Das Metamodell `SaltCoreMM` habe ich im darauf folgenden Abschnitt als Basis für das Metamodell `SaltCommonMM` genutzt. Dieses realisiert die strukturellen Konzepte Zeitkonzept , Primärtextkonzept , Multitextkonzept , Tokenkonzept , Spannenkonzept , Hierarchiekonzept , Dokument- und Korpuskonzept und Nicht-AVR-Konzept . Eine semantische Anreicherung der Daten kann mit Hilfe des Metamodells `SaltSemanticsMM` erreicht werden. In diesem sind die semantischen Konzepte Wort-Semantik-Konzept , Satz-Semantik-Konzept , POS-Semantik-Konzept , Lemma-Semantik-Konzept und Cat-Semantik-Konzepts umgesetzt. Ich habe gezeigt, dass die Vereinigung der Metamodelle `GraphMM` , `SaltCoreMM` , `SaltCommonMM` und `SaltSemanticsMM` in Form von `Salt` alle Konzepte, die ich in Abschnitt 3.2 herausgearbeitet habe, umsetzt.

[6] dies sind die Konzepte Wort-Semantik-Konzept , Satz-Semantik-Konzept , POS-Semantik-Konzept , Lemma-Semantik-Konzept oder Cat-Semantik-Konzept

5. Entwicklung des Konverterframeworks

In diesem Kapitel stelle ich das Konverterframework `Pepper` vor, das ich im Rahmen dieser Arbeit für die Konvertierung von Korpora entwickelt habe.
Dieses Framework bildet die Plattform für die Integration von Mappings zwischen den untersuchten Formaten und dem gemeinsamen (Meta-)Modell `Salt` . `Pepper` ist damit der zentrale Kern für die praktische Umsetzung der Ziele dieser Arbeit. Nur durch die Entwicklung und Realisiserung dieses Frameworks und der dazugehörigen Mappings für die entsprechenden Formate kann der Bezug zur praktischen Korpusarbeit gewährleistet werden.
Zur Entwicklung von `Pepper` habe ich untersucht, welche Anforderungen in der linguistischen Domäne an ein solches Framework bestehen. Mit der Beschreibung dieser Anforderungen werde ich dieses Kapitel beginnen. Anschließend werde ich mich dem Prozess der Konvertierung widmen und ihn in seine Bestandteile zerlegen. Anhand der einzelnen Bestandteile werde ich beschreiben, auf welchen Ebenen die Konvertierung eines Korpus erfolgen kann. Dann werde ich die Architektur von `Pepper` vorstellen und den Ablauf einer Konvertierung an einem Beispiel erklären. Ich werde zeigen, wie sich die Performanz von `Pepper` durch die Parallelisierung von Prozessen verbessern lässt. Abschließen werde ich dieses Kapitel mit einer Bewertung der Umsetzung der formulierten Anforderungen an das Konverterframework.

5.1. Anforderungen an das Konverterframework

Die im Folgenden genannten Anforderungen an das Konverterframework dienen zum einen der Nachhaltigkeit und der Erweiterbarkeit einzelner Komponenten und zum anderen der praktischen Nutzbarkeit für die Konvertierung von Korpora.

5.1.1. Modularisierung

In Kapitel 1 habe ich anhand der Abbildung 1.1 gezeigt, wie die Konvertierung von Daten aus einem Format A in ein Format B über das gemeinsame (Meta-)Modell erfolgen soll. Die Konvertierung von A nach B kann in zwei Teilkonvertierungen unterteilt werden. Zum einen in die Konvertierung der Daten aus Format A in das gemeinsame (Meta-)-Modell `Salt` und zum anderen in die Konvertierung der Daten aus `Salt` in das Format B. Eine Verringerung der Anzahl der Konvertierungen kann nur erreicht werden, wenn die einzelnen Komponenten, die die Teilkonvertierungen ausführen unabhängig von den ihnen vor- bzw. nachgelagerten Komponenten sind. Das bedeutet beispielsweise, dass die

Komponente, die Daten von A nach `Salt` mappt, nicht erwarten darf, dass danach eine Komponente, die Daten in das Format B mappt, folgt. Genauso muss es möglich sein, die Daten in ein Format C zu mappen. Analog darf die Komponente, die Daten nach B mappt, nicht davon ausgehen, dass die Daten aus Format A stammen. Nur wenn die Komponenten unabhängig voneinander sind, kann die Verringerung der Anzahl benötigter Komponenten erreicht werden, da sie so miteinander kombiniert werden können. Diese Anforderung bezeichne ich als die Modularisierung der Komponenten.

5.1.2. Datenmanipulation

Die Komponenten, die Daten von oder nach `Salt` mappen, sollen allgemein Daten eines Formates mappen und nicht nur die Daten eines bestimmten Korpus. Auf diese Weise kann sichergestellt werden, dass eine Komponente für mehrere Korpora verwendet werden kann und die Anzahl der Konvertierungskomponenten verringert wird. Wie am Beispiel von EXMARaLDA in Abschnitt 3.1.3 zu sehen ist, kommt es in der Korpuslinguistik vor, dass bestimmte Korpora Spezifika besitzen, die nicht durch das Format ausgedrückt werden. In solchen Fällen kann es bei der Konvertierung durch eine allgemein formatbasierte Komponente zu ungewollten Ergebnissen kommen. Daher muss es Möglichkeiten geben, die Daten während der Konvertierung automatisch durch Komponenten verändern zu lassen. Diese Anforderung bezeichne ich als Datenmanipulation.

5.1.3. Erweiterbarkeit

Bereits zum jetzigen Zeitpunkt existiert eine Vielzahl an linguistischen Datenformaten. Die zukünftige Entwicklung in diesem Bereich ist nicht vorhersagbar. Aus diesen beiden Gründen ist es nicht möglich, im Rahmen dieser Arbeit eine vollständige Abdeckung aller Formate zu erreichen.
Der Wert des entwickelten Konverterframeworks für den realen Einsatz steigt mit der Menge an integrierten Mappings. Je mehr Mappings und somit auch unterstützte Formate vorhanden sind, desto stärker kann das Konverterframework in die Korpusarbeit eingebunden werden. Ich halte es daher für eine wichtige Anforderung, dass weitere Mappings in das Konverterframework integriert werden können. Diese Erweiterungsmöglichkeiten sollen aus Sicht des Modulentwicklers und aus Sicht des Anwenders möglichst einfach zu nutzen sein.

5.1.4. Nutzung unterschiedlicher Techniken

Das entwickelte Konverterframework soll in der Nutzung und bedingt auch in der Weiterentwicklung, also der Entwicklung weiterer Mappings, möglichst einfach sein. Da viele der in der Korpuslinguistik verwendeten Formate auf der Technik XML aufbauen, werden für das Mapping von Daten oft XML-Techniken wie XSLT verwendet. Aber auch die Nutzung anderer Techniken wie Perl ist für diesen Einsatz nicht unüblich. Um die Einstiegshürde für die Entwicklung weiterer Mappings für `Pepper` gering zu halten, betrachte ich die Einbettung diverser technologischer Räume in `Pepper` als eine weitere Anforderung.

5.1.5. Performanz

Eine nicht-funktionale Anforderung an das Konverterframework ist die Performanz des Konvertierungsprozesses. Da das im Rahmen dieser Arbeit entwickelte Framework nicht nur eine theoretische Untersuchung, sondern auch einen praktischen Beitrag zur Korpusarbeit darstellen soll, habe ich die Dauer der Ausführung ebenfalls in die Liste der Anforderungen aufgenommen. Zwar spielt die Betrachtung der Performanz nur eine untergeordnete Rolle, es soll jedoch zumindest untersucht werden, wie sie sich steigern lässt. Ein konkretes Ziel ist für diese Anforderung schwer zu formulieren, da die Bewertung der Performanz subjektiv ist und von Nutzer zu Nutzer unterschiedlich sein kann. Daher betrachte ich es als Ziel, eine lineare Zeitentwicklung für einfach strukturierte Korpora (flach annotierte Korpora) bezogen auf die Anzahl der Token bzw. Dokumente zu erreichen.

5.2. Beschreibung des Konvertierungsprozesses

In diesem Abschnitt werde ich schematisch beschreiben, wie ein Konvertierungsprozess von Korpora aus einem Format in ein anderes erfolgt. Dafür werde ich die Begriffe Phase, Schritt und Modul einführen, um den Prozess zu unterteilen. Anschließend werde ich zeigen, wie ein Prozess bezüglich der Ebenen, auf denen die Konvertierung vollzogen wird, durch Transformationen dargestellt und beschrieben werden kann.
In der Anforderung der Modularisierung aus Abschnitt 5.1.1 habe ich dargelegt, warum ein Konvertierung in zwei Teilprozesse (realisiert durch Komponenten) aufgeteilt werden muss. Die Prozesse werde ich im Folgenden als Phaseen bezeichnen.

Begriff 16 (Phase) *Eine Phase bezeichnet den Teilprozess einer Konvertierung, in dem gleichartige Mappings stattfinden. Als Importphase bezeichne ich den Teilprozess, in dem Mappings von einem Format A in das gemeinsame (Meta-)Modell* `Salt` *durchgeführt werden. Unter Exportphase verstehe ich den Teilprozess, in dem Mappings von Daten aus* `Salt` *in ein Format B stattfinden.*

Während einer Phase können mehrere Mappings stattfinden. Bspw. kann eine Konvertierung in zwei Formate vorgenommen werden.

Begriff 17 (Schritt) *Ein Schritt bezeichnet die Umsetzung genau eines Mappings. Ein Schritt ist immer genau einer Phase zugeordnet.*

Zur Ausführung eines Mappings ist eine Softwarekomponente erforderlich. Diese werde ich als Modul bezeichnen.

Begriff 18 (Modul) *Ein Modul ist eine Softwarekomponente, die einen Schritt realisiert. Module, die Schritte während der Importphase umsetzen, bezeichne ich als Importmodule. Analog heißen Module für Schritte während der Exportphase Exportmodule.*

Abbildung 5.1 zeigt den Zusammenhang zwischen einer Phase, einem Schritt und einem Modul innerhalb einer Konvertierung mit `Pepper` .

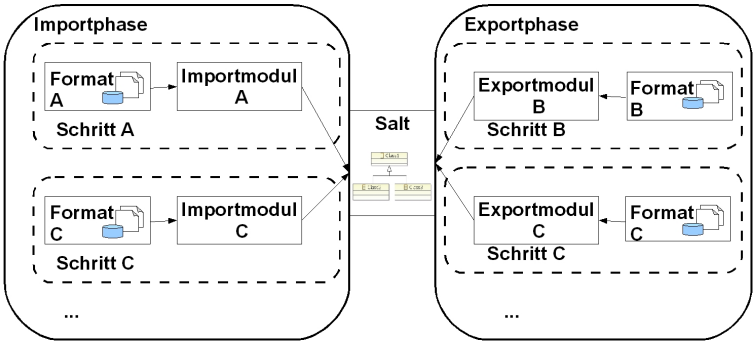


Abbildung 5.1.: Aufteilung eines Konvertierungsprozesses in Phasen und Schritte

In Abschnitt 5.1.2 habe ich erklärt, warum es wichtig sein kann, Daten abseits allgemeiner Import- und Exportmodule korpusspezifisch zu verändern. Datenmanipulationen können aber nicht nur wichtig werden, um auf bestimmte Korpora einzugehen. Oft können gleiche Manipulationen auch für unterschiedliche Korpora verwendet werden. Bspw. besitzen Daten aus dem Tiger-Format keinen Primärtext. Dieser kann zwar aus den Primärfragmenten zusammengebaut werden, die Information über die Separatorzeichen zwischen den Token ist aber nicht in dem Format enthalten. Ein Importmodul kann bspw. immer ein Leerzeichen als Separator annehmen. Das kann bei den drei Primärfragmenten 'habe' , ',' und 'gegessen' zu folgendem Primärtext führen: 'habe , gegessen'. Das überflüssige Leerzeichen zwischen 'habe' und ',' ist dann nicht ein korpusspezifisches Problem, sondern ein formatbedingtes. Um sowohl korpusspezifische als auch formatbedingte Probleme zu beheben, habe ich zwischen der Import- und der Exportphase eine weitere Phase, die Manipulationsphase, vorgesehen. In dieser können Veränderungen an den Daten eines `Salt` -Modells vorgenommen werden. Dass Veränderungen basierend auf `Salt` -Modellen vorgenommen werden, hat den Vorteil, dass das gerade gezeigte Problem der Primärtextgenerierung mit einer Softwarekomponente nicht nur für ein, sondern für unterschiedliche Formate gelöst werden kann.
Parallel zur Im- bzw. Exportphase verstehe ich unter der Manipulationsphase eine Phase, in der Daten aus einem `Salt` -Modell auf ein anderes `Salt` -Modell gemappt werden. Ein Manipulationsmodul ist ein Modul, das dieses Mapping realisiert. Abbildung 5.2 zeigt die Einordnung der Manipulationsphase in den Konvertierungsprozess.

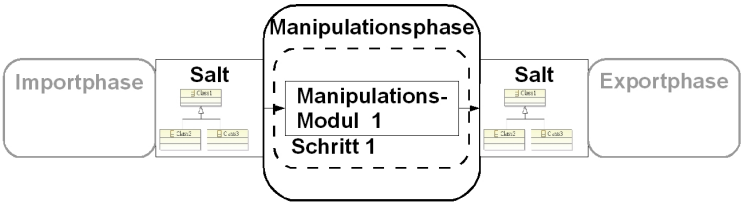


Abbildung 5.2.: Einbettung der Manipulationsphase und der Manipulationsmodule in die Konvertierung

In der Anforderung aus Abschnitt 5.1.4 habe ich die Nutzung verschiedener technologischer Räume für die Entwicklung von Mappings erwähnt. Auf `Salt` bezogene Mappings können auf zwei verschiedenen Ebenen erfolgen. Zum einen können sie auf der Forma-

tebene und zum anderen auf der Modellebene realisiert werden (wenn ein Modell für das entsprechende Format existiert). Den Zusammenhang zwischen einem Modell und Daten in einem Format habe ich in Abschnitt 2.3 beschrieben.
An dieser Stelle werde ich den Zusammenhang von Konvertierung, Mapping und Transformation formal beschreiben. Diese Beschreibung werde ich als Grundlage nutzen, um das auf die beiden Ebenen bezogende Mapping zu erklären. Dazu sei $P_A(Daten)$ die Persistenzierung von Daten in einem Format A. $M_A(Daten)$ sei die Modellrepräsentation der Daten. Bezogen auf das Metamodell `Salt` bedeutet dies, dass $M_{\texttt{Salt}}(Daten)$ Daten eines `Salt` -Modells sind, $P_{\texttt{Salt}}(Daten)$ sind Daten in der durch EMF nativ erzeugten XML-Repräsentation.

Bspw. für Tiger bedeutet dies $P_{Tiger}(Daten)$ sind Daten im Tiger-XML-Format und $M_{Tiger}(Daten)$ sind Daten in dem Modell, das durch die Tiger-API bereitgestellt wird.
Ein Mapping von Daten eines Formates A in ein Format B kann als eine Abbildung $map : P_A \rightarrow P_B$ beschrieben werden. Ein Mapping eines Importmoduls aus einem Format A auf das Metamodell `Salt` ist eine Abbildung $im : P_A \rightarrow M_{\texttt{Salt}}$, ein Mapping eines Exportmoduls, das Daten in das Format B exportiert, ist eine Abbildung $ex : M_{\texttt{Salt}} \rightarrow P_B$. Ein Manipulationsmodul kann durch die Abbildung $man : M_{\texttt{Salt}} \rightarrow M_{\texttt{Salt}}$ beschrieben werden. Ein Konvertierungsprozess lässt sich als Konkatenation einzelner Mappings wie folgt beschreiben:
Sei im ein Importprozess, ex ein Exportprozess, seien man_i mit $i \in \mathbb{N}$ Manipulationsprozesse und $P_A(Daten)$, $P_B(Daten)$ Persistenzierungen von Daten. Ein Konvertierungsprozess k ist die Abbildung $k = im \circ m_1 \circ ... \circ m_i \circ ex : P_A(Daten) \rightarrow P_B(Daten)$.
Aufgrund der beiden beteiligten Ebenen kann ein Mapping in einzelne horizontale ($\rightarrow^h$), vertikale ($\rightarrow^v$) und diagonale ($\rightarrow^d$) Transformationen aufgespalten werden. Eine diagonale Transformation ist eine Transformation von einem Modell $M_{\texttt{Salt}}$ auf eine Persistenzierung P_B oder andersherum.

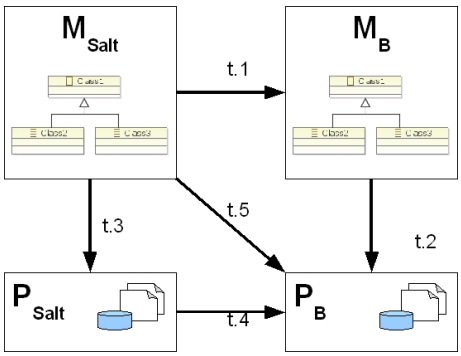


Abbildung 5.3.: Transformationen, die zu einem Exportmapping konkateniert werden können

Abbildung 5.3 zeigt die Aufspaltung eines Exportmappings. Analog dazu kann aber auch das Import- und das Manipulationsmapping in Transformationen aufgespalten werden.
Ein Importmapping kann wie folgt durch Transformationen dargestellt werden:

- $im_1 : P_A \rightarrow^v M_A \rightarrow^h M_{\texttt{Salt}}$, oder

- $im_2 : P_A \rightarrow^d M_{\texttt{Salt}}$, oder
- $im_3 : P_A \rightarrow^h P_{\texttt{Salt}} \rightarrow^v M_{\texttt{Salt}}$

Ein Exportmapping kann wie folgt durch Transformationen dargestellt werden:

- $ex_1 : M_{\texttt{Salt}} \rightarrow^h M_B \rightarrow^v P_B$ (siehe Transformationen $t.1$, $t.2$ aus Abbildung 5.3), oder
- $ex_2 : M_{\texttt{Salt}} \rightarrow^d P_B$ (siehe Transformation $t.5$ aus Abbildung 5.3), oder
- $ex_3 : M_{\texttt{Salt}} \rightarrow^v P_{\texttt{Salt}} \rightarrow^h P_B$ (siehe Transformationen $t.3$, $t.4$ aus Abbildung 5.3)

Ein Manipulationsmapping kann wie folgt durch Transformationen dargestellt werden:

- $man_1 : M_{\texttt{Salt}} \rightarrow^v P_{\texttt{Salt}} \rightarrow^h P_{\texttt{Salt}} \rightarrow^v M_{\texttt{Salt}}$, oder
- $man_2 : M_{\texttt{Salt}} \rightarrow^h M_{\texttt{Salt}}$, oder
- $man_3 : M_{\texttt{Salt}} \rightarrow^v P_{\texttt{Salt}} \rightarrow^d M_{\texttt{Salt}}$, oder
- $man_4 : M_{\texttt{Salt}} \rightarrow^d P_{\texttt{Salt}} \rightarrow^v M_{\texttt{Salt}}$

Ein Mapping kann nicht unbedingt mit jeder Technik durchgeführt werden. Beispielsweise können Transformationen mit der Technik XSLT nur innerhalb des technologischen Raums XML durchgeführt werden. Dafür kämen nur horizontale Transformationen auf Formatebene in Frage. In Abschnitt 6.5 habe ich drei verschiedene Techniken bzgl. der Performanz der Konvertierungen miteinander verglichen.

5.3. Architektur des Konverterframeworks

In diesem Abschnitt werde ich die Architektur des Konverterframeworks `Pepper` vorstellen. `Pepper` besteht aus drei Komponenten: dem gemeinsamen (Meta-)Modell `Salt` , das die Basis aller Mappings bildet, den Modulen, die die Mappings umsetzen, und dem eigentlichen Framework, das die Kontrollflusssteuerung übernimmt. `Pepper` wurde (wie auch `Salt`) modellbasiert mit EMF entwickelt und in Java implementiert. Abbildung 5.4 zeigt die aus den drei Komponenten bestehende Architektur von `Pepper` .

Für jeden Konvertierungsprozess wird ein `Salt` -Modell erzeugt, auf dem die an der Konvertierung beteiligten Module arbeiten. Die Erzeugung des Modells wird durch die Kontrollflussteuerung übernommen. Jedes Modul, das an einem Konvertierungsprozess beteiligt ist, bekommt durch die Kontrollflussteuerung eine Referenz auf das Modell und kann dieses entsprechend des Mappings, das es durchführt, manipulieren. Die Kontrollflusssteuerung übernimmt das Erzeugen, Starten und Beenden der Module. Ein Importmodul wird darüber benachrichtigt, wo sich das zu konvertierende Korpus befindet, ein Exportmodul darüber, wohin ein erstelltes Korpus zu exportieren ist. Die Kontrollflusssteuerung übernimmt die korrekte Orchestrierung der Import-, Manipulations- und Exportmodule, um den Konvertierungsprozess entsprechend eines definierten Workflows auszuführen. Wie ein solcher Workflow beschrieben wird, werde ich später in diesem Kapitel

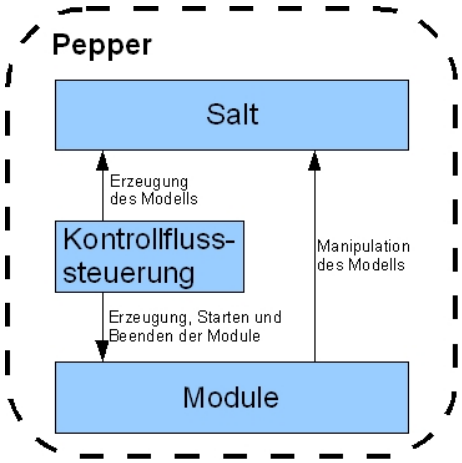


Abbildung 5.4.: Zusammenhang der drei Komponenten in `Pepper`

erläutern.

In der Anforderung aus Abschnitt 5.1.3 habe ich beschrieben, warum es wichtig ist, weitere Mappings in Form von Modulen in `Pepper` zu integrieren. Um hierfür einen relativ einfach zu benutzenden PlugIn-Mechanismus zur Verfügung zu stellen, habe ich das Framework OSGi verwendet (siehe Abschnitt 2.7). Ein Modul kann als OSGi Bundle zu `Pepper` hinzugefügt werden. `Pepper` bietet eine Ordnerstruktur an, in die Bundles einfach hineinkopiert und entfernt werden können, um sie für `Pepper` nutzbar zu machen. Diese Ordnerstruktur wird beim Starten des Frameworks nach allen enthaltenen Bundles gescannt. Im Anschluss folgt das zweischrittige Zur-Verfügung-Stellen, das ich in Abschnitt 2.7 beschrieben habe. Module können durch die Nutzung des OSGI-Frameworks als Bundles dynamisch in das Konverterframework geladen werden.
Damit ein Modul in `Pepper` integriert werden kann, muss es ein vorgegebenes Interface implementieren. Dieses werde ich an dieser Stelle kurz beschreiben. Abbildung 5.5 zeigt einen Überblick über die drei unterschiedlichen Module Importmodul, Manipulationsmodul und Exportmodul.

Jedes Modul besitzt einen Namen, über den es von der Kontrollflusssteuerung identifiziert werden kann. Zusätzlich dazu besitzen Import- und Exportmodule die Möglichkeit, über eine Formatbeschreibung („supportedFormats“) identifiziert zu werden. In einem Workflow können die Module entweder durch ihren Namen oder das Format, in dem sich das Korpus befindet bzw. in das es exportiert werden soll, angegeben werden. Jedes Modul besitzt eine Referenz auf Ressourcen, wie bspw. Property-Dateien, in denen sich Informationen zur Verarbeitung befinden. Einem Modul wird durch die Kontrollflusssteuerung über das Attribut „saltProject“ eine Referenz auf ein durch alle beteiligten Module gemeinsam genutztes `Salt` Modell zugewiesen.
Für die in Abschnitt 6.5 miteinander verglichenen Techniken QVT [OMG, 2007] und XSLT [Clark, 1999] habe ich Module entwickelt, die als eine Art Container für die jeweilige Technik dienen. Diesen Container habe ich in Java implementiert. Java dient hier als Brückentechnologie, um die Techniken anderer technologischer Räume zu integrieren. Der eigentliche Prozess der Konvertierung wird dadurch in einen anderen Raum

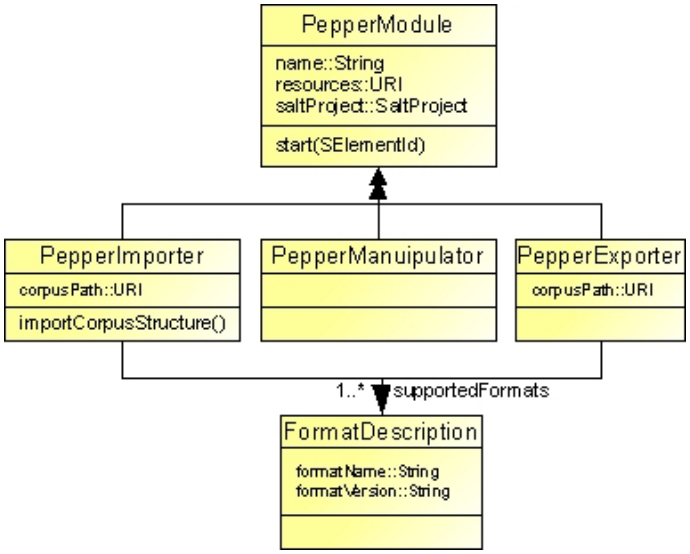


Abbildung 5.5.: Interface der in `Pepper` integrierbaren Module

verlagert. Konkret bedeutet das im Falle der Konvertierung mit XSLT, dass sich der Container darum kümmert, einen XSLT-Prozessor zu erzeugen, diesem die Daten und das Transformationsskript zu übergeben und die Transformation zu starten. Eine XSLT-Transformation kann, wie ich bereits erwähnt habe, nur als horizontale Transformation auf Formatebene erfolgen. Im Falle eines Exporters ist dies die Transformation „t.4“ aus Abbildung 5.3. Die notwendige Transformation „t.3“ wird durch den Container übernommen.
Den Ablauf einer Konvertierung durch die Angabe eines Workflows sowie die Methoden „start(SElementId)“ und „importCorpusStructure()“ werde ich im folgenden Abschnitt 5.4 beschreiben.

5.4. Konvertierung eines Beispiels

Ein Konvertierungsprozess in `Pepper` wird durch einen Workflow beschrieben. Dieser wird in Form einer XML Datei angegeben und kann durch ein EMF-PlugIn in der IDE Eclipse modelliert werden. Ein Workflow besteht aus den drei genannten Phasen und der Angabe eines Moduls für jeden Schritt, der bei einer Konvertierung ausgeführt werden soll. Die Reihenfolge der einzelnen Schritte wird durch den Workflow bestimmt. Dabei gilt, dass die Schritte der Importphase vor denen der Manipulationsphase und die Schritte der Manipulationsphase vor denen der Exportphase beginnen. Abbildung 5.6 zeigt ein Beispiel einer Workflowbeschreibung.

In dem in Abbildung 5.6 gezeigten Beispiel wird ein Korpus aus dem Tiger-Format importiert, während der Manipulationsphase verändert und in die beiden Formate relANNIS und TreeTagger exportiert. Nachdem `Pepper` mit der Angabe des Workflows gestartet wurde, werden die Bundles zu den im Workflow angegebenen Modulen gestartet. Das Importmodul, das Manipulationsmodul, sowie das zweite Exportmodul werden über den angegebenen Namen identifiziert. Das erste Exportmodul wird über die Angabe des

```
<pepperJobParams id="1">
        <importerParams       moduleName="TigerImporter"
                              sourcePath="SOURCE_PATH" specialParams="..."/>
        <manipulatorParams    moduleName="PrimaryDataCleaner"
                              specialParams="..."/>
        <exporterParams       formatName="relANNIS"
                              formatVersion="3.1"
                              destinationPath="DEST_PATH" specialParams="..."/>
        <exporterParams       moduleName="TreetaggerExporter"
                              destinationPath="DEST_PATH" specialParams="..."/>
</pepperJobParams>
```

Abbildung 5.6.: Beispiel einer Workflowbeschreibung für `Pepper`

Formates, in das exportiert werden soll, und die dazugehörige Formatversion identifiziert. Wenn für jedes angegebene Modul ein registriertes Bundle gefunden wurde, wird der Konvertierungsprozess gestartet. Dazu wird die Methode „importCorpusStructure()“ des gefundenen Importmoduls aufgerufen. Diese Methode erzeugt die Korpusstruktur, bestehend aus Korpora, deren Subkorpora und den enthaltenen Dokumenten. Anschließend ruft die Kontrollflussteuerung nacheinander für jedes beteiligte Modul die Methode „start(SElementId)“ auf, wobei der Parameter „SElementId“ ein Identifizierer auf ein Dokument im `Salt` -Modell ist. Wird die start-Methode eines Importmoduls aufgerufen, beginnt das Modul mit dem Importieren des Dokumentinhalts. Ein Manipulationsmodul beginnt mit der Veränderung und ein Exportmodul mit dem Exportieren der Daten eines Dokuments. Die Kontrollflusssteuerung sorgt dafür, dass die „start“-Methoden der Module in der im Workflow angegebenen Reihenfolge aufgerufen werden. Die Einhaltung der Reihenfolge ist wichtig, damit bspw. ein Dokument erst dann exportiert wird, nachdem alle Manipulationen darauf ausgeführt wurden.
In Abschnitt 5.1.5 habe ich geschildert, dass neben den vorgestellten funktionalen Anforderungen die Performanz für den realen Einsatz des Konverterframeworks von Bedeutung ist. Daher werde ich in Abschnitt 5.5 vorstellen, wie die Performanz durch die Parallelisierung des Konvertierungsprozesses gesteigert werden kann.

5.5. Parallelisiserung der Konvertierung

Bei der Parallelisierung des Konvertierungsprozesses muss darauf geachtet werden, dass einzelne Prozesse weiterhin in der richtigen Reihenfolge stattfinden, da das Ergebnis sonst unbrauchbar wäre. In `Pepper` habe ich drei Möglichkeiten der Parallelisierung lokalisiert. Diese sind die Parallelisierung der drei Phasen, die Parallelisierung einzelner Schritte innerhalb einer Phase und die Parallelisierung innerhalb der einzelnen Module. Jede Parallelisierung bezieht sich auf unterschiedliche Dokumente eines zu konvertierenden Korpus. Die Parallelisierung der Phasen bedeutet, dass diese bezogen auf ein Dokument zwar nicht parallel stattfinden können, aber wenn bspw. ein Dokument i importiert wurde, kann es von der nächsten Phase bearbeitet werden, während gleichzeitig der Import von Dokument $i + 1$ erfolgt. Gleiches gilt für die einzelnen Schritte. Die Parallelisierung der

Schritte führt automatisch zu einer Parallelisierung der Phasen. Eine Parallelisierung der einzelnen Module kann nicht durch das Konverterframework übernommen werden, da nicht immer gewährleistet ist, dass dadurch keine Probleme entstehen. Wenn bspw. der Inhalt von zwei Dokumenten parallel in die gleiche Datei geschrieben wird, ist das Ergebnis höchstwahrscheinlich unbrauchbar. Eine Parallelisierung der Module wird daher zwar durch `Pepper` unterstützt, aber nicht vorausgesetzt.

	Schritte\Zeitabschnitt	1	2	3	4	5
Importphase	Schritt1 (TigerImporter)	Dokument1	Dokument2	Dokument3	-	-
Manipulationsphase	Schritt2 (PrimaryDataCleaner)	-	Dokument1	Dokument2	Dokument3	-
Exportphase	Schritt3 (relANNISExporter)	-	-	Dokument1	Dokument2	Dokument3
	Schritt4 (TreeTaggerExporter)	-	-	Dokument1	Dokument2	Dokument3

Abbildung 5.7.: Beispiel für die Parallelisierung des Workflows aus Abbildung 5.6

In Abbildung 5.7 zeige ich, wie ein Dokument über die Zeitabschnitte hinweg von unterschiedlichen Modulen bearbeitet wird, während andere Module zur gleichen Zeit andere Dokumente verarbeiten. Für die Parallelisierung der Schritte gilt, dass ein Dokument zur gleichen Zeit nur genau von einem Importmodul oder genau einem Manipulationsmodul verarbeitet werden kann, wohingegen zwei Exportmodule das Dokument zur gleichen Zeit exportieren können. Dies ist möglich, da Exportmodule keine nachgelagerten Module besitzen und keine relevanten Änderungen auf dem `Salt` Modell ausführen können. Welchen Vorteil die Parallelisierung der Schritte für die Performanz bedeutet, werde ich in Abschnitt 6.4 anhand von Messungen zeigen.

5.6. Umsetzung der Anforderungen

An dieser Stelle werde ich mich erneut mit den in Abschnitt 5.1 formulierten Anforderungen beschäftigen und zeigen, inwiefern diese durch die Architektur und den Ablauf von `Pepper` erfüllt wurden.
Die einzelnen Komponenten zur Umsetzung eines Mappings werden in `Pepper` als lose Module betrachtet. Daher erlaubt die Architektur eine Unabhängigkeit der einzelnen Konvertierungsprozesse. Die Anforderung zur Modularisierung aus Abschnitt 5.1.1 wurde damit umgesetzt, auch wenn das Framework nicht sicherstellen kann, dass jedes Modul mit jedem Modul einwandfrei zusammenarbeiten kann. Die im Rahmen dieser Arbeit entwickelten Module für die Formate TreeTagger, Tiger, EXMARaLDA, PAULA und relANNIS nutzen ausschließlich die Semantik, die die Metamodelle `SaltCommonMM` und `SaltSemanticsMM` bereitstellen, und sind daher miteinander kompatibel.
Mit der Einführung der Manipulationsphase ist die Manipulation von Korpora basierend auf `Salt` möglich. Die Anforderung aus Abschnitt 5.1.2 wurde damit umgesetzt.

Durch das PlugIn-Framework OSGi [OSGiAlliance, 2009] besteht die Möglichkeit, das Framework um beliebige Module zu erweitern. Dies können Im- und Exportmodule sein, um weitere Formate in die Konvertierung zu integrieren, oder Manipulationsmodule, um eine Veränderung oder Anreicherung der Daten vorzunehmen. Die Anforderung der einfachen Erweiterbarkeit für den Nutzer aus Abschnitt 5.1.3 wurde somit umgesetzt. Durch den modularen Aufbau des Konverterframeworks ist es ebenfalls für Entwickler möglich, weitere Bundles für `Pepper` zu entwickeln, allerdings erfordert die Verwendung von OSGi hier einen höheren Einarbeitungsaufwand.
Durch die Nutzung von Java als Brückentechnologie und die Möglichkeit, Module in Form von Containern zu erzeugen, können neben Java weitere Techniken zur Konvertierung verwendet werden. Dabei übernehmen andere Techniken nicht unbedingt das gesamte Mapping, sondern bspw. nur eine horizontale Transformation auf der entsprechenden Ebene. Im Rahmen dieser Arbeit habe ich Module entwickelt, die die beiden Techniken XSLT und QVT integrieren. Auf gleiche Weise können ebenfalls weitere Techniken in `Pepper` integriert werden. Somit wurde die Anforderung aus Abschnitt 5.1.4 umgesetzt.
Auf die nicht-funktionale Anforderung einer performanten Durchführung von Konvertierungsprozessen aus Abschnitt 5.1.5 bin ich in Form von Parallelisierungen eingegangen. Einige der in dieser Arbeit verwendeten Ansätze und Techniken wirken sich negativ auf die Performanz aus. Genannt seien hier als Beispiele die Verwendeung von EMF, die sich, wie in Abschnitt 4.1 gezeigt, durch den Listenzugriff negativ auswirken kann und die Verwendung von OSGi, deren Auswirkungen auf die Performanz ich in Abschnitt 6.3 anreißen werde. Die Messergebnisse aus den beiden Abschnitten 6.4 und 6.5 zeigen, dass das Ziel der Linearität für einfach strukturierte Korpora umgesetzt wurde (zumindest bezogen auf die Technik Java). In dieser Arbeit kann die Performanz des Konverterframeworks nicht verglichen werden, da keine ähnlichen Arbeiten auf diesem Gebiet existieren. Die Performanz des Frameworks `Pepper` kann bspw. nicht mit der Konvertierung eines Annotationswerkzeuges von Fremdformaten in das jeweilige genutzte Format verglichen werden, da die Werkzeuge neben der Konvertierung noch weitere Aufgaben wie die Visualisierung wahrnehmen. Ein Vergleich würde daher nicht unter gleichen Bedingungen stattfinden können. In Abschnitt 6.4 werde ich jedoch zeigen, dass die hier vorgestellte Parallelisierung der Prozesse eine postive Auswirkung auf die Performanz hat.
In diesem Kapitel habe ich beschrieben, welche Anforderungen an ein Framework zur Konvertierung von Korpora bestehen. Ich habe einen Konvertierungsprozess in seine Bestandteile zerlegt und diese genauer beschrieben. Anschließend habe ich die Architektur von `Pepper` vorgestellt und beschrieben, welche Techniken ich zur Implementierung genutzt habe. Ich habe ein Beispiel eines Konvertierungsprozesses skizziert und daran gezeigt, wie dieser parallelisiert werden kann. Abschließend habe ich beschrieben, wie und inwiefern die Anforderungen durch `Pepper` umgesetzt werden.

6. Evaluation

In diesem Kapitel werde ich einige ausgewählte Techniken und Ansätze bewerten und darauf eingehen, inwiefern sich diese für die vorliegende Arbeit bewährt haben. Inwieweit ich die Ziele dieser Arbeit erreicht habe, beschreibe ich in Abschnitt 7.1.

6.1. Bewertung der Konvertierung über ein gemeinsames (Meta-)Modell

Der Vorteil der Konvertierung von Daten über ein gemeinsames (Meta-)Modell liegt in der Verringerung der Anzahl der notwendigen Konverter. Weiter entsteht eine höhere Nachhaltigkeit, als dies bei einem Konverter der Fall ist, der nur von Format a nach Format b konvertieren kann. Diese Nachhaltigkeit bezieht sich vor allem auf die einzelnen Module, da diese in verschiedenen Kombinationen innerhalb des Frameworks ständig wiederverwendet werden können (bspw. kann das Modul für Format a auch genutzt werden, wenn von a nach c konvertiert werden soll). Diese Modularisierung erfordert allerdings eine höhere Genauigkeit bei der Konverterentwicklung. Es muss sichergestellt sein, das ein Modul nicht von einem bestimmten vor- oder nachgelagerten Modul ausgeht. Außerdem erfordert die formatbezogene Konvertierung im Gegensatz zu der korpusbezogenen Konvertierung einen allgemeineren Umgang mit den Daten. Korpusspezifika können nicht weiter berücksichtigt werden. Die Im- und Exportmodule müssen sich streng an eine Formatbeschreibung halten und alle im Format ausdrückbaren Informationen abdecken.
Wenn ein Konverter zur Direktkonvertierung von A nach B existiert, ist die Kette der benötigten Konvertierungen kürzer, als bei der Konvertierung über ein gemeinsames (Meta-)Modell . Existiert jedoch keine Direktkonvertierung, dafür aber Zwischenschritte wie bspw. eine Konvertierung von A nach C, C nach D und D nach B, entstehen Konvertierungsketten unterschiedlicher Länge. Gleichzeitig erhöht sich die Menge potentieller Informationsverluste. Die länge der Konvertierungskette bei der Konvertierung über ein gemeinsames (Meta-)Modell hingegen liegt konstant bei zwei.
Offensichtlich ist, dass Konvertierungen über ein gemeinsames (Meta-)Modell nur dann verlustfrei sein können, wenn das (Meta-)Modell mächtig genug ist, strukturell und semantisch alle Informationen des Ursprungsformates darzustellen. `Salt` wurde basierend auf den hier herausgearbeiteten Konzepte entwickelt. Formate, die keine weiteren Konzepte beinhalten sind daher mit `Salt` darstellbar. Es ist jedoch wahrscheinlich, dass außerhalb der betrachteten Menge, Formate existieren, die Konzepte beinhalten, die nicht Teil von `Salt` sind. Wenn diese Konzepte nicht auf den abstrakten Teil von `Salt` (`SaltCoreMM`) abgebildet werden können, kommt es bei einer Konvertierung zwangsweise zu einem Informationsverlust. Daher kann `Salt` nur als gemeinsames (Meta-)Modell der betrachteten

Formate bezeichnet werden, für die Darstellung weiterer Konzepte ist eine Erweiterung von `Salt` nötig.

6.2. Bewertung des modellbasierten Ansatzes

Die modellbasierte Entwicklung und die Verwendung von EMF hat sich für die Entwicklung von `Salt` als vorteilhaft erwiesen. Wie ich in Abschnitt 2.3 erwähnt habe, bietet sie gegenüber der formatbasierten Entwicklung einige Vorteile. Diese haben u.a. dazu geführt, dass es eine klare Trennung zwischen formalem Metamodell und Persistenzierung gibt. Den Vorteil dieser Trennung habe ich in Abschnitt 2.3 erläutert. Außerdem ist parallel zur Modellierung eine *API* entstanden, die ich für die Entwicklung von Modulen für die Konvertierung nutzen konnte. Für die Entwicklung von Import- und Export-Modulen für die untersuchten Formate habe ich zum Teil ebenfalls formale Metamodelle mit EMF erzeugt. Diese habe ich in Kapitel 3.1 vorgestellt. Ich habe Mappings für Formate basierend auf einem Metamodell und einer *API* entwickelt, sowie Mappings, die direkt auf den Formaten basieren. Bewertend lässt sich sagen, dass die Entwicklung eines Mappings weit weniger zeitintensiv ist, wenn sie auf einer solchen *API* aufbaut.
Diese Vorteile wurden bei der Entwicklung des Tiger Importmodules deutlich, da ich hierfür die Tiger-API verwendet habe. Für andere Formate wie EXMARaLDA, TreeTagger etc. lag der größte Teil der Entwicklungszeit bei der Entwicklung der *API* und nicht in der Entwicklung des Mappings. Dies lag zum Teil daran, dass an einigen Stellen nur schwer ersichtlich war, welche Informationen aus den Daten nur formatbedingt sind und welche in das Metamodell übertragen werden müssen.
Nachteile des modellbasierten Ansatzes liegen in der Performanz. In Kapitel 4.1 habe ich an dem Beispiel des Listenzugriffs gezeigt, dass diese mitunter recht zeitintensiv sind. Ich habe auch gezeigt, dass sich diese Zugriffe über die Nutzung von Indizes beschleunigen lassen. Deren Modellierung ist aber recht umständlich und sollte nicht Teil des Metamodells sein. Weitere performanzbeeinträchtigende Faktoren liegen in dem hohen Overhead für abstrakte Zugriffsmethoden, die durch EMF erzeugt werden. Wie stark diese sich auf die Performanz auswirken, habe ich hier nicht genauer untersucht.
Neben dem Metamodell `Salt` habe ich auch untersucht, inwiefern `Pepper` mit den Techniken von EMF entwickelt werden kann. Dies hat sich jedoch als wenig vorteilhaft erwiesen. EMF bietet einige Unterstützung für die Entwicklung von Datenmodellen. Im Gegensatz zur UML existiert aber so gut wie gar keine Unterstützung, um Prozessmodelle durch bspw. Sequenz- oder Ablaufdiagramme zu beschreiben. Der Vorteil, den die EMF bietet, liegt dann lediglich in der Dokumentation der Architektur. Fairerweise sei an dieser Stelle erwähnt, dass es auch nicht zu den erklärten Zielen der EMF gehört, Prozesse zu modellieren.

6.3. Bewertung der Verwendung von OSGI

Die Verwendung von OSGi zur Realisierung der Erweiterbarkeit des Konverterframeworks hat sich im Bezug auf die funktionalen Anforderungen als sinnvoll erwiesen. Mit dieser

Technik ist es möglich Module in Form von Import-, Export oder Manipulations-Modulen durch das Kopieren eines Bundles in den PlugIn-Ordner von `Pepper` zu integrieren. Die in diesem Verzeichnis liegenden Module werden beim Starten von `Pepper` geladen und sind automatisch verfügbar.
Durch die Nutzung von OSGi entstehen aber auch eine Reihe von Nachteilen und Änderungen der Projektstruktur. Wie ich in 2.7 gezeigt habe, müssen Zyklen in Bundles aufgelöst werden, wodurch eine größere Projektstruktur entsteht.
Der Einsatz von OSGi hat auch Auswirkungen auf die Performanz. Für den Start der *Service Platform* und das zur Verfügung stellen ihrer Dienste wird eine gewisse Zeit benötigt. Das Laden der einzelnen Bundles benötigt ebenfalls mehr Zeit, als dies bei einer festen Integration der Fall ist. Insgesamt entsteht also ein zusätzlicher fixer Zeitbedarf durch das Starten der *Service Platform* und ein zusätzlicher variabler Zeitbedarf durch das Laden jedes einzelnen Bundles. Der zusätzliche fixe Zeitbedarf liegt bei dem von mir genutzten Rechner[1] etwa bei 5-6 Sekunden. Der zusätzliche variable Zeitbedarf hängt sehr von der Implementation des entsprechenden Bundles ab.
Außerdem erfordert OSGi einige Umstellungen bei der Programmierung, da neben anderen Umstellungen bspw. Log-Nachrichten nur über einen OSGi-LogService verwaltet werden können.

6.4. Bewertung der Parallelisierung

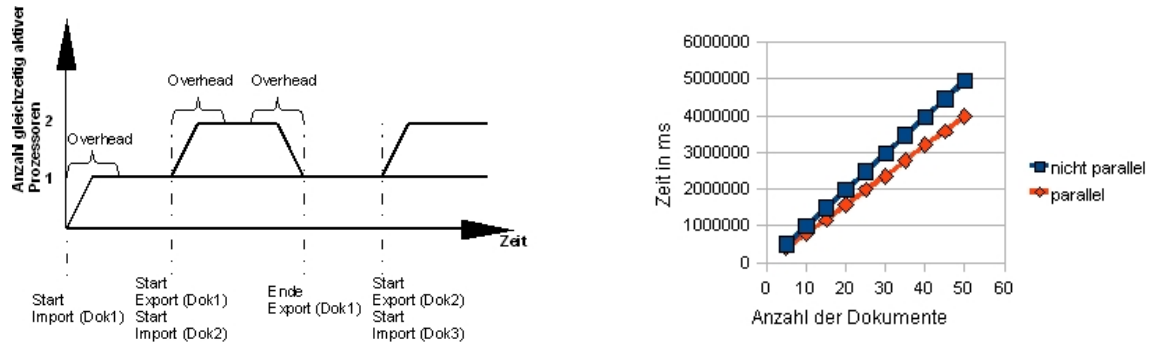
Die Parallelisierung der Phasen führt in etwa zu den erwarteten Performanzvorteilen. Für die Messungen des Zeitbedarfs parallelisierter und nicht- parallelisierter Durchläufe habe ich mehrere Quellkorpora in mehrere Zielkorpora transformiert. Die Anzahl der Dokumente pro Korpus lag dabei zwischen 5 und 50. Jedes Dokument war flach annotiert und enthielt 5000 Token. Zur Überprüfung der verlustfreien Konvertierung habe ich Daten aus dem TreeTagger-Format in das TreeTagger-Format konvertiert. Bei der sequentiellen Ausführung wurde ein Dokument zuerst importiert, dann exportiert, bevor der Import des nächsten Dokumentes gestartet wurde. Bei der parallelen Ausführung konnte der Import des zweiten Dokumentes bereits beginnen, während das erste Dokument noch exportiert wurde. Die Messergebnisse beider Messungen zeige ich in Tabelle 6.1. Abbildung 6.1 stellt diese graphisch dar.

In Abbildung 6.1 ist zu sehen, dass der Zeitverbrauch beider Methoden linear zur Anzahl der Dokumente ansteigt. Die parallele Ausführung weist einen schwächeren Anstieg auf. Tabelle 6.1 zeigt in der Spalte „prozentualer Unterschied“ den prozentualen Zeitvorteil der parallelisierten Ausführung gegenüber der nicht parallelisierten Ausführung. Dieser liegt im Schnitt bei etwa 20%. Dass der Performanzvorteil bei 20% liegt, erklärt sich dadurch, dass der Exportprozess schneller ist als der Importprozess. Das Exportmodul muss zwischenzeitlich auf das Importmodul warten. Ein Exportmodul benötigt etwa etwa 25% bis 28% der insgesamt benötigten Zeit für die Konvertierung eines Dokumentes. Die Differenz von etwa 5% bis 8% zwischen dem Performanzvorteil und der benötigten Zeit eines Exportmoduls erklärt sich z.T. durch den Zeitverbrauch, den das Laden und Beenden eines Moduls (Overhead) benötigt. Abbildung 6.1(b) zeigt den gleichzeitigen

[1] ein Laptop (Dell Inspiron 510m) mit mit einer CPU mit 1,6 GHz (Intel Pentium M) und 2 GB RAM

#Dokumente	Laufzeit in ms (nicht parallel)	(parallel)	prozentualer Unterschied
5	494175	393000	20,47
10	988350	793414,2	19,72
15	1482525	1158220,8	21,88
20	1976700	1575325,2	20,31
25	2470875	1985620,8	19,64
30	2965050	2351727,6	20,69
35	3459225	2771633,4	19,88
40	3953400	3202505,4	18,99
45	4447575	3563937,6	19,87
50	4941750	3974004,6	19,58

Tabelle 6.1.: Zeitmessungen für die parallele Verarbeitung von Import- und Export-Modul



(a) Darstellung der parallelen Prozesse

(b) Zeitverlauf der Messungen

Abbildung 6.1.: links der Ablauf der parallelen Prozesse, rechts die grafische Darstellung der Messungen aus Tabelle 6.1

Ablauf von Import- und Exportmodul.
Ich habe die Messungen auf einer 2-Prozessormaschine mit je zwei 32-bit Kernen (Intel Xeon) mit 2,0 GHz pro Kern und insgesamt 6 GB RAM durchgeführt.

6.5. Bewertung unterschiedlicher Techniken zur Konvertierung

In Kapitel 5 habe ich beschrieben, dass Techniken unterschiedlicher technologischer Räume zur Konvertierung von Modellen in `Pepper` genutzt werden können. Für diese Arbeit habe ich die Techniken XSLT [Clark, 1999] aus dem Raum XML, QVT [OMG, 2007] aus dem Raum der MDA und Java aus dem Raum Java bezogen auf die Performanz der Konvertierung miteinander verglichen. Dafür habe ich jeweils ein Exportmodul entwickelt, dass die entsprechende Technik nutzt. Für die Techniken XSLT und QVT habe ich einen Container in Java entwickelt, um diese Techniken in `Pepper` zu integrieren. Die Technik

#Token	Laufzeit in ms		
	Java	XSLT	QVT
100	50	731	11297
300	160	1552	223370
500	181	3063	1019135
600	421	4736	1762614
700	630	5378	2800958
1.000	1031	7681	-
5.000	19738	191837	-
10.000	83511	779991	-
15.000	186199	1711100	-

Tabelle 6.2.: Zeitmessungen für den Export eines `Salt` Modells in das TreeTagger-Format mit unterschiedlichen Techniken.

XSLT führt dabei die Transformation „t.4“ und die Technik QVT die Transformation „t.1“aus Abbildung 5.3 aus. Die Transformationen „t.3“ und „t.2“ werden jeweils durch den Container übernommen. Mit der Technik Java können alle beschriebenen Transformationen vorgenommen werden. Für diese Messungen habe ich das Mapping ex_1 in Java umgesetzt.
Das entwickelte Exportmodul transformiert jeweils ein Modell aus `Salt` in das Format von TreeTagger. Es werden also nur flache Annotationen in Form von Tokenannotationen betrachtet und keine rekursiven Einheiten. Für die Messungen habe ich jeweils ein Dokument mit 100 bis 15.000 Token in das TreeTagger-Format exportiert. Die gemessenen Zeiten beziehen sich ausschließlich auf die Dauer der Exportphase. Parallelisierungen habe ich dabei ausgeschlossen um die unterschiedlichen Techniken vergleichbar zu machen. Tabelle 6.2 zeigt die gemessenen Zeiten. Für die Technik QVT habe ich aufgrund des stark ansteigenden Zeitverbrauchs nur Dokumente zwischen 100 und 700 Token gemessen. Abbildung 6.2 stellt den Zeitbedarf der Techniken aus Tabelle 6.2 graphisch dar.

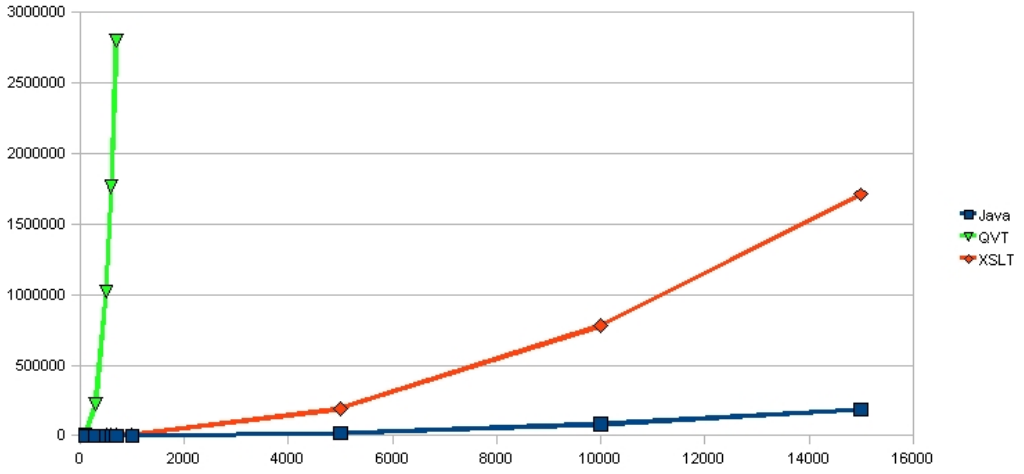


Abbildung 6.2.: Graphische Darstellung der Messungen aus Tabelle 6.2

Die gemessenen Werte für die Techniken QVT und XSLT hängen von der genutzten Brückentechnologie ab, die diese dem Raum Java zugänglich machen. Für QVT habe ich die Implementierung mediniQVT [ikv++ technologies ag, 2009] der Firma ikv verwendet. Als Implementierung eines XSLT-Prozessors habe ich den von Sun Microsystems hergestellten aus der XML-Bibliothek javax.xml [Microsystems, 2009] genutzt. Der Rechner, auf dem ich die Messungen aus Tabelle 6.2 durchgeführt habe, ist ein Laptop (Dell Inspiron 510m) mit einer CPU mit 1,6 GHz (Intel Pentium M) und 2 GB RAM.
Die Messungen zeigen, dass die verwendete Technik Java nicht nur den geringsten Zeitbedarf hat, sondern auch den geringsten Anstieg des Zeitbedarfs über die Anzahl der Token. Die Kurve der Technik Java aus Abbildung 6.2 weist einen leichten linearen Anstieg auf. An zweiter Stelle bezogen auf den Zeitbedarf und dessen Anstieg steht die Technik XSLT. Die Messungen zeigen leichten quadratischen Anstieg und einen Zeitbedarf der bei der Konvertierung von 15.000 Token etwa das 9-fache des Zeitbedarf der Technik Java beträgt. Der Zeitbedarf der Technik QVT liegt bei einer Anzahl von 100 Token schon etwa bei dem 226-fachen des Zeitbedarfs der Technik Java und steigt stark an. Zu Beginn ist der Anstieg exponentiell, schwächt sich dann aber leicht ab. Dennoch bleibt der Anstieg hoch, so dass die letzte Messung der Konvertierung mit QVT für 700 Token etwa bei dem 4000-fachen Zeitbedarf der Konvertierung mit Java liegt.
Ein grundsätzlich höherer Zeitbedarf von QVT und XSLT kann darüber erklärt werden, dass beide Techniken einen deklarativen Ansatz haben und somit nur wenig Einflussmöglichkeiten auf den Ablauf einer Konvertierung erlauben. Performanzbeeinflussungen sind dadurch kaum möglich. Beide Techniken bieten keine Möglichkeit der Definition von globalen Variablen, um bereits erreichte Zwischenergebnisse zu speichern. Diese müssen daher bei jeder Verwendung neu berechnet werden. In Abschnitt 4.1 bin ich bereits auf einen Performanznachteil, der durch das Framework EMF beim Zugriff auf Listen entsteht, eingegangen. Bezogen auf die Konvertierung mit QVT kommt dies zum tragen, da QVT ausschließlich auf den von EMF bereitgestellten Listen aufbaut.
Das Fehlen einer Zwischenspeicherungsmöglichkeit und der Performanznachteil der durch EMF entsteht erklärt den extrem hohen Zeitverbrauch von QVT jedoch nicht vollständig. Im Rahmen dieser Arbeit konnte ich nicht herausfinden, ob der hohe Zeitbedarf an der Technik QVT oder an der Implementierung durch mediniQVT liegt, da mir für den deklarativen Teil von QVT (QVT-Relations) keine weiteren Implementierungen bekannt sind.
Aus Performanzgründen bietet es sich an, Konvertierungen in einer imperativen Programmiersprache wie Java zu entwickeln. Die beiden anderen Techniken bieten mit ihrem deklarativen Ansatz zwar eine technische Abstraktion und damit eine niedrigere Einstiegshürde um eine Konvertierung zu entwickeln, haben aber negative Folgen auf die Performanz.

7. Zusammenfassung

Die vorliegende Diplomarbeit leistet einen Beitrag zur Konvertierung von unterschiedlichen Korpora der Korpuslinguistik, die in verschiedenen Formaten vorliegen. Ich habe wesentliche, häufig verwendete Formate untersucht und basierend auf den darin enthaltenen Aspekten ein gemeinsames (Meta-)Modell namens `Salt` entwickelt. Dafür habe ich untersucht, wie weit sich modellbasierte Ansätze nutzen lassen. Aufbauend auf `Salt` habe ich ein Framework namens `Pepper` zur Konvertierung von Daten der einzelnen Formate entwickelt.

In Kapitel 3 habe ich fünf wesentliche Formate vorgestellt und analysiert, zu welchem Zweck sie jeweils entwickelt wurden und welche linguistischen Phänomene mit den einzelnen Formaten darstellbar sind. Ich habe Konzepte erarbeitet, die die Eigenschaften aller untersuchten Formate abdecken.
Auf der Grundlage der Konzepte habe ich das gemeinsame (Meta-)Modell `Salt` erarbeitet, welches in Kapitel 4 vorgestellt wird. Dabei wurde gezeigt, dass Salt auf einem allgemeinen Graphen basiert und somit von Graph-Algorithmen bearbeitet werden kann. Außerdem habe ich einen Vorschlag vorgestellt, wie mit der Bedeutung von strukturellen Einheiten und Annotationen in einem Korpus umgegangen werden kann.
In Kapitel 5 wurde das Konverterframework Pepper zur Konvertierung linguistischer Daten vorgestellt. Ich habe gezeigt, welche Anforderungen an ein solches Framework bestehen und wie diese in Pepper umgesetzt werden.
Abschließend habe ich in Kapitel 6 einige ausgewählte Techniken und Ansätze bewertet, die im Rahmen dieser Arbeit zum Einsatz kamen. In diesem Kapitel habe ich deutlich gemacht, welche sich bewährt haben und welche nicht.

7.1. Fazit

In der vorliegenden Arbeit habe ich die Frage beantwortet, wie die Konvertierungen linguistischer Daten aus unterschiedlichen Formaten nachhaltig und korpusübergreifend entwickelt werden können.
Ich habe gezeigt, dass unterschiedliche linguistische Analysen auf gemeinsamen Grundbausteinen aufbauen. Hierfür habe ich unterschiedliche relevante Formate untersucht und deren Aspekte beschrieben. Diese unterschiedlichen Aspekte habe ich zu gemeinsamen Konzepten zusammengeführt, anhand derer sich die unterschiedlichen Formate beschreiben und vergleichen lassen. Ich habe gezeigt, wie auf der Grundlage dieser Konzepte ein gemeinsames (Meta-)Modell entwickelt werden kann, das in der Lage ist, die linguistischen Analysen, für die die Formate entwickelt wurden, darzustellen. Auf der Basis dieses gemeinsamen (Meta-)Modells ist es möglich, unterschiedlichste Korpora in unterschiedliche Formate zu überführen.

Das gemeinsame (Meta-)Modell deckt nur die Konzepte explizit ab, die ich hier identifiziert habe. Weitere Formate, die ich im Rahmen dieser Arbeit nicht untersucht habe, können Aspekte beinhalten, die nicht durch die Konzepte abgedeckt werden. Für diese Fälle ist zu untersuchen, inwiefern das gemeinsame (Meta-)Modell erweitert werden muss.
Durch die Verwendung des gemeinsamen (Meta-)Modells bei der Konvertierung habe ich gezeigt, dass der theoretische Ansatz zur Verringerung der Anzahl an Mappings praktisch umsetzbar ist. Schlussendlich habe ich gezeigt, wie ein Framework zur Konvertierung linguistischer Daten auf der Grundlage des gemeinsamen (Meta-)Modells entwickelt werden kann. Die Entwicklung der Mappings fand nicht bezogen auf bestimmte Korpora, sondern entsprechend der Konzepte statt, die ein Format bietet. Durch die Kombinierbarkeit der zur Konvertierung erzeugten Module können diese nachhaltig entwickelt und wiederverwendet werden.

7.2. Ausblick

Die untersuchte Menge an Formaten deckt nur einen kleinen Ausschnitt der existierenden Formate ab. Daher wäre für viele weitere Formate wie bspw. mmax (siehe
[Müller and Strube, 2006]) oder urml (siehe [Reitter and Stede, 2003]). zu prüfen ob diese auf das hier entwickelte gemeinsame (Meta-)Modell abbildbar sind. Damit verbunden ist die Entwicklung weiterer Import- und Export-Module, die in das Framework integriert werden können.
Es kann untersucht werden, welche Manipulationen zwischen der Import- und Exportphase sich verallgemeinern lassen und inwiefern dafür allgemeine Manipulationsmodule nutzbar sind. Beispielsweise könnte eine interessante Anwendung das Verschmelzen von Daten aus unterschiedlichen Ursprungsformaten sein. Angenommen unterschiedliche Annotationen zu demselben Korpus liegen sowohl im EXMARaLDA-Format als auch im Tiger-Format vor. Nun sollen beide Annotationsarten miteinander verschmolzen werden, so könnte dies durch ein Manipulationsmodul geschehen und in den Konvertierungsprozess integriert werden. Eine ähnliche Problemstellung wäre die Anreicherung eines Korpus mit seinem Primärtext. In Kapitel 3.1 habe ich beschreiben, dass bspw. das Tiger-Format nicht primärtexteerhaltend ist. Wenn der Primärtext bspw. noch als einfache Textdatei existiert, könnte eine Anwendung sein, diesen während einer Konvertierung dem Korpus hinzuzufügen.
Hier wäre zu prüfen, auf welchen Ebenen eine solche Datenanreicherung stattfinden kann und wie die entsprechenden Stellen (bspw. im Primärtext) identifiziert werden können. Außerdem könnte eine konkrete Umsetzung in Form eines Manipulations-Moduls erfolgen.
Das entwickelte Konverterframework kann bisher nur als Konsolenapplikation genutzt werden. Um es in der linguistischen Domäne weiter integrieren zu können, wäre es nützlich, eine grafische Oberfläche zu entwickeln. Diese könnte der bisherigen Applikation vorgelagert werden und mit dieser über die in Kapitel 5 beschriebene Workflow-Beschreibung kommunizieren.
Die Performanz der Konvertierung kann an verschiedenen Stellen des Systems noch gestei-

gert werden. Eine Möglichkeit könnte darin liegen, Konvertierungen nicht nur auf einem Rechner, sondern auf einem Cluster von Rechnern auszuführen. Hierfür kann untersucht werden, inwiefern sich das System Hadoop (siehe [White, 2009]) der Apache Foundation für diese Aufgabe eignet.
`Salt` nutzt zur Persistenzierung bisher nur das durch *EMF* nativ erstellte *XML*-Format. An dieser Stelle kann geprüft werden, inwiefern dieses durch die Ressource-Technik von *EMF* durch ein standardisiertes Format wie GrAF (siehe [Ide et al., 2007]) ersetzt werden kann.
Sobald das System ISOCat ([ISO-TC-37, 2009]) genügend Akzeptanz durch die linguistische Domäne besitzt und mit hinreichend Datenpunkten gefüllt ist, wäre zu untersuchen, inwiefern das jetzige gemeinsame (Meta-)Modell diese Technik integrieren kann bzw. ob dafür eine Erweiterung vorgenommen werden muss.
EMF bietet nativ Möglichkeiten an, Editier-PlugIns für Eclipse, basierend auf einem Metamodell, zu erzeugen. Diese könnten an linguistische Anforderungen angepasst und erweitert werden. So könnte auf *EMF*-, `Salt` - und Eclipse-Basis ein Editierwerkzeug entwickelt werden. Hierfür kann untersucht werden, inwiefern sich das Graphical Modeling Framework (GMF, siehe [Eclipse Foundation, 2009b]), das ebenfalls von der Eclipse Foundation entwickelt wurde, für die unmittelbare Annotation von Mehrebenenkorpora nutzen lässt. Die Annotation konnte bisher nur durch die Arbeit mit mehreren getrennten Annotationswerkzeugen erfolgen.
Eine weitere Nutzungsmöglichkeit für das Konverterframework `Pepper` kann die Integration in bereits bestehende Werkzeuge zur Korpusanalyse sein. Bspw. könnte `Pepper` in das Anfrage- und Suchsystem ANNIS (siehe [Zeldes et al., 2009]) integriert werden. Dadurch könnten gefundene Treffer in die durch `Pepper` unterstützen Formate exportiert werden.
In *EMF* gibt es Ansätze, aus einem Modell eine spezifische modellbezogene Anfragesprache zu generieren (siehe [Eclipse Foundation, 2009c]). Hier kann untersucht werden, inwiefern eine modellbezogene Anfragesprache im Bereich der Korpuslinguistik angewandt werden kann. Diese Sprache kann durch die Nähe des gemeinsamen (Meta-)Modells zur linguistischen Domäne an diese angepasst werden und so einen intuitiveren Umgang als allgemeine Sprachen wie *XSLT*, *XQuery* oder *QVT* bieten. Diese Sprache kann ggf. in ein Modul eingebunden und für die Erzeugung von Mappings genutzt werden.

A. Appendix

A.1. MDA

MDA ist die Kurzform für **M**odel **D**riven **A**rchitecture oder zu Deutsch modellgetriebene Architektur. Die MDA ist eine von der OMG weitergeführte Form der „modellgetriebenen Softwareentwicklung“, zu der auch das „Software Engineering“ gehört. Die MDA ist eine Art Sammelbegriff für verschiedene Techniken aus der modellbasierten Softwarentwicklung, die diesen Prozess modellzentriert gestalten. Unter anderem gehören hierzu:

- die „Unified Modeling Language“, kurz *UML* (siehe [OMG, 2009b], als eine Möglichkeit der Beschreibung von Software,
- die „Object Constraint Language“, kurz *OCL* (siehe [OMG, 2006]), die eine Beschreibungssprache zur Verfügung stellt, um ein Modell mit Bedingungen anzureichern
- die „Query View Transformation“, kurz QVT (siehe [OMG, 2007], als eine Transformationssprache zwischen Modellen. Auf die QVT werde ich in Abschnitt 2.5 näher eingehen.

Die MDA fasst den Prozess der Softwaremodellierung mit Hilfe von Modellen in vier Stufen zusammen, die einen unterschiedlichen Abstraktionsgrad aufweisen. Diese Stufen sind:

1. CIM (Computational Independent Model)
   Das CIM ist die technisch abstrakteste Ebene, hier geht es nur um das Anwendungsgebiet und nicht um Techniken. In dieser Ebene bewegen sich die Spezialisten der Anwendungsdomäne, hier die Spezialisten der Linguistik.
2. PIM (Platform Independent Model)
   Das Platform Independent Model beschreibt die Schnittstelle zwischen Anwendungsgebiet und technischem Gebiet. Hier bewegen sich Spezialisten beider Domänen, hier können linguistische Metamodelle wie `Salt` angesiedelt werden. Die Spezifikationen dieser Ebene sind komplett technikunabhängig.
3. PSM (Platform Specific Model)
   Auf dieser Ebene liegt ein schon technikbezogenes Modell vor. Hier bewegen sich hauptsächlich die Spezialisten der technischen Domäne. Das Modell dieser Ebene ist nicht mehr so abstrakt wie das des PIM und wird schon an bestimmte technische Umsetzungen gebunden.

4. IM (Implementation Model)
   Dies ist die technischste Ebene dieser Betrachtung, die hier vorliegenden Modelle sind an die genutzten Techniken angepasst und in deren Sprachumfang formuliert. Hier können die Formatbeschreibungen oder auch die *APIs* der Metamodelle angesiedelt werden.

Die Überführungen eines abstrakteren Modells in ein weniger abstraktes wird ebenfalls als eine Transformation bezeichnet. Eine ausführlichere Beschreibung ist unter [Nolte, 2009b] S. 8ff oder [Miller and Mukerji, 2003] zu finden.

A.2. EMF

In diesem Abschnitt werde ich den Prozess der Modellierung und Entwicklung mit EMF beschreiben. In Abschnitt 2.4 habe ich bereits erwähnt, dass es zur Philosophie der EMF gehört, dass sich Programmierung und Modellierung immer wieder abwechseln können. Der Prozess beginnt mit der Metamodellerzeugung, hierfür können die durch EMF definierten Metamodellelemente Paket (EPackage), Klasse (EClass), Datentyp (EDataType), Operation (EOperation), Attribut (EAttribute), Referenz (EReference), Vererbung (EInheritance), Aufzählung (EEnum) verwendet werden. Eine Beschreibung der einzelnen Metamodellelemente ist [Steinberg et al., 2009] zu entnehmen. Ein Metamodell kann neu über ein grafisches Modellierungswerkzeug (wie GMF, das Graphical Modeling Framework von EMF) erstellt werden. Es kann aber auch aus bereits bestehendem Java-Quellcode oder aus XML-Schema-Definitionen (wie XSD) generiert werden. EMF bietet eine breite Palette vorgegebener Eigenschaften, die die einzelnen Metamodellelemente beschreiben, um auf das Verhalten der Metamodellelemente Einfluss zu nehmen. Beispielsweise kann in einem Metamodell angegeben werden, ob ein Metamodellelement zu persistenzieren ist, ob es von einem bereits existierenden Modellelement abgeleitet ist und ob es automatisch mit Inhalt gefüllt werden soll oder nicht. Ein in EMF modelliertes Metamodell kann zusätzlich um *OCL*-Bedingungen (Object Constraint Language, siehe [OMG, 2006]) angereichert werden, mit denen die Integrität eines Modells näher beschrieben werden kann. Über ein Validierungs-Framework existiert so die Möglichkeit, das Modell zu validieren und zu prüfen ob die enthaltenen Daten den Bedingungen des Metamodells entsprechen.

Der nächste Schritt ist die Codeerzeugung. Wie ich bereits erwähnt habe, bietet EMF die Möglichkeit aus dem Metamodell nativ Java-Code oder bspw. über eine eigene M2T Code in anderen Zielsprachen zu erzeugen. Der erzeugte Code besteht hauptsächlich aus

- Zugriffsoperationen für die Daten des Modells in Form von *Get- und Set-Methoden*,
- abstrakte Zugriffsoperationen die über die Java-Reflection Technik funktionieren, um anderen modellbasierten Werkzeugen Zugriffsmöglichkeiten zu bieten,
- Benachrichtigungsmethoden, sog. Notifier, mit denen es auf einfache Weise möglich ist, bestimmte Modellelemente zu benachrichtigen, sobald sich der Wert eines anderen Modellelementes geändert hat, und

- einer XML-Serialisierung, mit der ein erzeugtes Modell automatisch in der Technik *XML* persistenziert werden kann.

Damit kann aus einem Metamodell automatisch eine *API* erzeugt werden. Operationen zur Geschäftslogik können angegeben werden, aber die auszuführende Geschäftslogik kann nicht automatisch generiert werden.

Nach der Erzeugung kann der Code verändert und manipuliert werden. Nachdem man sich in der etwas komplexen Projektstruktur des erzeugten Codes zurechtgefunden hat, können Operationen überschrieben und implementiert werden. Hierfür werden sog. *protected regions* erzeugt. Eine *protected region* bezieht sich auf einen Codeblock, bspw. den Inhalt einer Operation, und sorgt dafür, dass bei erneuter Codeerzeugung der Block der *protected region* nicht überschrieben wird. Dieser Mechanismus ist die Grundlage des Zusammenspiels von Modellierung und Programmierung als sich abwechselnde Prozesse. Das Metamodell kann sich ändern und somit auch der erzeugte Code, eine *protected region* bleibt jedoch unberührt. Auf diese Weise kann die Geschäftslogik in den erzeugten Code implantiert werden, auch wenn das Modell keine Beschreibungsmöglichkeiten hierfür erlaubt.
Neben dem eigentlichen Programmcode können zusätzliche u.U. nützliche Klassen und Features erzeugt werden. Dazu gehören JUnit-Testklassen und Testumgebungen, aber auch ein Editierframework, das als Eclipse PlugIn genutzt werden kann und eine simple grafische Editoroberfläche für das erzeugte Metamodell bietet. Der Editor kann erweitert und an das Metamodell angepasst werden, um spezifische Visualisierungen anzubieten.

A.3. QVT

QVT ist eine Modelltransformationssprache und steht für **Q**uery **V**iew **T**ransformation. Mit QVT ist es möglich, horizontale Modelltransformationen vorzunehmen. Dabei kann das zu transformierende Modell (Quellmodell) von dem gleichen Metamodell abgeleitet sein wie das Modell, in das transformiert wird (Zielmodell). Es kann sich aber auch um zwei Modelle mit unterschiedlichen Modellpfaden handeln. „Query“ steht für die Extrahierung der zu transformierenden Daten des Quellmodells, „View“ für die Anordnung der Daten in dem Zielmodell und „Transformation“ für den eigentlichen Prozess der Transformation. QVT ist eine abstrakte Sprache, da sie auf der Metamodellebene angesiedelt und somit unabhängig von einer Programmiersprache ist.
QVT ist eine relativ junge Entwicklung und hat gute Chancen zu einem Standard auf diesem Gebiet zu werden, da QVT ein Vorschlag der OMG ist und gut in bereits bestehende Standards wie UML (siehe [OMG, 2009b]) oder OCL (siehe [OMG, 2006]) integriert wurde.
Ähnlich wie der Zusammenhang zwischen XQuery und XPath ist auch der zwischen QVT und QCL. OCL wird als Basissprache für QVT genutzt, Modellzugriffe und zu prüfende Bedingungen werden daher in OCL formuliert. Die Sprache QVT ist in vier Bausteine aufgeteilt und besteht im Grunde aus zwei Sprachen (einer deklarativen und einer imperativen) mit einer unterschiedlichen Herangehensweise.

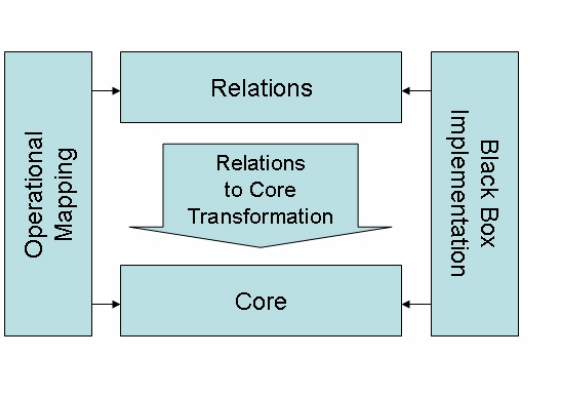


Abbildung A.1.: Spracharchitektur von QVT. Quelle: www.wiki.org (letzter Zugriff: 09.08.2009)

Abbildung A.1 zeigt ein Modell der Sprache, aufgeteilt in die vier Bausteine: „Operational Mapping“, „Relations“, „Core“ und „Black Box Implementation“.

Begriff 19 (Core) *Core stellt den Sprachkern von QVT dar. Hier befinden sich alle Grundkonzepte der Sprache. Aus diesen Grundkonzepten müssen alle anderen Sprachkonzepte abgeleitet sein bzw. müssen sie auf die Konzepte in Core voll abgebildet werden können.*

Begriff 20 (Relations) *Relations ist der deklarative Teil von QVT. Die hier angesiedelten Sprachkonstrukte treffen keinerlei Aussage über die Verarbeitung, sondern nur über ihre Funktion (ähnlich der SQL).*

Begriff 21 (Operational Mapping) *Dies ist der imperative Sprachteil der QVT. Die hier angesiedelten Sprachkonstrukte sind nicht rein deklarativ, sie sind einer Programmiersprache wesentlich ähnlicher als die Konzepte von Relations. Ihre Ablaufreihenfolge bzw. die Auswirkungen auf das Laufzeitsystem kann besser bestimmt werden.*

Begriff 22 (Black Box Implementation) *Mit den Black-Box-Funktionen erlaubt es die QVT, Sprachkonstrukte außerhalb der QVT anzusiedeln. Bspw. können in herkömmlichen Programmiersprachen (wie Java) Konstrukte formuliert und anschließend in die QVT integriert werden. Die Bedingung dabei ist, dass die Funktionen, die innerhalb der Black Box angeboten werden, auf die Core-Sprache abgebildet werden können.*

Im Rahmen dieser Arbeit werde ich mich mit dem deklarativen Ansatz der QVT (QVT-Relations (siehe [Nolte, 2009b])) beschäftigen und werde diesen hier detailierter beschreiben. Weiterführende Literatur zu dem imperativen Ansatz von QVT kann unter [OMG, 2007] oder unter [Nolte, 2009a] gefunden werden.
Um eine Transformation zwischen einem Quell- und einem Zielmodell zu beschreiben, wird eine Relation zwischen Query und der View erzeugt. Diese Relation ist eine Abbildung von Datenpunkten des Quellmodells auf Datenpunkte des (noch nicht vorhandenen) Zielmodells. Dabei bedient man sich einer Hilfsvariablen und bindet sie an das Element

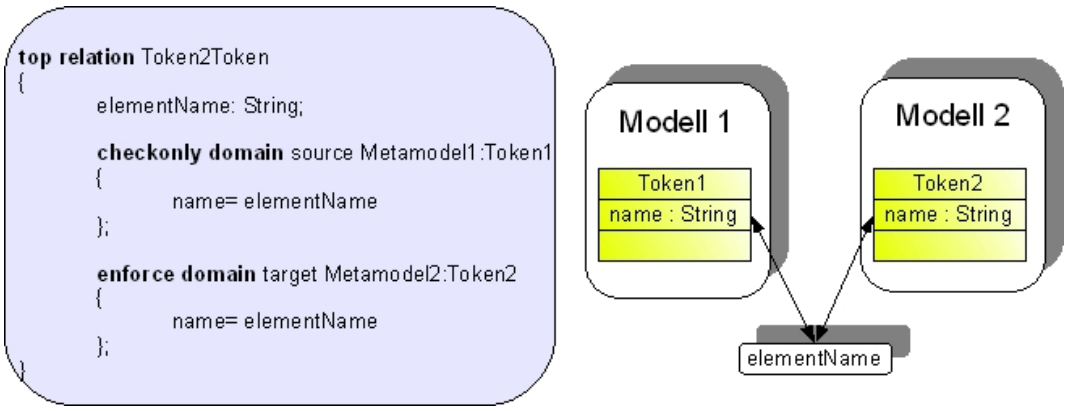


Abbildung A.2.: Abbildung eines Quellelements auf ein Zielelement über eine Hilfsvariable

aus dem Quellmodell und das Element aus dem Zielmodell. Die folgende Abbildung A.2 zeigt ein Codefragment aus QVT-Relations und eine Darstellung der Bindung der Hilfsvariablen.

Dieses Beispiel beschreibt die Transformation eines Modellelements („Token1“) aus dem Metamodell „Metamodel1“ in ein Modellelement („Token2“) aus dem Metamodell „Metamodel2“. Für die Transformation des Attributs „Token1.name“ in das Attribut „Token2.name“ wird die Hilfsvariable „elementName“ erzeugt, die an beide Attribute gebunden wird. Erst über diese Dreierbindung kann eine Transformation erfolgen.
QVT-Relations hat den Anspruch, eine bidirektionale Transformation vornehmen zu können. Das heißt, mit einer Relation soll von Quellmodell in Zielmodell und andersherum transformiert werden können. Die in Abbildung A.2 gezeigte Relation wird jedoch unidirektional ausgeführt und führt von „Metamodel1“ nach „Metamodel2“. Die Richtungsangabe wird durch die Attribute „checkonly“ und „enforce“ bestimmt. Dabei kann der „checkonly“ Block als Bedingung betrachtet werden. In diesem Fall bedeutet das, nur wenn es ein Attribut „Token1.name“ in dem Quellmodell gibt, wird der „enforce“ Block ausgeführt. Für eine bidirektionale Abbildung müssen also zwei „enforce“-Blöcke erzeugt werden. Die Angabe eines oder mehrerer „checkonly“-Blöcke ist optional.
Die Abarbeitungsreihenfolge der Ausführung einer Transformation kann aufgrund des deklarativen Ansatzes nicht beeinflusst werden. QVT-Relations bietet jedoch die Möglichkeit, mittels der Schlüsselwörter „where“ und „when“ die Ausführung von Relationen von der Ausführung anderer Relationen abhängig zu machen. Die tatsächliche Reihenfolge der Abarbeitung wird aber der jeweiligen QVT-Relations-Implementierung überlassen. Diese deklarative Beschreibung kann zugleich als Vor- und als Nachteil gesehen werden. Zum einen erfordert es ein Umdenken, wenn man imperative Abläufe gewohnt ist, zum anderen ist es so kaum möglich, die Abarbeitungszeit zu beeinflussen. Ein weiterer Vorteil der QVT-Relations ist der geringe syntaktische Sprachumfang und, interessant für diese Arbeit, ist die Nähe an dem zu transformierenden Metamodell. Transformationen werden, wie Abbildung A.2 gezeigt hat, auf Basis der in den Metamodellen definierten Elemente getroffen. Dieser Aspekt ist hier interessant, da es so evtl. möglich wäre, mit wenigen technischen Mitteln Transformationen zwischen linguistischen Modellen vorzunehmen.

A.4. OSGi und Equinox

In Abschnitt 2.7 habe ich bereits erwähnt, wofür OSGi steht und was sich dahinter verbirgt. Ich habe ebenfalls erwähnt, dass ein Softwareprodukt in Komponenten unterteilt wird. Diese Komponenten werden in OSGi als Bundles bezeichnet. An dieser Stelle beginne ich direkt mit der Beschreibung eines Bundles.
Ein Bundle ist eine Sammlung von fachlich oder technisch zusammengehörenden Einheiten. Technisch besteht ein Bundle aus einer Bundle-Beschreibung (einem Manifest), Java-Klassen, Java-Packages, die diese Klassen beinhalten, und einer festgelegten Schnittstelle nach außen. Damit ein Bundle mit anderen Bundles interagieren kann, wird die Schnittstelle in der Bundle-Beschreibung festgelegt. Es wird angegeben, welche Abhängigkeiten ein Bundle zu anderen Bundles bzw. Packages aus diesen Bundles besitzt und welche Funktionalitäten in Form von Packages anderen Bundles zur Verfügung gestellt werden. Diese Funktionalitäten werden in OSGi Services (zu Deutsch Dienste) genannt.
Bundles spielen die zentrale Rolle im OSGi-Framework, daher wird in OSGi auch alles bis auf die OSGi *Service Platform* als ein Bundle verstanden. Die OSGi-Spezifikation stellt eine Menge von Services zur Verfügung, die von einem Bundle genutzt werden können. Jeder dieser Services ist selbst wiederum ein Bundle. Das Starten von Bundles erfolgt in zwei Schritten. Nach dem Start der Laufzeitumgebung muss ein Bundle 1) installiert und 2) gestartet werden:

1. Installieren
   Beim Installieren wird ein Bundle an der OSGi *Service Registry*, einem Dienst innerhalb der *Service Platform* angemeldet und bekannt gemacht. Die *Service Registry* registriert außerdem alle Abhängigkeiten, die das Bundle benötigt, um gestartet werden zu können, und alle durch dieses Bundle angebotenen Dienste.

2. Starten
   Wird der Befehl zum Starten des Bundles gegeben, versucht die *Service Platform* alle Abhängigkeiten des Bundles aufzulösen. Dafür wird mit Hilfe der *Service Registry* jeder Service, der eine Abhängigkeit darstellt, gesucht. Ist ein Bundle installiert, das einen entsprechenden Service anbietet, wird es ebenfalls gestartet. Das Starten des Bundles, das die Abhängigkeit darstellt, wird vor dem eigentlichen Bundle gestartet und erst wenn alle Bundles, die Abhängigkeiten darstellen, gestartet wurden, wird das eigentliche gestartet. Dieser Vorgang wird rekursiv ausgeführt, bis alle benötigten Bundles gestartet wurden.

Um die Startreihenfolge der Bundles brauchen sich Anwender sowie Entwickler keine Gedanken zu machen, das übernimmt die *Service Platform*. Bei der Entwicklung von Bundles muss allerdings darauf geachtet werden, dass keine zyklischen Abhängigheiten entstehen. Zyklen müssen aufgelöst werden, indem ein Bundle aufgebrochen und auf mehrere Bundles aufgeteilt wird. Abbildung A.3(a) zeigt das Zyklenproblem und Abbildung A.3(b) zeigt das Aufbrechen des Bundles sowie die Verteilung der Dienst auf mehrere Bundles.

Die Auflösung solcher Zyklen führt jedoch zu einer aufgeblähten Projektstruktur, da für jede zyklische Abhängigkeit ein weiteres Bundle geschaffen werden muss, auch wenn

(a) Bundles mit Zyklus (b) Bundles ohne Zyklus

Abbildung A.3.: Abbildung a zeigt eine Abhängigkeit der Dienste 1, 2, 3 und 4. Durch die Aufteilung der Dienste auf zwei Bundles entsteht eine nicht zulässige zyklische Abhängigkeit der beiden Bundles. Abbildung b löst das Problem aus Abbildung a, indem ein drittes Bundle hinzugefügt wird.

die Dienste keinen Zyklus bilden.
Ein weiterer interessanter Aspekt, den OSGi eingeführt hat, sind sog. Extension Points. Ein Extension Point (zu Deutsch Erweiterungspunkt) ist eine Möglichkeit, innerhalb eines Bundles Punkte zu definieren, an denen dieses Bundle erweitert werden kann. Angenommen, Bundle B möchte Bundle A um einen bestimmten Dienst erweitern, so muss Bundle A einen Extension Point und eine Beschreibung der Erweiterungsmöglichkeit anbieten. Bundle B kann nun diese Schnittstelle implementieren und sich bei Bundle A als Erweiterung anmelden. Wird nun der Dienst in A aufgerufen, der den Extension Point definiert, so ruft Bundle A automatisch den Dienst von Bundle B auf. Technisch gesprochen, kann ein solcher Extension-Point-Mechanismus als umgedrehte Vererbung betrachtet werden. Umgedreht, weil er nicht von dem erbenden Objekt bzw. erweiternden Bundle ausgeht, sondern von dem Ursprungsobjekt/ Bundle.

Der Mechanismus der Extension Points bietet ebenfalls eine gute Möglichkeit, bestehende Software erweiterbar zu halten. So können nicht nur neue Funktionalitäten in Software eingebaut werden, sondern es können auch geringfügigere Änderungen oder Erweiterungen integriert werden.
Durch die beschriebenen Eigenschaften bietet sich OSGi als ideale Plattform an, einen PlugIn-Mechanismus in Software zu integrieren, um sie so bis in den Kern erweiterbar zu halten. Zukünftige Funktionen können auf relativ einfache Weise in bestehende Software integriert werden.
OSGi stellt jedoch nur eine Spezifikation der Möglichkeiten dar, OSGi beinhaltet keine Implementierung. Daher möchte ich an dieser Stelle Equinox als die von mir verwendete Implementierung vorstellen.
Equinox (abgeleitet von dem lateinischen Wort für Tagundnachtgleiche) ist die OSGi-Implementierung der Eclipse Foundation. Equinox wird seit 2003 entwickelt und war ursprünglich dafür gedacht, die Eclipse (zu Deutsch Finsternis) IDE auf ein PlugIn-Framework umzustellen. Seit der Version „Eclipse Callisto“ basiert Eclipse vollständig auf Equinox. Jede Funktionalität wird als Bundle zur Verfügung gestellt, was eine einfache Erweiterungsmöglichkeit für den Nutzer zur Folge hat. PlugIns müssen einfach nur in ein entsprechendes PlugIn-Verzeichnis kopiert werden und werden dann automatisch beim Starten von Eclipse berücksichtigt. Die Möglichkeit, bspw. neue Menüpunkte in

die IDE zu integrieren, wird über die Extension-Point-Technik realisiert. Später wurde Equinox aus der IDE herausgelöst, um auch außerhalb („out-of-the-box“) für andere Softwareprojekte zur Verfügung zu stehen. Dieser Punkt ist für diese Arbeit sehr wichtig, da die entwickelte Software auch außerhalb einer IDE einsetzbar sein soll.
Weitere Implementierungen sind neben anderen: Felix von Apache (siehe [Apache Foundation, 2009]), Knopflerfish (siehe [The Knopflerfish Project, 2009]) und Spring-OSGI der SpringCommunity (siehe [spring source community, 2009]). Ich habe Equinox gewählt, da diese Implementierung schon in einem sehr fortgeschrittenen Stadium ist. Außerdem ist die nutzende Community durch den Hauptnutzer Eclipse sehr aktiv und es existieren umfangreiche Dokumentation. Ein weiterer Vorteil während der Entwicklungsphase war die native Unterstützung von Equinox durch die Eclipse IDE. Equinox bietet einige Erweiterungen von OSGi an, die aber nicht Teil des Standards sind. Wenn das Projekt auch unter anderen Implementierungen genutzt werden soll, ist es an dieser Stelle wichtig, darauf zu achten, nur die spezifizierten Techniken zu nutzen. Dafür bietet Eclipse die Möglichkeit an, zwischen Equinox-Verträglichkeit und OSGi-Verträglichkeit zu wählen.

A.5. Glossar

Begriff	Beschreibung
Abdeckungsvererbung	siehe Abschnitt 2.1.1
anaphorische Kette	Unter einer anaphorischen Kette kann, vereinfacht gesagt, die Verknüpfung eines Wortes (bspw. ein Personalpronomen) mit einem Bezugspunkt verstanden werden. Bspw. besteht in dem Satz „Hans kam gestern nach Hause, er ging dann gleich schlafen.“ eine anaphorische Beziehung zwischen „er“ und „Hans“.
API (Aplication Programming Interface)	Unter einer API wird eine Schnittstelle verstanden, über die ein Programm mit einem anderen verbunden werden kann. Eine API stellt in der Regel Zugriffsmethoden für das Programm bereit, für das sie erstellt wurde. Im Rahmen der Metamodellerzeugung besteht eine API bspw. aus Methoden, um auf die Metamodellelemente zuzugreifen.
Baum	Ein Baum ist ein Graph mit einem ausgezeichneten Knoten als Wurzel. Formal ist ein Baum Folgendes: Sei $T = (V, E)$ ein Graph, so gilt: T ist ein Baum genau dann, wenn T zusammenhängend und kreisfrei ist.
CSV	CSV steht für **C**haracter **S**eparated **V**alues und bezeichnet ein Dateiformat. Die einzelnen Daten in einem solchen Format werden durch ein festgelegtes Zeichen (Character) voneinander getrennt. Tabulator-Dateien (tab-Dateien), die oft für den Datenaustauch in relationalen Datenbanken genutzt werden, sind bspw. CSV-Dateien.
DAG	Ein Directed Acyclic Graph ist ein kreisfreier gerichteter Graph. Das bedeutet, eine Kante hat eine ausgewiesene Richtung. Dabei wird der erste Knoten als Quell- und der zweite Knoten als Zielknoten bezeichnet. Zudem darf es zwischen zwei beliebigen Knoten nur genau einen Pfad (Weg) geben.
Eclipse	Mit Eclipse kann zum einen die Eclipse Foundation (siehe http://www.eclipse.org/) und zum anderen die Eclipse IDE, eine integrierte Entwicklungsumgebung der Eclipse Foundation bezeichnet werden.

Ecore	Ecore ist eine Modellierungssprache, die im Rahmen des EMF von der Eclipse Foundation entwickelt wurde. Ecore ist kompatibel zu eMof und besteht aus einer Reihe von Metamodellelementen, die miteinander in Beziehung gesetzt werden können, um Metamodelle zu erzeugen.
EMF	Das Eclipse Modeling Framework stellt eine Reihe von Techniken zur Modellierung bereit. Unter den Schirm der EMF fallen Ecore, das GMF (Graphical Modeling Framework) zur grafischen Erzeugung von Metamodellen, die XMI-Persistenzierung von Modellen und einige weitere Techniken.
eMOF	eMof steht für Essential Meta Object Facility und ist die von der OMG entwickelte Kernsprache zur Modellentwicklung. eMof ist die Metasprache der UML und weiterer von der OMG entwickelten Modellierungssprachen .
Framework	Der Begriff Framework (zu dt. Fachwerk) ist ein weitläufiger Begriff und bezeichnet in der Informatik eine Art Programmiergerüst. Ein Framework definiert einige Vorgaben, die zu erfüllen sind um ein Programm in dieses Framework einzubinden. Wie diese Spezifikationen aussehen, ob sie auf Papier oder direkt in Programmcode vorgegeben werden, ist sehr unterschiedlich.
Get-Set-Methode	Get- und Set-Methoden werden in der Programmierung eingesetzt, um einen Zugriff auf Attribute bestimmter Objekte zu erhalten. Besitzt ein Objekt das Attribut „att1“, so kann für dieses eine Methode „getAtt1“ und „setAtt1“ erzeugt werden, um dieses Attribut auszulesen bzw. um es mit einem Wert zu belegen.
Graph	Ein Graph $G = (V, E)$ ist ein Paar, bestehend aus einer Menge von Knoten (V) und einer Menge von Kanten (E). Ein Knoten $v \in V$ ist ein Punkt innerhalb des Graphen, eine Kante $e \in E = (VxV)$ ist ebenfalls ein Paar, bestehend aus zwei Knoten.

Hashtable Hashtable.html.	Unter einer Hashtable wird eine Schküssel-Wert-Zuordnung verstanden. Ein Schlüssel ist ein Identifizierer für einen Wert, der meist aus einer Menge an Objekten besteht. Über verschiedene Speichermethoden bietet eine Hashtable bezogen auf den Schlüssel eine relativ schnelle Zugriffsmethode auf den Wert. Eine Hashtable ist auch ein konkretes Objekt der Programmiersprache *Java* siehe hierfür http://java.sun.com/j2se/1.4.2/docs/api/java/util/
IDE	IDE steht für integrated development environment, zu Deutsch integrierte Entwicklungsumgebung. Darunter ist in der Regel eine grafische Umgebung zur Entwicklung von Software zu verstehen. Eine IDE bündelt die Quelltexterzeugung, Compiler und Debugger sowie eine Reihe anderer Techniken, die das Entwickeln vereinfachen sollen. Beispiele für IDE´s sind u.a. Eclipse oder Netbeans.
ISO	ISO steht für International Organisation for Standardisation, zu Deutsch internationale Standardisierungsorgansisation. Die ISO beschäftigt sich in vielen Geschäftsfeldern mit der internationalen Standardisierung verscheidener Techniken, Formate oder Vorgehensweisen. Näheres ist unter http://www.iso.org/ zu finden.
Java	Java ist eine Programmiersprache, die von der Firma Sun Microsystems entwickelt wurde. Java ist objektorientiert und zeichnet sich gegenüber anderen objektorientierten Sprachen wie C++ dadurch aus, dass es interoperabel auf verschiedenen Betriebssystemen eingesetzt werden kann, ohne ein Programm neu zu kompilieren.
Konvertierung	Unter einer Konvertierung ist im Rahmen dieser Arbeit die Überführung eines linguistischen Annotationsformates in ein anderes linguistisches Annotationsformat zu verstehen. Der Begriff Konvertierung wird sowohl als Beschreibung des Prozesses als auch als Prozessdurchführung verwendet.

Konverter	Ein Konverter ist ein Werkzeug, bzw. ein Modul, dass eine Konvertierung vornimmt.
Konverterframework	Ein Konverterframework ist eine Umgebung mit klar definierten Schnittstellen, in die verschiedene Konverter integriert werden können. Ein Konverterframework übernimmt damit nicht den tatsächlichen Prozess der Konvertierung, sondern bildet lediglich eine Grundlage, auf der die Konverter aufgebaut sind. Im Rahmen dieser Arbeit beinhaltet das hier entwickelte Konverterframework Pepper zusätzlich noch die Adminsistration der integrierten Konverter, sowie die Programmflusssteuerung.
M2M	Unter M2M verstehe ich hier die Model-2-Model-Transformation. Also eine Abbildung von einem Modell auf ein anderes Modell.
M2T	Unter M2T verstehe ich hier die Model-2-Text-Transformation. Dabei wird aus einem Metamodell ein Text erzeugt. Dieser Text kann auch der Programmcode in einer bestimmten Programmiersprache zu einem Metamodell (also aine API) sein.
Modul	Allgemein ist ein Modul, in der Informatik, ein Programmteil, der eine klare Schnittstellenbeschreibung hat und relativ selbständig eine beschränkte Aufgabe übernimmt. Module sind zudem eine Sammlung von Objekten, die eine gleiche Funktion haben. In dieser Arbeit wird unter einem Modul auch ein spezieller Teil innerhalb des Frameworks Pepper bezeichnet. Eine Erklärung hierzu findet sich in Abschnitt 5.1.1.
OCL	OCL steht für Object Constraint Language und ist eine deklarative Beschreibungssprache für Modellzustände. Mit OCL können Bedingungen formal festgelegt werden, die ein valides Modell erfüllen muss. OCL ist ein von der OMG vorgeschlagener Standard.
protected regions	Eine protected region bezeichnet in der Modellierung die Möglichkeit, einen Bereich in einem Quellcode als geschützt zu markieren. Diese Markierung soll dann von einem eine M2T ausführenden Programm berücksichtigt werden. Das bedeutet, wenn ein solches Programm aus einem Modell Quellcode erzeugt, soll der geschützte Bereich nicht verändert, vor allem nicht überschrieben werden.

SGML	SGML ist die Abkürzung für **S**tandard **G**eneralized **M**arkup **L**anguage. SGML ist eine Sprach die die Idee verfolgt, Inhalt und Layout eines Dokumentes voneinander zu trennen. Dies wird mit sogenannten Tags ermöglicht. SGML gilt als Vorläufer von HTML und XML.
Service Platform	Die Service Platform ist die von OSGi verwendete Laufzeitumgebung. Sie sorgt u.a. dafür, dass alle Bundles installiert und gestartet werden können (siehe Abschnitt 2.7).
Service Registry	Die Service Registry ist ein Dienst, den die Service Platform anbietet. Dieser Dienst registriert alle in der OSGi-Laufzeitumgebung installierten Bundles mitsamt ihrer Abhängigkeiten und angebotenen Dienste.
Transformation	siehe Abschnitt 2.3
Traversierung	Unter einer Traversierung ist in der Graphentheorie die Durchwandereung eines Graphen zu verstehen. Dabei wird in einer festgelegten Reihenfolge, ausgehend von einem Knoten, ein Nachbarknoten, über eine beide Knoten verbindende Kante, besucht. Eine vollständige Traversierung liegt dann vor, wenn jeder Knoten über jede Kante im Graphen besucht wurde.
UML	UML steht für Unified Modelling Language und ist in erster Linie eine Sammlung von Diagramarten, um Software zu beschreiben. UML wurde von der OMG entwickelt und hat sich inzwischen zu einer Art Standard auf dem Gebiet der modellbasierten Entwicklung vorgearbeitet.
URI	URI steht für Unified Resource Identifier und bezeichnet die eindeutige Identifizierung eines Objektes. Dieses Objekt kann eine Internetreferenz, eine lokale Datei oder auch ein Objekt im Hauptspeicher sein.

B. Erklärungen

B.1. Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich dies vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Berlin, ...

B.2. Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, ...

Literaturverzeichnis

[Apache Foundation, 2009] Apache Foundation (2009). Apache felix. http://felix.apache.org/site/index.html. zuletzt besucht am 15.08.2009.

[Bies, 1995] Bies, A. (1995). Bracketing guidelines for treebank ii style penn treebank project.

[Clark, 1999] Clark, J. (1999). Xsl transformations (xslt), version 1.0. http://www.w3.org/TR/xslt. zuletzt besucht am 06.10.2009.

[Dipper, 2005] Dipper, S. (2005). XML-based stand-off representation and exploitation of multi-level linguistic annotation. In Eckstein, R. and Tolksdorf, R., editors, Berliner XML Tage, pages 39–50.

[Eclipse Foundation, 2009a] Eclipse Foundation (2009a). Eclipse modeling framework project (emf). http://www.eclipse.org/modeling/emf/. zuletzt besucht am 30.11.2009.

[Eclipse Foundation, 2009b] Eclipse Foundation (2009b). Graphical modeling framework. http://www.eclipse.org/modeling/gmf/. zuletzt besucht am 29.09.2009.

[Eclipse Foundation, 2009c] Eclipse Foundation (2009c). Model query. http://www.eclipse.org/modeling/emf/?project=query#query. zuletzt besucht am 30.11.2009.

[Eclipse Foundation, 2009d] Eclipse Foundation (2009d). Model to model (m2m). http://www.eclipse.org/m2m/. zuletzt besucht am 01.08.2009.

[GAAV, 2006] GAAV (2006). Prague markup language (pml). http://ufal.mff.cuni.cz/jazz/PML/index_en.html. zuletzt besucht am 14.08.2009.

[Grust et al., 2004] Grust, T., van Keulen, M., and Teubner, J. (2004). Accelerating XPath evaluation in any RDBMS. ACM Trans. Database Syst., 29:91–131.

[Ide and Romary, 2003] Ide, N. and Romary, L. (2003). Outline of the international standard linguistic annotation framework. In Proceedings of the ACL 2003 workshop on Linguistic annotation, pages 1–5, Morristown, NJ, USA. Association for Computational Linguistics.

[Ide et al., 2007] Ide, N., Suderman, K., and (2007). GrAF: A graph-based format for linguistic annotations. In Proceedings of the Linguistic Annotation Workshop, pages 1–8, Prague, Czech Republic. Association for Computational Linguistics.

[ikv++ technologies ag, 2009] ikv++ technologies ag (2009). medini qvt engine. http://www.ikv.de/index.php?option=com_content&task=view&id=75&Itemid=77. zuletzt besucht am 20.09.2009.

[ISO-TC-37, 2009] ISO-TC-37 (2009). Data Category Registry-Defining widely accepted linguistic concepts. ISO, http://www.isocat.org/. zuletzt besucht am 17.08.2009.

[JUNG, 2009] JUNG (2009). Jung - java universal network/graph framewor. http://jung.sourceforge.net/. zuletzt besucht am 26.05.2009.

[Kurtev et al., 2002] Kurtev, I., Bezivin, J., and Aksit, M. (2002). Technological spaces: An initial appraisal. In International Symposium on Distributed Objects and Applications, DOA 2002.

[Lüdeling et al., 2008] Lüdeling, A., Doolittle, S., Hirschmann, H., Schmidt, K., and Walter, M. (2008). Das lernerkorpus falko. In Deutsch als Fremdsprache, pages 67–73.

[Lothar Lemnitzer, 2006] Lothar Lemnitzer, H. Z. (2006). Korpuslinguistik. Eine Einführung. narr studienbücher.

[Marsh et al., 2005] Marsh, J., Veillard, D., and Walsh, N. (2005). xml:id version 1.0. http://www.w3.org/TR/xml-id/. zuletzt besucht am 19.09.2009.

[Mengel and Lezius, 2000] Mengel, A. and Lezius, W. (2000). An XML-based encoding format for syntactically annotated corpora. In Proceedings of the Second International Conference on Language Resources and Engineering (LREC 2000), pages 121–126, Athen.

[Mens and Gorp, 2005] Mens, T. and Gorp, P. V. (2005). A taxonomy of model transformation. In Proc. Int'l Workshop on Graph and Model Transformation.

[Microsystems, 2009] Microsystems, S. (2009). Class transformer. http://java.sun.com/j2se/1.4.2/docs/api/javax/xml/transform/Transformer.html. zuletzt besucht am 20.09.2009.

[Miller and Mukerji, 2003] Miller, J. and Mukerji, J. (2003). Mda guide version 1.0.1. Technical report, Object Management Group (OMG).

[Müller and Strube, 2006] Müller, C. and Strube, M. (2006). Multi-level annotation of linguistic data with MMAX2. In Braun, S., Kohn, K., and Mukherjee, J., editors, Corpus Technology and Language Pedagogy: New Resources, New Tools, New Methods, pages 197–214. Peter Lang, Frankfurt a.M., Germany.

[Nolte, 2009a] Nolte, S. (2009a). QVT - Operational Mappings: Modellierung mit der Query Views Transformation. Springer, Berlin.

[Nolte, 2009b] Nolte, S. (2009b). QVT - Relations Language: Modellierung mit der Query Views Transformation. Springer-Verlag Berlin Heidelberg.

[O., 1994] O., C. (1994). A modular and flexible architecture for an integrated corpus query system.

[OMG, 2002] OMG (2002). Meta-object facility (mof) 1.4 specification. http://www.omg.org/technology/documents/formal/mof.htm. zuletzt besucht am 15.06.2009.

[OMG, 2006] OMG (2006). Object Constraint Language Object Constraint Language, OMG Available Specification, Version 2.0.

[OMG, 2007] OMG (2007). Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification. `http://www.omg.org/spec/QVT/1.0/PDF/`. zuletzt besucht am 26.08.2009.

[OMG, 2008] OMG (2008). Mof model to text transformation language. `http://www.omg.org/spec/MOFM2T/1.0/`. zuletzt besucht am 18.09.2009.

[OMG, 2009a] OMG (2009a). The object management group (omg). `http://www.omg.org/`. zuletzt besucht am 30.11.2009.

[OMG, 2009b] OMG (2009b). Omg unified modeling languagetm (omg uml),infrastructure. `http://www.omg.org/technology/documents/formal/uml.htm`. zuletzt besucht am 30.11.2009.

[OSGiAlliance, 2009] OSGiAlliance (2009). Osgi - the dynamic module system for java. `http://www.osgi.org`. zuletzt besucht am 05.07.2009.

[Project, 2009] Project, T. E. (2009). Elver. zuletzt besucht am 30.10.2009.

[Reitter and Stede, 2003] Reitter, D. and Stede, M. (2003). Step by step: underspecified markup in incremental rhetorical analysis. In Proceedings of the 4th International Workshop on Linguistically Interpreted Corpora (LINC-03) (at EACL 2003), pages 77–84, Budapest, Hungary.

[Schmid, 1994] Schmid, H. (1994). Probabilistic part-of-speech tagging using decision trees. In Proceedings of International Conference on New Methods in Language Processing.

[Schmidt, 2002] Schmidt, T. (2002). Exmaralda - ein system zur diskurstranskription auf dem computer. Arbeiten zur Mehrsprachigkeit, Folge B, 34:1 ff. DE.

[Schmidt, 2005] Schmidt, T. (2005). Time-based data models and the text encoding initiative's guidelines for transcription of speech. Arbeiten zur Mehrsprachigkeit, Folge B, 62:1 ff. EN.

[Schmidt, 2009] Schmidt, T. (2009). Exmaralda. `http://www.exmaralda.org/`. zuletzt besucht am 30.07.2009.

[SFB 632, 2007] SFB 632 (2007). The paula standoff format. `http://www.sfb632.uni-potsdam.de/~d1/paula/doc/`. zuletzt besucht am 25.06.2009.

[SFB 632, 2009] SFB 632 (2009). Annis query language - aql. http://www.sfb632.uni-potsdam.de/d1/annis/. zuletzt besucht am 30.11.2009.

[spring source community, 2009] spring source community (2009). Spring dynamic modules for osgi(tm) service platforms. http://www.knopflerfish.org/. zuletzt besucht am 15.08.2009.

[Steinberg et al., 2009] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional.

[Stuttgart, 2003] Stuttgart, U. (2003). The tiger-xml treebank encoding format. Technical report, Universität Stuttgart.

[Stuttgart, 2009] Stuttgart, U. (2009). Treetagger. http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/. zuletzt besucht am 06.06.2009.

[The Knopflerfish Project, 2009] The Knopflerfish Project (2009). Knopflerfish - open source osgi. http://www.knopflerfish.org/. zuletzt besucht am 15.08.2009.

[Universität-Stuttgart et al., 2007] Universität-Stuttgart, des Saarlands, U., and Universität-Potsdam (2007). TIGER PROJECT Linguistic Interpretation of a German Corpus.

[Vitt, 2005] Vitt, T. (2005). DDDquery: Anfragen an komplexe Korpora. Diplomarbeit, Universität zu Berlin, Berlin, Germany.

[White, 2009] White, T. (2009). Hadoop: The Definitive Guide. O'Reilly Media, Inc.

[Wütherich et al., 2008] Wütherich, G., Hartmann, N., Kolb, B., and Lübken, M. (2008). Die OSGi Service Platform: Eine Einführung mit Eclipse Equinox. dpunkt, Heidelberg.

[Zeldes et al., 2009] Zeldes, A., Ritz, J., Lüdeling, A., and Chiarcos, C. (2009). Annis: A search tool for multi-layer annotated corpora. In Proceedings of Corpus Linguistics 2009, Liverpool, July 20-23, 2009.