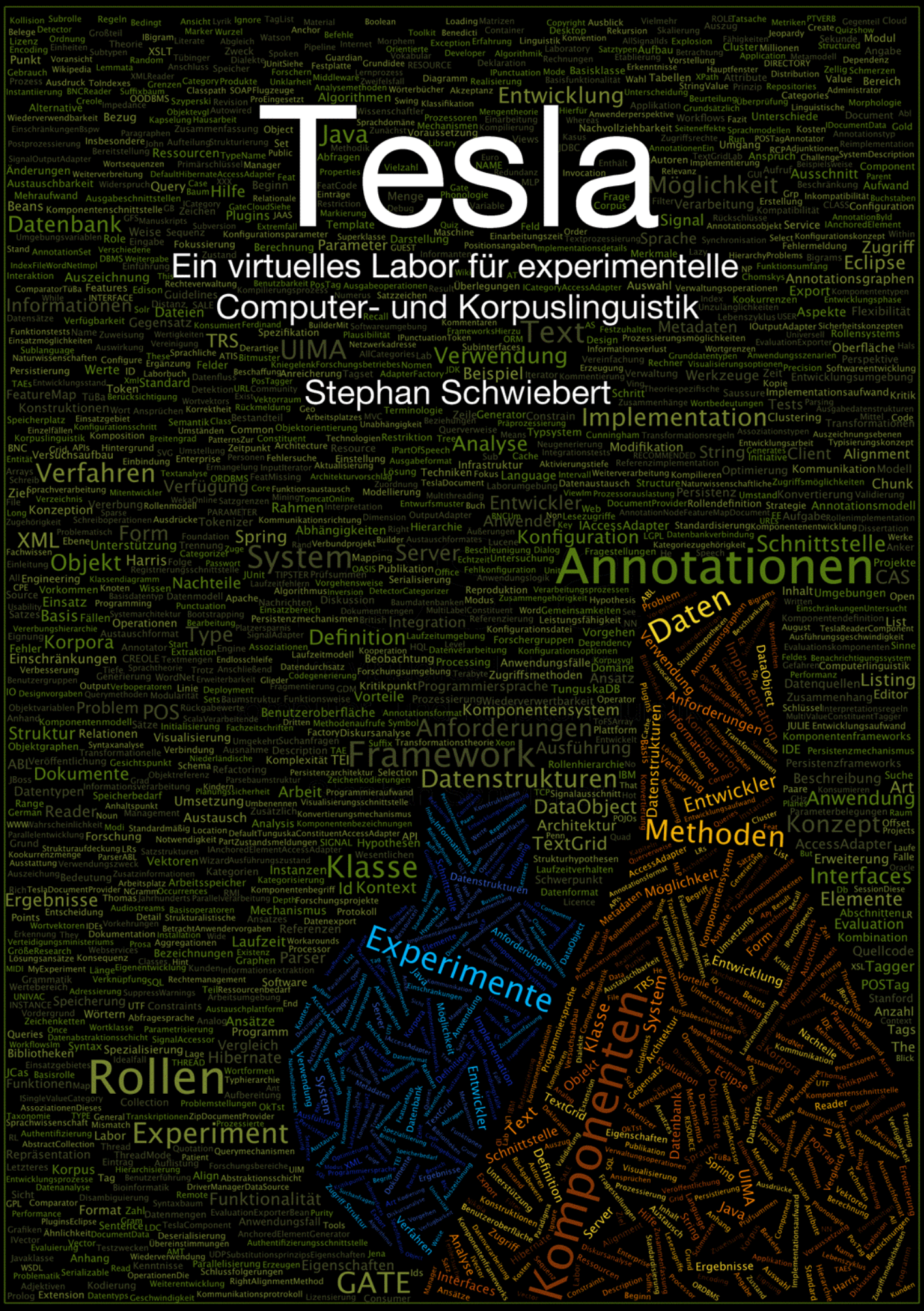
Tesla
Ein virtuelles Labor für experimentelle
Computer- und Korpuslinguistik
Stephan Schwiebert

Vorwort

> Though free to think and act, we are held together, like the stars in the firmament, with ties inseparable. These ties cannot be seen, but we can feel them.
>
> *(Nikola Tesla)*

Dieses Dokument ist die aktualisierte Fassung meiner Dissertation, die im Wintersemester 2011/2012 von der Philosophischen Fakultät der Universität zu Köln angenommen wurde. Die Referenten waren Prof. Dr. Jürgen Rolshoven und Prof. Dr. Manfred Thaller. Die Disputation fand am 18. Januar 2012 statt.

Auslöser meines Promotionsvorhabens war ein Stellenangebot als wissenschaftlicher Mitarbeiter in der Sprachlichen Informationsverarbeitung der Uni Köln. Damit verbunden bot sich die einmalige Gelegenheit, ein sowohl inhaltlich als auch softwaretechnisch äußerst interessantes und anspruchsvolles Projekt zu konzipieren und umzusetzen. Abgesehen von der Kernfunktionalität des zu implementierenden Systems hatten wir keine weiteren Vorgaben, sondern durften vielmehr unseren Ideen und Wünschen frei nachgehen. Meinem Doktorvater, Prof. Dr. Jürgen Rolshoven, möchte ich daher nicht nur für seinen Rat und seine Unterstützung in zahlreichen Gesprächen, sondern insbesondere auch für das Vertrauen, das er uns entgegengebracht hat, danken.

Wie aus dem vorangegangenen Abschnitt bereits ersichtlich wird, ist Tesla kein Ein-Mann-Projekt – im Laufe der Zeit wurde die Entwicklung u.a. durch zahlreiche inzwischen ehemalige studentische Hilfskräfte vorangetrieben, denen hier ebenfalls gedankt sei. Großer Dank gebührt auch dem Promotions-Leidensgenossen, Tesla-Mitentwickler und Schreibtisch-Nachbarn Dr. Jürgen Hermes, denn ohne Zusammenarbeit, Diskussion und gegenseitige Inspiration wäre Tesla nicht das System, das es heute ist. Ebenfalls möchte ich ihm für seine Anmerkungen zu meiner Dissertation danken, auch wenn ich manchmal vermute, dass er nur deshalb Kritik geäußert hat, damit er das Kopf-an-Kopf-Rennen gewinnen und somit an dieser Stelle bereits mit neuem Titel erwähnt werden kann.

Neben den eingangs erwähnten Referenten ist Claes Neuefeind die zum Zeitpunkt der Veröffentlichung einzige Person, die diese Arbeit vollständig gelesen hat, insbesondere aber hat er mir wertvolle Hinweise und inhaltliche Anmerkungen zu jedem Kapitel gegeben. Hierfür danke ich ihm sehr, ebenso wie für zahlreiche weitere gute Ratschläge, die teilweise universitäre, teilweise aber auch eher weltliche Aspekte betrafen.

Weiterer Dank gebührt ehemaligen Kommilitonen und ehemaligen Mitarbeitern der Sprachlichen Informationsverarbeitung: Christoph Benden danke ich für seine Diskussionsbereitschaft und sein großartiges und konstruktives Feedback zum Thema Strukturalismus, aber auch für den sehr, sehr guten Vorschlag, das Projekt *Tesla* zu nennen. Sonja Subicin danke ich für die Bereitschaft, ihre magischen Fähigkeiten hinsichtlich Grafik und Design auf zahlreiche Abbildungen in dieser Arbeit anzuwenden, Daniel Alberts danke ich insbesondere für diverse softwaretechnische Anmerkungen und Tipps. Fabian Steeg danke ich schließlich für sein allumfassendes Wissen hinsichtlich der Entwicklung von Eclipse-Plugins, und natürlich für die großartige Idee, den Quellcode von *Cloudio* an das GEF-Projekt zu spenden.

Alena Geduldig danke ich für ihre großartige Last-Minute-Unterstützung durch die Implementation diverser Komponenten (siehe Anhang B). Den weiteren Hilfskräften der Sprachlichen Informationsverarbeitung sei ebenfalls gedankt: Mandy Neumann für die Auseinandersetzung mit der Hierarchisierung von POS-Tags, Frauke Schmidt für engelsgleiche Geduld beim Administrieren der Server und Miha Atanassov für die Implementation der Volltextsuche im Tesla-Client.

Allen erwähnten Personen danke ich zudem für ihre Bereitschaft, verschiedene Aspekte aus Softwareentwicklung und -architektur zu diskutieren: Durch ihre Einwände und Ergänzungen konnten verschiedene Probleme frühzeitig erkannt und vermieden werden. Hin und wieder ist es jedoch auch hilfreich, Idee und Motivation eines solchen Projektes mit fachfremden Personen zu diskutieren, weshalb ich an dieser Stelle Tina Lahl für die Gelegenheit danken möchte, sowohl Tesla als auch meine Disputationsthesen mit einer Promotionsstudentin der Germanistik diskutieren zu dürfen.

Danken möchte ich auch meiner Familie, die mich stets unterstützt hat, ohne genau zu wissen, worum es bei dieser Arbeit eigentlich geht.

Torsten Keller und Kathi Schürmann möchte ich ebenfalls meinen Dank aussprechen: Für die häufig in Anspruch genommene Möglichkeit, auf dem Rückweg von der Universität noch einen Kaffee zu trinken, und für den dabei gerne wiederholten Vorschlag, die Arbeit doch endlich abzuschließen.

Schließlich danke ich all jenen Freunden in Köln, Hamburg, Bern, London, Sydney und andernorts: Die Energie, die in die Fertigstellung dieser Arbeit geflossen ist, wurde nicht zuletzt durch sie stets regeneriert.

Inhaltsverzeichnis

1 Einleitung

> Our first endeavors are purely instinctive prompting of an imagination vivid and undisciplined. As we grow older reason asserts itself and we become more and more systematic and designing. But those early impulses, though not immediately productive, are of the greatest moment and may shape our very destinies.
>
> *(Nikola Tesla)*

Die Computerlinguistik kann als eine der ältesten Disziplinen der angewandten Informatik betrachtet werden – nicht zuletzt aufgrund der Tatsache, dass es sich bei jeder Programmiersprache um eine formale Sprache handelt, die von einem Parser in eine maschinennahe Repräsentation übersetzt werden muss. Dennoch liegen für die meisten computerlinguistischen Problemstellungen noch keine optimalen Lösungsansätze vor – bereits die scheinbar triviale Aufgabe, Satzgrenzen innerhalb eines unstrukturiert vorliegenden Textes zu ermitteln, konnte bisher nicht völlig fehlerfrei umgesetzt werden (vgl. bspw. Gillick 2009). Umgekehrt hat die Menge der potentiellen Anwendungsfelder für computerlinguistische Algorithmen und Lösungsansätze in den letzten Jahrzehnten drastischen Zuwachs erhalten: Ein Großteil der Kommunikation findet seit langem digital statt, der textbasierte Informationsaustausch (u.a. durch Email, SMS, Blogs, Foren oder sonstige Kommentarfunktionen diverser Web-Angebote) steht dabei an erster Stelle – eine Beschleunigung des Forschungsbetriebes im *Text Engineering* ist somit nicht nur aus wissenschaftlicher, sondern insbesondere auch aus praktischer Sicht wünschenswert, umfasst dieser doch zahlreiche wirtschaftlich nicht uninteressante Anwendungsgebiete wie etwa maschinelle Übersetzung, *Text Mining*, *Information Extraction* oder *Sentiment Detection*. Dies wiederum ist nur möglich, wenn der inter- und multidisziplinäre Austausch zwischen Wissenschaftlern ebenso wie der Austausch mit Anwendern verbessert wird, wenn der Zugang zu und

die Bereitstellung von neu entwickelten Verfahren vereinfacht wird, und wenn die in diesem Zusammenhang in wissenschaftlichen Publikationen veröffentlichten Ergebnisse nicht nur nachvollzogen, sondern auch unmittelbar am jeweiligen Forschungsgegenstand evaluiert werden können. Von diesem Zustand ist die aktuelle Computerlinguistik jedoch noch weit entfernt.

Die vorliegende Arbeit schlägt einen möglichen Lösungsweg vor: Ihr Schwerpunkt liegt in der Konzeption, Architektur und Implementation eines computerlinguistischen Komponentensystems, das die geschilderten Kritikpunkte ebenso beseitigen kann wie es Nachteile und Einschränkungen, die in bereits existierenden Systemen auftreten können, vermeidet.

Die Argumentation in dieser Arbeit entspricht der Aufteilung in einzelne Kapitel und gliedert sich wie folgt:

- Kapitel 2 wird anhand der von Zellig Harris entwickelten Verfahren zur distributionellen Analyse den theoretischen Hintergrund diverser unüberwacht arbeitender Verfahren zur Extraktion syntagmatischer oder paradigmatischer Relationen aus Texten vorstellen und somit einen exemplarischen Anwendungsfall beschreiben. Zudem wird mit *Alignment Based Learning* (ABL) eines dieser Verfahren vorgestellt und dessen Funktionsweise näher erläutert, um anhand dieses Beispiels die technischen Anforderungen an ein computerlinguistisches Komponentensystem definieren zu können.

- In Kapitel 3 wird auf Basis dieser Anforderungen eine Evaluation bereits vorhandener Systeme zur Verarbeitung natürlichsprachlicher Texte durchgeführt; untersucht werden Konzept und Architektur von *GATE*, *UIMA* und *TextGrid*. Dabei wird sich herausstellen, dass keines der genannten Systeme für ein experimentelles, forschungsorientiertes Arbeiten (wie oben knapp und in Kapitel 2 ausführlich beschrieben) geeignet ist.

- Daher werden in Kapitel 4 Konzepte, Architektur und Implementation des *Text Engineering Software Laboratory* (Tesla) vorgestellt, das im Kontext dieser Arbeit und der Dissertation von Hermes (2011) entwickelt wurde, das die in Kapitel 2 aufgeführten Anforderungen erfüllt und das gleichzeitig viele der in Kapitel 3 veranschaulichten Einschränkungen umgeht – dieses Kapitel stellt den softwaretechnologischen Schwerpunkt der Arbeit dar.

- In Kapitel 5 werden die Überlegungen aus Kapitel 2 mit der in Kapitel 4 vorgestellten Architektur kombiniert und es wird gezeigt, wie sich experimentelles, forschungsorientiertes und den wissenschaftlichen Austausch unterstützendes Arbeiten mit Tesla umsetzen lässt. Die ABL-Verfahren werden nicht nur an verschiedenen Korpora evaluiert, sondern auch bezüglich der Prä- und Postprozessierung variiert, um verschiedene Hypothesen zu überprüfen und ggfs. zu falsifizieren. Ferner werden die Experimente erweitert, um einzelne Aspekte der verwendeten Verfahren zu optimieren.

- Schließlich werden in Kapitel 6 die Ergebnisse dieser Arbeit zusammengefasst und diskutiert; zudem wird dort ein Ausblick auf mögliche weiterführende Entwicklungen gegeben.

Festzuhalten ist, dass es nicht der Anspruch dieser Arbeit ist, ein von der strukturalistischen Sprachwissenschaft motiviertes Verfahren zur Extraktion syntagmatischer und paradigmatischer Relationen aus Texten vollständig umzusetzen.[1] Vielmehr wird gezeigt, wo die Möglichkeiten und Grenzen unüberwachter Alignment-Verfahren beim syntaktischen Bootstrapping liegen, indem unterschiedliche Ansätze kritisch evaluiert werden und die Plausibilität der Ergebnisse diverser Publikationen zu diesem Thema überprüft wird.

Die in den Kapiteln 2 und 5 vorgestellten und evaluierten Verfahren repräsentieren eine Teilmenge der Verfahren zur Informationsextraktion. Zu diesen gehören bspw. auch Clustering- und Klassifikationsansätze, die u.a. zur semantischen Kategorisierung von Dokumenten oder zur Disambiguierung von Wortbedeutungen verwendet werden können (vgl. etwa Manning *et al.* 2008, Kapitel 6 und Witten *et al.* 2005, Kapitel 3, siehe auch Kapitel 5.5.3 dieser Arbeit).

Ihnen ist gemein, dass sie auf unstrukturierten Daten operieren und kein linguistisches Zusatzwissen benötigen, was u.a. für die Verarbeitung großer Textmengen von Vorteil ist. Problematisch ist jedoch insbesondere die Evaluation und der Vergleich der unterschiedlichen Ansätze: Für eine wissenschaftlich korrekte Analyse muss ein Gold-Standard verwendet werden, in dem hier skizzierten Einsatzgebiet also ein Korpus, das manuell mit zusätzlichen Informationen annotiert wurde, die syntaktische und semantische Merkmale der Texte abbilden – nur so kann gewährleistet werden, dass die Ergebnisse einer Untersuchung von Dritten reproduzierbar und mit alternativen Verfahren vergleichbar sind.

[1] Tatsächlich werfen die in Kapitel 2 aufgeführten, theoriebezogenen Kritikpunkte ebenso wie die in 5 diskutierten, anwendungsbezogenen Probleme die Frage auf, ob ein derartiges Ziel mit ausschließlich strukturalistisch motivierten, symbolverarbeitenden Verfahren überhaupt erreicht werden kann. Diese Frage wird in Abschnitt 5.6 erneut diskutiert.

Im Fall von dokumentbasierten Clustering-Verfahren ist dies vergleichsweise einfach umzusetzen: So können Dokumente beispielsweise manuell einer oder mehreren Kategorien zugeordnet werden, so dass zur Evaluation lediglich die Übereinstimmungen und Abweichungen zwischen Clustern und Kategorien analysiert werden müssen. Deutlich komplexer ist die Evaluation von Verfahren bereits im Fall von Clustering auf Wortebene, wie etwa bei Verfahren zur Disambiguierung von Wortbedeutungen: Zwar könnte bspw. ein Vergleich der Cluster mit den über *WordNet* zu Verfügung gestellten Assoziationen[2] einen ersten Anhaltspunkt liefern, für aussagekräftige Ergebnisse muss jedoch auch hier ein manuell annotierter Gold-Standard verwendet werden (vgl. Edmonds & Kilgarriff 2002), oder zumindest, wie im Rahmen der *Senseval*-Workshops (siehe http://www.senseval.org/[3]) geschehen, ein gemeinsam definiertes Evaluationsverfahren angewandt werden (vgl. Palmer *et al.* 2006). Dieses Vorgehen scheitert jedoch bei paradigmatischen Relationen zwangsläufig, wie bereits de Saussure (1931, S. 151) feststellte:

> Während ein Syntagma sofort die Vorstellung einer Anordnung in der Aufeinanderfolge und einer bestimmten Anzahl von Bestandteilen hervorruft, bieten sich die Glieder assoziativer Art weder in bestimmter Zahl noch in bestimmter Ordnung dar [...und man kann] nicht von vornherein sagen, wie groß die Anzahl der Wörter sein wird, die das Gedächnis darbietet, noch in welcher Ordnung sie auftreten.

Um dennoch auch in diesem Bereich eine Strategie zur Evaluation bzw. zum Vergleich verschiedener Verfahren anbieten zu können, müsste zunächst ein Korpus erstellt werden, in dem sämtliche paradigmatischen Relationen aller enthaltenen Ausdrücke abgebildet wurden. In Ermangelung eines solchen Korpus wird in Kapitel 5 lediglich eine syntagmatische Evaluation an vier Korpora und anhand von jeweils zwei unterschiedlichen Referenzen durchgeführt – es wird jedoch gezeigt, wie sich die untersuchten Verfahren um die Detektion paradigmatischer Relationen erweitern lassen (etwa indem Verfahren zur Erkennung syntaktischer Strukturen mit den oben erwähnten Clustering-Verfahren kombiniert werden) und wie eine entsprechende Evaluierung umgesetzt werden kann, sobald ein die Anforderungen erfüllendes Korpus verfügbar ist.

Zudem ist jedoch auch eine Evaluation auf syntagmatischer Ebene (bspw. durch Vergleich der von einem Verfahren generierten Strukturgrenzen mit den in einem Gold-Korpus

[2] Dieses Beispiel wird in Kapitel 5.5.3 erneut aufgegriffen und vertieft.
[3] Sämtliche in dieser Arbeit angegebenen URLs wurden am 14.11.2011 zuletzt überprüft.

annotierten Daten) nicht zwingend objektiv: Üblicherweise werden *Precision* und *Recall* zur Bewertung eines Verfahrens herangezogen, wie in Edelman *et al.* (2004) oder van Zaanen & Geertzen (2008). Precision beschreibt das Verhältnis zwischen der Menge von Strukturen, die von einem Verfahren entdeckt wurden (A) zu der Menge aller vorhandenen Strukturen (B) und ist i.d.R. definiert als $\frac{|A \cap B|}{|A|}$. Recall gibt umgekehrt an, wie hoch der Anteil der entdeckten Strukturen an der Zahl sämtlicher Strukturen ist, formell darstellbar durch $\frac{|A \cap B|}{|B|}$ (vgl. auch Witten *et al.* 2005, S. 171).

Die Ergebnisse beider Metriken sind jedoch nicht ohne weitere Informationen zu den untersuchten Korpora und den berücksichtigten Strukturen interpretierbar, insbesondere eignen sich diese Werte nicht für den Vergleich unterschiedlicher Verfahren (etwa anhand von in Fachpublikationen veröffentlichten Daten): So weisen bspw. van Zaanen & Geertzen (2008) darauf hin, dass *triviale* Strukturen (wie Wort- und Satzgrenzen) bei der Evaluation nicht berücksichtigt werden sollten, um die Ergebnisse nicht zu verfälschen. Weitere Schwierigkeiten in der Evaluation von Verfahren zur Strukturaufdeckung werden bspw. von Cramer (2007) beschrieben – die Ergebnisse der dort analysierten Verfahren ADIOS (vgl. Edelman *et al.* 2004) und ABL entsprechen in keiner Weise den Ergebnissen, die von den jeweiligen Entwicklern publiziert wurden. Am Ende von Kapitel 5 wird sich zwar zeigen, dass einige der von Cramer (2007) beschriebenen Ergebnisse mit großer Wahrscheinlichkeit auf eine fehlerhafte Analyse zurückgeführt werden können – das Beispiel macht jedoch deutlich, dass eine Evaluation derartiger Verfahren komplex ist und die Nachvollziehbarkeit von Untersuchungen (und deren Ergebnissen) verbesserungswürdig ist.

Design, Konzeption und Architektur des hier vorgestellten Komponentenframeworks Tesla ermöglichen es, derartige Unklarheiten oder nicht-eindeutige Evaluationsmetriken zu vermeiden (wie in Kapitel 5 demonstriert wird). Zudem bietet Tesla Möglichkeiten an, die eingangs beschriebenen Probleme im wissenschaftlichen Austausch in der Computerlinguistik zu lösen und bspw. Verfahren oder Verfahrenskombinationen zu variieren und in neuen Bereichen anzuwenden, so dass Tesla auch als ein Architekturvorschlag interpretiert werden kann, durch den die Forschung in der Computerlinguistik vereinfacht und beschleunigt wird.

Die strukturalistische Methodik dient in dieser Arbeit hingegen in erster Linie als Anwendungsbeispiel, um zu demonstrieren, dass Tesla auch komplexen Anforderungen an ein computerlinguistisches Labor genügt, und dass die in diesem Zusammenhang entwickelten Konzepte die Wiederverwertbarkeit und Austauschbarkeit von Komponenten im Vergleich zu anderen Systemen verbessern können.

Während in Hermes (2011) der Frage nachgegangen wird, wie sich Anforderungen unterschiedlicher Bereiche der Textprozessierung in Form eines gemeinsamen virtuellen Arbeitsplatzes umsetzen lassen, liegt der Schwerpunkt dieser Arbeit darauf, zu untersuchen, welche technischen Anforderungen ein linguistisch motiviertes Komponentensystem erfüllen muss, um auch für komplexe computerlinguistische Experimente einsetzbar zu sein und um empirische Forschung und Evaluation zu ermöglichen.

2 Alignment

> Whatever the future may bring,
> the universal application of these
> great principles is fully assured,
> though it may be long in coming.
>
> *(Nikola Tesla)*

In diesem Kapitel wird die grundsätzliche Funktionsweise von Alignment-basierten Verfahren zur Syntaxanalyse vorgestellt. Diese werden meist als motiviert durch die Arbeiten von Zellig S. Harris präsentiert (vgl. bspw. die unterschiedlichen Ansätze in Monson 2004, Solan *et al.* 2003b und van Zaanen 2002), weshalb zunächst in Kapitel 2.1 der notwendige theoretische Hintergrund erläutert wird. Auch wenn keiner der genannten Alignment-Ansätze die Methodik von Harris auch nur annähernd vollständig umsetzen kann, wird dies die Anforderungen an eine Komponente innerhalb eines Frameworks zur computerlinguistischen Textverarbeitung ebenso wie insbesondere auch an das Framework selber verdeutlichen, denn dessen Architektur sollte flexibel genug sein, um beliebige Erweiterungen von Komponenten zuzulassen, selbst wenn diese in der Realisierung einer umfassenden Sprachtheorie wie der transformationellen Analyse (vgl. Abschnitt 2.1.2) münden würden.

In Kapitel 2.2 wird anschließend exemplarisch das Alignment-Toolkit ABL von Menno van Zaanen vorgestellt, um in 2.3 die Möglichkeiten, Einschränkungen und potentiellen Verbesserungen des Ansatzes in Bezug auf strukturalistische Analysen im Sinne Harris' ebenso wie auf rein pragmatische Anwendbarkeit zu diskutieren, sowie die Einbettung in ein Komponenten-Framework zu skizzieren.

2.1 Strukturalistische Sprachwissenschaft

Die strukturelle Linguistik entstand Anfang des 20. Jahrhunderts, maßgeblich durch die Arbeiten Ferdinand de Saussures vorangetrieben.

Saussure unterschied zwischen *Sprache* und *Sprechen* und stellte fest, dass eine Sprache nur durch ihre Verwendung innerhalb einer Sprechergemeinschaft definiert wird, die sie jedoch gleichzeitig dabei verändert, so dass eine Sprache nicht universal, sondern nur bezüglich zeitlicher und räumlicher Grenzen beschrieben oder untersucht werden kann

(vgl. de Saussure 1931, S. 91ff). Saussure differenzierte zudem zwischen Signifikant und Signifikat, d.h. Ausdruck und Inhalt eines sprachlichen Zeichens, und postulierte, dass ein Zeichen *in sich nicht einen Namen und eine Sache, sondern eine Vorstellung und ein Lautbild* (de Saussure, 1931, S. 77) vereinigt, sowie dass sich unterschiedliche Bedeutungen von Zeichen ausschließlich durch unterschiedliche, aber gesellschaftlich etablierte Verwendung dieser Zeichen ergeben (vgl. auch de Saussure 1931, S. 116f). Jungen & Lohnstein (2006, S. 78) fassen diese Sicht wie folgt zusammen:

> Saussure zufolge ist die Sprache ein synchrones System von Einzelteilen, die durch ihre syntagmatischen und paradigmatischen Wertigkeiten bestimmt sind und zu allen anderen Elementen in Opposition stehen. Der Wert eines Einzelteils ergibt sich nicht allein aus seinen Eigenschaften, sondern bestimmt sich nur aufgrund seines Verschiedenseins, seiner Differenz, in den Beziehungen zu allen anderen Elementen des Systems.

Nach Lyons (1971, S. 160) war das Ziel der strukturellen Linguistik, „[...] eine Technik oder ein Verfahren zu entwickeln, das auf ein Korpus von belegten Äußerungen angewendet werden könnte und das es [...] erlauben würde, die Regeln der Grammatik mit Sicherheit aus dem Korpus selbst abzuleiten." Dies muss zunächst nicht bedeuten, dass die Ableitung der Grammatik algorithmisch zu erfolgen hat, sondern vielmehr, dass sämtliches Wissen, das zur Ableitung einer Grammatik benötigt wird, implizit in den Texten des Korpus enthalten ist und ohne Rückgriff auf externe Ressourcen (wie Lexika) extrahiert werden kann. Harris sah dies jedoch tatsächlich als Ziel an und versuchte, Sprache als ein mathematisches, idealiter (turing-)berechenbares System zu beschreiben (vgl. Harris 1968). Während er sich in seinem Spätwerk zunehmend mit abstrakteren bzw. komplexeren Sprachkonzepten wie der Operatorengrammatik (u.a. Harris 1976) beschäftigte, beziehen sich seine früheren Arbeiten stärker auf die Überlegungen de Saussures, insbesondere aber auf die Überlegung, inwieweit sich grammatikalisches Wissen unmittelbar aus der Analyse von Texten ergeben kann. Da im Allgemeinen auf genau diese Methoden verwiesen wird, wenn der theoretische Hintergrund eines Alignment-Verfahrens begründet wird, sollen zwei dieser Analysemethoden im Folgenden kurz vorgestellt werden.

2.1.1 Diskursanalyse

Zellig Harris entwickelte auf Basis der Überlegungen von de Saussure verschiedene Theorien, die das System Sprache auf verschiedenen Ebenen (wie Phonologie, Morphologie oder

Syntax) formal abbilden sollen. Dabei spielt die Beobachtung, dass ein sprachliches Zeichen nicht innerhalb beliebiger Kontexte vorkommen kann, sondern kontextabhängigen Einschränkungen unterliegt, eine wesentliche Rolle. Plötz (1972, S. 2) fasst Vorgehensweise und Ziel von Harris' *Discourse Analysis*, die auf dieser Beobachtung basiert, wie folgt zusammen:

> Discourse Analysis ist eine Methode, zusammenhängende Texte zu untersuchen mit dem Ziel einer Beschreibung des relativen Vorkommens aller Elemente innerhalb eines Textes. Das wird erreicht, indem man Klassen all der Elemente bildet, die in dem Text identische oder äquivalente Umgebungen haben. Jeder Satz wird dann als Folge solcher Klassensysteme repräsentiert. Die Struktur, die dem Text auf diese Weise zugeteilt wird, läßt sich dann auf semantische Information hin untersuchen.

Harris' Diskursanalyse hatte also nicht den Anspruch, zu einer vollständigen Beschreibung einer Sprache zu führen, sondern versuchte lediglich, entsprechend de Saussures Konzept von Inhalt und Ausdruck, die Sprachverwendung in den untersuchten Texten zu klassifizieren. Da jedoch, wie Harris (1957, S. 285) festhält, nahezu jedes Element von einer einzigartigen Kookurrenzmenge umgeben ist und es zudem nahezu unmöglich ist, sämtliche Kookurrenzen eines solchen Elementes aufzulisten[4], spezifiziert er die Definition einer Klasse genauer, indem er *Konstruktionen* (d.h. Abfolgen von nonterminalen Symbolen, wie *A N* in der Syntax) mit einbezieht:

> For classes K, L in a construction c, the K-co-occurrence of a particular member L_i of L is the set of members on K which occur with L_i in c: For example, in the AN construction found in English grammar, the A-co-occurrence of *hopes* (as N) includes *slight* (*slight hopes of peace*) but probably not *green*. The K-co-occurrence of L_i is not necessarily the same in two different KL constructions: the N-co-occurrents of *man* (as N_i) in N_i *is a* N may include *organism*, *beast*, *development*, *searcher*, while the N-co-occurrents of *man* in N_i*'s* N may include *hopes*, *development*, *imagination*, etc.

Durch Diskursanalyse werden nicht nur Vorkommen individueller Klassen in ihren Kontexten betrachtet, sondern es ist im Idealfall auch möglich, syntaktische Rückschlüsse über diese Klassen oder ihre Kontexte zu ziehen. So lässt sich beispielsweise beobachten,

[4] Während dies innerhalb der Phonologie noch machbar erscheint, ist ein solches Vorgehen spätestens in der Syntax nicht mehr durchführbar, da das Inventar unterschiedlicher Symbole zu groß ist, um sämtliche korrekten Kombinationen zu ermitteln.

dass Klassen unter Umständen substituierbar sein können – dass, wie Beispiel 2.1 zeigt, die Konstruktion $A\ N$ in einem Kontext vorkommen kann, in dem ebenfalls N vorkommt, nicht jedoch A.

2.1 a. Tesla developed the first radio.
b. Tesla developed the radio.
c. *Tesla developed the first.[5]

Daraus, so Harris, lässt sich schließen, dass es sich bei A um ein von N abhängiges Element handeln muss. Darüber hinaus kann beobachtet werden, dass es eine Korrelation zwischen Klassen der Typen N und V gibt, die zwischen A und V nicht existiert: Werden Nomen bezüglich ihrer Verb-Kookurrenzen analysiert, zeigt sich, dass die Menge möglicher Verben von Adjektiven, die den Nomen vorangestellt sind, nicht beeinflusst wird. Umgekehrt betrachtet unterliegen Verb-Kookurrenzen von Adjektiven daher keiner Regelmäßigkeit (vgl. Harris 1957, S. 287). Festzuhalten ist, dass diese einfachste Form der Analyse grundsätzlich keine lexikalischen Informationen benötigt, d.h. dass es unerheblich ist, ob die Information darüber, dass es sich bei *radio* um ein Nomen und bei *first* um ein Adjektiv handelt, vorhanden ist oder nicht. Genügend ähnliche Beispiele innerhalb der untersuchten Korpora vorausgesetzt[6], ließe sich anhand der von Harris beschriebenen Beobachtungen eine automatische Detektion von Dependenz für beliebige Wortarten umsetzen[7].

2.1.2 Transformationelle Analyse

Auf Basis dieser Erkenntnisse (die grundsätzlich nicht nur auf Syntax, sondern auch auf Morphologie oder Phonologie angewendet werden können) entwickelt Harris einen als *Transformationelle Analyse* bezeichneten Grammatikformalismus, in dem er vorschlägt, Transformationsregeln zu definieren, die es ermöglichen, Konstruktionen ineinander zu überführen, wie im Fall der Symbolfolgen $N\ v\ V\ N$ (v steht hier für die Klasse der Auxiliarverben) und *N's Ving N* (*Tesla will beat Edison* und *Tesla's beating Edison*). Grundidee der Transformationstheorie ist, dass aus einer kleinen, finiten Menge von Kernsatzformen

[5] Ungrammatische Sätze werden hier und im Folgenden durch ein vorangehendes Stern-Symbol (*) markiert.

[6] Tatsächlich kann dies i.d.R. nicht vorausgesetzt, sondern gegenteilig als Kritik an dem beschriebenen Ansatz verwendet werden (vgl. dazu auch Abschnitt 2.1.3).

[7] Ein Beleg für diese Vermutung wurde bisher allerdings nicht erbracht – daher ist unklar, ob die Methodik wirklich ausreichend wäre, bzw. in welchen natürlichen Sprachen sie erfolgreich angewandt werden könnte.

mittels Substitution von Kernvokabular eine finite Menge von Kernsätzen abgeleitet werden kann, anhand derer durch Anwendung von Transformationsregeln schließlich jeder beliebige Satz einer Sprache erklärt werden kann (vgl. Harris 1965, S. 382ff). In Harris (1959) wird dies zusammengefasst wie folgt:

> The result of the transformational method is families of equivalent sentences or sentential structures (or of sentences equivalent to sequences of other sentences). [... The] criterion for saying that two sentences or sentential structures A, B are transforms of each other is that the extension of A (i.e., the list of n-tuples of words which actually occur in the n classes of A) should be approximately identical with the extension of B.

Anders als im einfachsten Fall der Diskursanalyse, der nur den unmittelbaren sprachlichen Kontext berücksichtigt, wird hier lexikalisches Wissen benötigt, da die Regeln ohne zusätzliche morphosyntaktische Informationen nicht angewendet werden können.

Harris (1968, S. 65ff) definiert sieben Familien elementarer Basisoperatoren ϕ, die die Syntax der englischen Sprache beschreiben können sollen. Durch (mehrfache, auch rekursive[8]) Anwendung dieser Operatoren sollen, so Harris, sämtliche Transformationen eines Basissatzes beschrieben werden können. Er vermutet zudem, dass dies grundsätzlich auch für alle weiteren Sprachen gelte, wobei die Menge der Basisoperatoren für jede Sprache individuell sei.

1. ϕ_a bezeichnet Adjunktionen der Form *word* $\rightarrow$ *expanded word*, bspw. *groß* $\rightarrow$ *sehr groß* oder *läuft* $\rightarrow$ *läuft schnell.*

2. Mit ϕ_s werden Satzoperatoren zusammengefasst, die einen Satz um zusätzliche Elemente erweitern: *Edison invented* $\rightarrow$ *I wonder whether Edison invented; That Edison invented surprised me; Edison's inventing surprised me.*

3. ϕ_v repräsentiert Verboperatoren, die nicht durch Regeln aus ϕ_s abgeleitet werden können, und die bspw. den Satz *He studies.* in Formen wie *He is studious, He is studying, He is a student* oder *He has studied* transformieren.

4. Verbindungen einzelner Sätze werden durch den Operator ϕ_c ermöglicht, wie *Nikola kam. Thomas ging.* $\rightarrow$ *Nikola kam als Thomas ging. Nikola kam und Thomas ging. Nikola kam, nachdem Thomas ging.*

[8] Rekursion ist der „Bezug einer Datenstruktur oder einer Funktion innerhalb ihrer Definition auf sich selbst“ Hermes (2011, Kapitel 6.3.4.), vgl. ebd. für ein Beispiel der wechselseitigen, trotzgleich endlosen Rekursion.

5. Als ϕ_p bezeichnet Harris Permutationen wie *Der Wissenschaftler liest das Buch.* → *Das Buch liest der Wissenschaftler.*

6. ϕ_z beschreibt die Tilgung abhängiger Elemente, bspw. in *Er kam, er sah und er siegte.* → *Er kam, sah und siegte.*

7. Der Operator ϕ_m repräsentiert schließlich morpho-phonemische Transformationen, wie *Sie ist schön.* → *Sie war schön.*

Nach Harris genügen diese Operatoren, um (mit Ausnahme einiger Sonderfälle) alle Transformationen des Englischen abzudecken – allerdings werden keine Belege für diese Hypothese erbracht.

Zusätzlich zu dem Problem, dass Transformationen nicht immer bijektiv sind (also nicht immer in beide Richtungen überführbar sind, wie es in obigem Beispiel der Fall ist), weist Harris darauf hin, dass die Größe (bzw. Breite) des zu berücksichtigenden Kontexts für eine Transformation nicht ohne weiteres bestimmt werden kann: So scheinen bspw. die Konstruktionen *N Ved* (*Tesla worked*) und *N will V* (*Tesla will work*) ineinander transformiert werden zu können, was jedoch in einem größeren Kontext (*Tesla worked yesterday* und *Tesla will work tomorrow*) nicht mehr möglich ist, da sich daraus widersprüchliche Konstruktionen ergeben würden (**Tesla worked tomorrow* bzw. **Tesla will work yesterday*). Harris schlägt daher vor, den Kookurrenzkontext auf die größten grammatischen Einheiten auszudehnen, in der beide Konstruktionen vorhanden sind (vgl. bspw. Harris 1957, S. 289).

Während eine algorithmische Diskursanalyse in den linguistischen Teildisziplinen der Phonologie und der Morphologie (zumindest in schwach flektierenden Sprachen wie dem Englischen) noch realistisch erscheint, da die Anzahl unterschiedlicher sprachlicher Zeichen dort relativ gering ist, ist davon auszugehen, dass eine Anwendung eines solchen Algorithmus auf Syntax im Sinne von Harris scheitern muss, da nicht zu erwarten ist, dass sich genügend Transformationen eines Satzes finden lassen, um aus diesen korrekte Regeln zu generieren (siehe dazu auch Abschnitt 2.1.3).

Harris schlägt daher verschiedene Verfahren vor, mit denen die Kookurrenzen zweier Konstruktionen analysiert werden sollen, um so zu prüfen, ob diese Konstruktionen Transformationen sind oder nicht. Um zu testen, ob es sich bei N_1 *v* *V* N_2 und N_2 *v be Ven by* N_1 um eine gültige Transformation handelt, lässt sich eine konkrete Instanz der ersten Konstruktion erzeugen, in der ein Element weggelassen wird, wie in *The cat drank ø.* Anschließend wird überprüft, ob die Elemente, die an der Leerstelle eingefügt werden können, auch an der entsprechenden Stelle in *ø was drunk by the cat* stehen können. Aus der

Ähnlichkeit der so generierten Listen lässt sich anschließend über die Transformierbarkeit der getesteten Konstruktionen urteilen.

Harris (1957, S. 292) weist jedoch darauf hin, dass sich dieses Verfahren nur bedingt automatisieren lässt, da ein „impracticably large body of texts or other material“ benötigt würde. Alternativ schlägt er vor, das Urteil über die generierten Konstruktionen einem muttersprachlichen Informanten zu überlassen. Durch Einsatz eines Informanten wird jedoch nicht nur eine zusätzliche Fehlerquelle mit einbezogen, es muss zudem auch davon ausgegangen werden, dass der Informant nicht jede generierte Konstruktion als eindeutig korrekt oder falsch bezeichnen wird, sondern nicht-binäre Aussagen über die *Akzeptierbarkeit* machen wird, was die Kategorisierung von Konstruktionen erschwert. Weiterhin führen die Größe des verwendeten Vokabulars und die Länge natürlichsprachlicher Sätze unter Umständen dazu, dass eine manuelle Bewertung von Konstruktionen nicht umsetzbar ist. In Harris *et al.* (1989) beschränkten die Autoren die Anwendung der transformationellen Analyse daher auf wissenschaftliche Texte aus der Immunologie, wobei sie feststellten, dass die Anzahl von Wortklassen[9] ebenso wie die Anzahl von Elementen pro Klasse sowie die Art unterschiedlicher Satztypen in dieser als *Sublanguage* bezeichneten Sprachdomäne relativ gering war (vgl. Harris *et al.* 1989, Kapitel 1).

Transformationelle Analyse führt zwar zu einer Beschreibung von Sprache, die sowohl syntaktische als auch semantische Aspekte berücksichtigt, jedoch ist eine semantische Klassifikation oder Strukturierung von Bedeutung aufgrund der Kontextabhängigkeit des Verfahrens nur beschränkt möglich, insbesondere dann, wenn die Auswahl der analysierten Texte auf eine Teilsprache reduziert wird. So weist Harris (1988, S. 62) beispielsweise darauf hin, dass die Verben *divide* und *multiply* u. U. synonym verwendet werden können, was der allgemeinen, mathematischen Verwendung widerspricht: „The operator divide has virtually the same meaning as the operator multiply when its argument is a cell name: for a cell, to divide is to multiply“.

Zwar kann dies aus pragmatischer Sicht hilfreich sein, wie die in Abschnitt 2.3 vorgestellten Arbeiten am *Medical Language Processor* zeigen, doch ist dies hinderlich, wenn eine Sprache als Ganzes analysiert oder beschrieben werden soll.

Die hier beschriebenen Ansätze der strukturalistischen Sprachwissenschaft wurden nicht ausschließlich positiv aufgenommen – auch wenn einige der Beobachtungen vergleichsweise einfach und mit geringem Ressourcenbedarf algorithmisch umgesetzt werden können, sind

[9] Der Begriff *Wortklasse* wird im Kontext von angewandter transformationeller Analyse deutlich anders verwendet als in den bisher aufgeführten Beispielen und umfasst sowohl morphosyntaktische als auch semantische Aspekte (siehe auch Seite 27).

aus linguistischer Perspektive nicht alle Aspekte einer natürlichen Sprache analysierbar, wie im folgenden Abschnitt gezeigt wird.

2.1.3 Kritik und Anwendung

Harris' Transformationstheorie wurde in verschiedenen Punkten kritisiert – sowohl aus linguistischer als auch aus technischer Perspektive. Zur linguistischen Kritik gehört beispielsweise, dass die syntaktische Ambiguität von Sätzen wie

2.2 flying planes can be dangerous

weder erklärt noch aufgelöst werden kann: *flying planes* kann sowohl als $[\text{A N}]_{NP}$ wie auch als $[\text{Ving N}]_{NP}$, also als Gerundivum interpretiert werden, entsprechend sind entweder fliegende Flugzeuge gefährlich, oder aber das Fliegen von Flugzeugen. Harris (1959, S. 290) argumentiert, dass dies durch die Transformationelle Analyse möglich ist: „Here flying planes comes from two transformational sources, from two different kernels (*Planes fly; Someone flies planes*)." Es wäre jedoch wünschenswert, dass eine Grammatiktheorie die unterschiedlichen Lesarten eines solchen Satzes auch strukturell unterschiedlich repräsentieren kann, oder dass auch semantische Ambiguität, d.h. Ambiguität bei syntaktisch identischer Struktur, berücksichtigt wird. Während für den praktischen Einsatz implementierte Systeme (wie beispielsweise der Medical Language Processor, vgl. Seite 26) lexikalische Ambiguität und die Beobachtung, dass ein Wort mehreren Klassen angehören kann, berücksichtigen (müssen), wird dieser Umstand theoretisch nur unzureichend behandelt (vgl. auch van Zaanen 2002, S. 18ff).

Ähnliches gilt für die Beispiele in 2.3, die zwar hinsichtlich der Sequenz ihrer Wortarten identisch sind, in denen das Nomen jedoch im ersten Fall ein Objekt, im zweiten hingegen ein Subjekt ist. Dieser syntaktische Unterschied wird in einer transformationellen Analyse jedoch nicht strukturell berücksichtigt (vgl. auch Searle 1972).

2.3 a. John is easy to please.
N v A to V

b. John is eager to please.
N v A to V

Unter technischen Aspekten lässt sich an der transformationellen Analyse vor allem kritisieren, dass das *Sparse Data Problem* ungenügend berücksichtigt wird. Im Kontext von maschineller Sprachverarbeitung wird mit diesem Begriff der Umstand bezeichnet, dass ein Korpus (je nach Anwendungsfall) nicht sämtliche Merkmale enthält, die für eine voll-

ständige Analyse notwendig wären, wie Manning & Schütze (1999, S. 198f) festhalten: „While there are a limited number of frequent events in language, there is a seemingly never ending tail to the probability distribution of rarer and rarer events, and we can never collect enough data to get to the end of the tail.“ Searle (1972) fasst Chomskys Kritik diesbezüglich wie folgt zusammen :

> Chomsky argued that since any language contains an infinite number of sentences, any *corpus*, even if it contained as many sentences as there are in all the books of the Library of Congress, would still be trivially small. Instead of the appropriate subject matter of linguistics being a randomly or arbitrarily selected set of sentences, the proper object of study was the speaker's underlying knowledge of the language, his *linguistic competence* that enables him to produce and understand sentences he has never heard before.

Das Konzept einer angeborenen Sprachkompetenz steht unmittelbar der Auffassung von Harris gegenüber, wie Matthews (2001, S. 144f.) beschreibt:

> For Zellig Harris, [...] the 'structure of a language' was not independent, but a product of the scientific study of the patterns that can be abstracted from the sounds of speech [...], in particular, the description of a language was based simply on the analysis of formal patterns.

Auch wenn mit den inzwischen digital im WWW verfügbaren Texten eine potentielle Datenquelle genutzt werden kann, die quantitativ weit über die Bestände der *Library of Congress* hinaus geht, bleibt dieser Kritikpunkt bestehen - unabhängig davon, dass die Rechenleistung, die benötigt würde, um sämtliche über das WWW erreichbaren Texte einer Transformations- oder Diskursanalyse zu unterziehen, zumindest bei jetzigem Stand der Technik nicht gegeben ist. Zudem konnte Gold (1967) formal zeigen, dass es für die meisten Sprachfamilien nicht möglich ist, fehlerfrei eine Grammatik zu generieren, wenn der Lernprozess ausschließlich anhand positiver Beispiele durchgeführt wird (vgl. auch Abschnitte 5.2 und 5.5). Da Chomskys Kritik an der Plausibilität der Transformationsgrammatik bei hoch produktiven Systemen wie der Syntax durchaus berechtigt war, verlor der Strukturalismus (nach Harris) mit Chomsky (1957) zunehmend an Popularität. Allerdings wurde die Theorie nicht vollständig aufgegeben, sondern, wie im folgenden Abschnitt beschrieben, vielmehr auf Domänen angewandt, in denen sowohl Syntax als auch Vokabular relativ eingeschränkt waren.

Harris' Diskursanalyse und transformationelle Analyse wurden nicht nur mit dem Ziel einer mathematischen Berechenbarkeit entwickelt, sondern auch unter dem Gesichtspunkt einer konkreten algorithmischen Umsetzung: Bereits 1957 wurde mit der Implementation eines syntaktischen Parsers für den *UNIVAC I*[10] begonnen, welcher, so Sager & Nhan (2002, S. 80), erfolgreich einen kurzen, wissenschaftlichen Text analysieren konnte (wenngleich auch nur unter Verwendung eines manuell erzeugten Lexikons):

> The algorithm of the UNIVAC program incorporated the major constructions of English grammar in considerable detail. While the dictionary was small, lexically ambiguous words were multiply classified [...], with provision in the algorithm for recognizing these as potential sources of alternative analyses.

Im Rahmen des *Linguistic String Projects* (LSP)[11], an dem Harris direkt beteiligt war, wurde ab 1965 damit begonnen, einen Parser für englischsprachige, medizinische Texte zu implementieren, der auf Harris' Theorien basierte und diese mit Hilfe einer eigens entwickelten Programmiersprache (*Restriction Language*, RL, vgl. Sager & Grishman 1975) umsetzte. Aus dem LSP entstand der *Medical Language Processor* (MLP), welcher den Parser des LSP benutzt, um unstrukturiert vorliegende medizinische Fachtexte in eine strukturierte, nicht-ambige Form zu überführen, die zusätzlich um fachspezifisches Wissen ergänzt wird. Dazu wird jeder Satz zunächst bezüglich seiner Grammatik analysiert und mit syntaktischer Struktur annotiert, um in einem zweiten Schritt semantisch inkorrekte Analysen auf Basis einer domänenspezifischen Liste von Wort-Kookurrenzen zu filtern. Im dritten Schritt werden schließlich, falls der zu prozessierende Satz Konjunktionen enthält, Transformationen ausgeführt, um die Struktur des Satzes zu vereinfachen – so würde beispielsweise der Ausdruck *Schmerzen in Hals und Kniegelenk* in die Ausdrücke *Schmerzen in Hals* und *Schmerzen in Kniegelenk* transformiert (vgl. Sager *et al.* 1994, S. 148).

MLP lässt sich dabei auch an weitere Sprachen anpassen, wofür die Grammatik einer solchen Sprache zunächst in RL definiert werden muss, aus der anschließend u.a. ein kontextfreier Parser kompiliert wird. Eine Anpassung von MLP an das Niederländische wird in Spyns *et al.* (1998) beschrieben, wobei die Autoren festhalten, dass MLP selbst nicht modifiziert werden musste, wohl jedoch Teile des verwendeten Lexikons, der Auszeichnung und der Parsebaumstruktur.

[10] Dabei handelte es such um einen der ersten kommerziell verfügbaren Computer, vgl. http://de.wikipedia.org/wiki/UNIVAC_I.

[11] Siehe http://cs.nyu.edu/cs/projects/lsp/.

Medical Classes	Description	Examples in English and French
H-PT	references to patient	she, candidate, Mrs. XXX, patient
H-PTPART	body part	arm, adrenal, carotid, liver, foie
H-PTVERB	verb with patient subj	complain, endure, suffer
H-AMT	amount or degree	much, partly, total, sévère
H-NEG	negation of finding	no, not, cannot, denied, ne pas

Tabelle 2.1: Ausschnitt der im MLP verwendeten Wortklassen (aus Sager & Nhan 2002, S. 95).

Schwerpunkt der algorithmischen Umsetzung und Evaluation der Transformationstheorie war stets die Analyse domänenspezifischer *Sublanguages*, was Sager & Nhan (2002, S. 91) wie folgt begründen:

> In a science sublanguage, some types of sentences are possible while others are simply outside the subject area or are such combinations of sublanguage words as are simply not sayable within the science. To use Harris's example [...], in the language of biochemistry, contrast the possible (1) *The polypeptides were washed in hydrochloric acid*, with (2) *Hydrochloric acid was washed in polypeptides*, which if it ever occurred would not be in the discourse of biochemistry.

Die Definition von Wortklassen weicht dabei stark von schulgrammatischen oder klassischen, linguistischen Definitionen ab, um fachspezifische, semantische Eigenarten berücksichtigen zu können, wie Tabelle 2.1 veranschaulicht. Diese Kategorien wurden manuell für den MLP erstellt, da die Eigenarten medizinischer Fachtexte (etwa reduzierte Satzstrukturen wie in *Patient complaining of increased breathlessness*) eine automatische Analyse syntaktischer Relationen erschweren und da syntaktische Ambiguität die Qualität automatisch extrahierter Ergebnisse verschlechtern würde (vgl. Sager & Nhan 2002, S. 94ff).[12]

2.1.4 Aktuelle Entwicklungen

Neben LSP und MLP wurden, insbesondere innerhalb der Korpuslinguistik, weitere Verfahren entwickelt, die zumindest teilweise auf Harris' Forschung zurückgeführt werden können. Habert & Zweigenbaum (2002, S. 221) analysieren unterschiedliche Ansätze zur automatischen Textanalyse, die Harris' Theorien aufgreifen, und unterscheiden zwei grundsätzliche Arten von korpusbasierter Bedeutungsextraktion. Dabei handelt es sich zum

[12] Zwar spekulieren Sager & Nhan (2002, ebenda) darüber, dass ein Bootstrapping-Verfahren erfolgreich sein könnte, gehen jedoch nicht weiter auf diesen Ansatz ein.

einen um Ansätze, deren Ziel es ist, Assoziationen zwischen Wörtern zu ermitteln, und um Verfahren, die detailliertere linguistische Strukturen oder Wissensstrukturen erlernen sollen. Zu den assoziativen Ansätzen gehören beispielsweise die bereits in Kapitel 1 erwähnten Vektor- und Clusteringmodelle, die Wörter auf Basis ihrer Verwendungskontexte auf einen multidimensionalen Raum abbilden, wie das *Hyperspace Analogue to Language*-Modell (vgl. Lund & Burgess 1996, siehe auch Kapitel 5.5.3 dieser Arbeit) oder die *Latent Semantic Analysis* (vgl. Deerwester *et al.* 1990). Diese Ansätze verzichten häufig auf eine genauere syntaktische Analyse der zu verarbeitenden Texte und verwenden lediglich Wortfenster unterschiedlicher Breite, um die Kookurrenzen der einzelnen Wörter zu ermitteln (vgl. Manning & Schütze 1999, Kapitel 8.5). Vorteilhaft an dieser Vorgehensweise ist die Robustheit und (relative) Sprachunabhängigkeit der Algorithmen, da lediglich eine Tokenisierung der Texte stattfinden muss, weitergehende syntaktische Analysen jedoch nicht grundsätzlich notwendig sind.[13] Allerdings existieren Ansätze, vorhandene strukturelle Informationen in die Generierung der Assoziationen mit einfließen zu lassen, um die Qualität der Ergebnisse zu verbessern. Padó & Lapata (2007) geben hier einen Überblick über solche Ansätze und stellen gleichzeitig ein Framework vor, mit dem syntaktische Informationen für die Extraktion von Wortbedeutungen genutzt werden können. Sowohl Habert & Zweigenbaum (2002, S. 225) als auch Padó & Lapata (2007, S. 166) merken jedoch an, dass die Berücksichtigung struktureller Informationen nicht notwendigerweise zu besseren Ergebnissen führt – eine weitere Auseinandersetzung mit diesem Themengebiet scheint also sinnvoll.

Im folgenden Abschnitt wird daher ein Verfahren vorgestellt, das grundsätzlich für die Detektierung syntaktischer Strukturen geeignet ist, und das, da es sich um einen relativ allgemein gehaltenen, erweiterbaren Ansatz handelt, als Ausgangspunkt für die in den folgenden Kapiteln vorgestellten Überlegungen zur Entwicklung eines Frameworks für strukturalistisch motivierte Textanalyse verwendet werden wird.

2.2 Alignment Based Learning

Das in van Zaanen (2000b) vorgestellte Framework *Alignment Based Learning* (ABL) war einer der ersten Versuche, natürliche Sprache ausschließlich mit rein distributiven Methoden und ohne Zugriff auf externe Ressourcen zu strukturieren. Da somit ein auf der transformationellen Analyse basierendes Verfahren nicht verwendet werden kann, bleibt nur der Bezug auf die Diskursanalyse, der jedoch ebenfalls abgeschwächt werden muss:

[13] Zumindest dann, wenn es sich nicht um eine morphologisch reiche Sprache handelt.

Hypothese von van Zaanen (2000a) ist, dass Konstituenten, die im gleichen Kontext auftreten, vermutlich zur gleichen *funktionalen*[14] Kategorie gehören, so dass der Austausch dieser Konstituenten zu neuen, syntaktisch korrekten Sätzen führt:

> [For] example in the sentence *What is a dual carrier* the noun phrase *a dual carrier* may be replaced by another noun phrase. Replacing the noun phrase with *the payload of an African Swallow* yields the sentence *What is the payload of an African Swallow*, which is another well-formed sentence.

Während Harris die Anforderung stellte, dass die Elemente einer Kategorie in nahezu allen Kontexten austauschbar sein müssen (vgl. Abschnitt 2.1.1), genügt es ABL, lediglich einen gemeinsamen Kontext zu finden, um zwei Konstituenten[15] der gleichen Kategorie zuzuordnen. Auf Basis von Abweichungen zwischen ähnlichen Sätzen[16] generiert ABL im ersten, als *Align* bezeichneten Schritt Hypothesen über die Konstituentengrenzen, die anschließend zur Detektion syntaktischer Struktur verwendet werden. Die generierten Hypothesen sind zunächst lokal und ausschließlich auf die beiden Sätze, aus denen sie extrahiert wurden, bezogen, so dass grundsätzlich auch widersprüchliche Strukturen erzeugt werden, wie Beispiel 2.4 aus van Zaanen (2000b, S.238) zeigt:

2.4 a. [Book Delta 128]$_{X1}$ from Dallas to Bosten
b. [Give me all flights]$_{X1}$ from Dallas to Boston

c. Give me [all flights from Dallas to Boston]$_{X2}$
d. Give me [information on reservations]$_{X2}$

ABL weist jeder Struktur, die extrahiert wurde, zunächst eine eindeutige Kategorie zu, so dass zu jeder Kategorie genau zwei Konstituenten gehören. In einer zweiten, als *Clustering* bezeichneten Phase des Algorithmus werden die Kategorien von Konstituenten unifiziert, falls ihre Kontexte identisch waren[17], und schließlich werden in der *Selection*-Phase Über-

[14] Diese Bezeichnung führt van Zaanen ein, um deutlich zu machen, dass es sich nicht um grammatische Kategorien handeln muss, sondern dass bspw. auch Adjektiv und Präpositionalphrase derselben Kategorie angehören können.

[15] Der Begriff *Konstituente* ist bezüglich der folgenden Beispiele etwas unglücklich gewählt, da es sich aus sprachwissenschaftlicher Sicht nicht um Konstituenten handelt, sondern lediglich um Wortsequenzen. Da diese in den hier zitierten Veröffentlichungen jedoch stets als *Constituents* bezeichnet werden, wurde er dennoch verwendet.

[16] Die genaue Funktionsweise der verwendeten Algorithmen wird Abschnitt 2.2.1 und Kapitel 5 beschrieben.

[17] Der Begriff *Clustering* darf hier nicht mit vektorbasiertem Clustering verwechselt werden: Mit Ausnahme davon, dass beide Verfahren dazu genutzt werden können, Elemente zu gruppieren, finden sich keine weiteren Gemeinsamkeiten.

lappungen wie in 2.4 aufgelöst, indem eine der widersprüchlichen Hypothesen verworfen wird (vgl. van Zaanen 2002, S. 31ff). Ineinander eingebettete, u.U. rekursive Hypothesen werden hingegen übernommen, wie 2.5 aus van Zaanen (2002, S. 78) veranschaulicht: Hier wurde die Struktur *first leg or the second leg* der gleichen Kategorie zugeordnet wie ihr Bestandteil *second leg*.

2.5 Is dinner served on the [first leg or the [second leg]$_1$]$_1$

Dadurch, dass Hypothesen auf Basis lokaler Ähnlichkeit generiert werden, und dadurch, dass van Zaanen größtenteils *Sublanguage*-Korpora verwendet[18], kann das *Sparse Data Problem* (vgl. S. 24) aus technischer Sicht reduziert werden. Dies macht das Verfahren vergleichsweise produktiv, insbesondere dann, wenn bereits ein kleiner gemeinsamer Kontext zur Generierung von Hypothesen führt. Allerdings führt die im Vergleich zu Harris deutlich abgeschwächte Definition identischer Kontexte auch dazu, dass die Elemente einer Kategorie i.d.R. weder semantisch verwandt noch grammatikalisch einheitlich sind, sondern lediglich auf Basis der sequentiellen Position im Satz zurückzuführen sind, wie Beispiel 2.6 aus van Zaanen (2003) veranschaulicht: Die Wörter *morning* und *nonstop* haben in ihrer Bedeutung keine Gemeinsamkeiten, zudem handelt es sich bei ersterem um ein Nomen und bei letzterem um ein Adjektiv.

2.6 a. Show me the [morning]$_X$ flights.
b. Show me the [nonstop]$_X$ flights.

Auffällig ist die Ähnlichkeit des Beispiels zu dem von Chomsky als Kritik am Strukturalismus vorgebrachten Beispiel 2.3 auf Seite 24 (*John is eager/easy to please*) – sie zeigt, dass die Kritik nicht nur theoretischer Natur, sondern auch praktisch relevant ist. Durch Einführung eines Konstrukts wie der funktionalen Kategorie[19] kann dieser Kritikpunkt zwar konzeptionell umgangen werden, allerdings bedeutet dies zwangsläufig, dass die Möglichkeit einer eindeutigen Abbildung von funktionalen auf grammatische Kategorien (oder umgekehrt) verloren geht. Es ist jedoch auch nicht zu erwarten, dass ein derartiger Ansatz, der ohne eine Form von Wissen über eine Sprache auskommt, dies leisten könnte, wie schon de Saussure (1931, S. 123) festhielt[20]:

[18] Wie das *Air Travel Information System* (ATIS) Korpus, welches ausschließlich Sätze enthält, die im Kontext von Reiseplanungen geäußert wurden (vgl. Hemphill *et al.* 1990).

[19] Vergleichbare Konzepte werden auch in anderen Alignment-Ansätzen benötigt, auch wenn sie dort anders bezeichnet werden, etwa als „paradigmatische Relationen“ (Schwiebert 2005) oder „structural regularities“ (Solan *et al.* 2003a).

[20] Saussure bezieht sich auf gesprochene Sprache und nicht auf maschinell verarbeitete Texte. Allerdings ist dies keine Einschränkung: Wie de Saussure (1931, S. 28) feststellt, handelt es sich bei Sprache und

> Im ersten Augenblick ist man versucht, die sprachlichen Zeichen mit sichtbaren Zeichen zu vergleichen, welche im Raum nebeneinander bestehen können, ohne sich zu vermischen; und man bildet sich ein, daß die Abtrennung der bedeutungsvollen Elemente auf die gleiche Weise vorgenommen werden könne, ohne daß irgendeine geistige Tätigkeit dabei nötig sei. [...] Aber bekanntlich ist die Haupteigenschaft der gesprochenen Kette, daß sie linear ist [...]. Wenn wir eine unbekannte Sprache hören, sind wir nicht imstande, zu sagen, wie die Folge der Laute analysiert werden müsse; das kommt daher, daß diese Analyse nicht möglich ist, wenn man nur die lautliche Seite der Sprache berücksichtigt. Wenn wir aber wissen, welchen Sinn und welche Rolle man jedem Teil dieser Sprache zuerkennen muß, dann sehen wir gewisse Teile sich voneinander ablösen und das gleichmäßig fortlaufende Band sich in Glieder abteilen; diese Analyse ist aber keineswegs materieller Natur.

Bezüglich der in Abschnitt 2.1.2 aufgeführten Basis-Operationen nach Harris lässt sich zudem festhalten, dass ein Großteil der beschriebenen Transformationen durch den hier skizzierten Algorithmus prinzipiell nicht detektiert werden kann.[21]

Trotz der aufgeführten sprachwissenschaftlichen Einwände könnte ein solcher Ansatz jedoch dafür genutzt werden, potentielle Wortklassen aus einem Korpus zu extrahieren (vgl. Abschnitt 5.5). Ebenfalls wäre zu evaluieren, ob er für strukturell einfache Transformationsregeln (bspw. zur Adjunktion optionaler Kategorien wie etwa Adjektiven) geeignet ist, oder ob er sich für pragmatische, anwendungsorientierte Aufgaben in geeigneten Domänen einsetzen lässt. Morphosyntaktische Transformationen (vgl. bspw. die Regeln zur Transformation von Sätzen (2) oder Verben (3) auf Seite 21) kann ein rein symbolisch arbeitendes Verfahren wie ABL in der hier vorgestellten Version hingegen nicht aufdecken. Im folgenden Abschnitt wird das Verfahren aus technischer Perspektive beschrieben, um die notwendigen Grundlagen für die Integration in ein Komponentenframework definieren zu können.

Schrift um verschiedene Systeme von Zeichen, die unterschiedlichen Regeln folgen. Daher ist es nicht möglich, durch Analyse von Schriftsprache die Sprache als solche vollständig zu analysieren: Dies sei, „... als ob man glaubte, um jemanden zu kennen, sei es besser, seine Photographie als sein Gesicht anzusehen“.

[21] Dies gilt auch ohne den Anspruch, dass eine semantische Äquivalenz oder Ähnlichkeit bestehen bleiben muss: Adjunktionen, Permutationen und Tilgungen (vgl. S. 21) können theoretisch durch Symbolvergleiche detektiert werden, komplexe Transformationen (wie etwa durch Satz- und Verboperatoren) können jedoch ebensowenig ermittelt werden wie satzübergreifende Zusammenhänge.

2.2.1 Algorithmen

ABL wurde in *C++* implementiert, allerdings existiert mit *ABL4J*[22] auch eine (vom Autor dieser Arbeit) nach Java portierte Variante von ABL 1.1, die hier verwendet wird. Diese Version unterstützt zwar nicht sämtliche Alignment-Verfahren der Originalfassung, dafür ist sie jedoch leicht um zusätzliche Verfahren erweiterbar (vgl. Kapitel 5 und Anhang B.5.1) und steht zudem unter einer Open Source Lizenz, so dass sie in anderen Programmen (wie den in Kapitel 3 beschriebenen Komponenten-Frameworks) verwendet werden kann. Die unterschiedlichen Algorithmen der *Align*- und *Select*-Phasen von ABL sind dabei objektorientiert umgesetzt worden und können wahlweise durch Modifikation des Quellcodes, durch Modifikation von Properties-Dateien oder durch die Übergabe von Properties beim Start der Java VM ausgewählt werden.

Insbesondere für die Align-Phase implementiert van Zaanen verschiedene Algorithmen, durch die die Generation von Hypothesen modifiziert wird. Dazu gehören mit *Left* und *Right* zwei Strategien, die zu jedem Eingabesatz der Länge n genau n Hypothesen über potentielle Konstituenten-Grenzen generieren, die sich hinsichtlich des Anfangs der Konstituente im Satz (*Left*) oder ihres Endes (*Right*) unterscheiden. Dieses naive Vorgehen (das trotz Zuordnung zur Align-Phase nicht als Alignment-Verfahren betrachtet werden kann, da die Generierung von Hypothesen hier einer auf einzelne Sequenzen anwendbaren Strategie folgt, die von den untersuchten Symbolfolgen nicht beeinflusst wird) hat den Vorteil, dass die Laufzeit des Algorithmus linear bezüglich der Menge der zu analysierenden Sätze bleibt und reduziert zudem die Anzahl der in folgenden Schritten zu analysierenden Hypothesen.[23] Als nachteilig kann jedoch betrachtet werden, dass die zur Detektion der korrekten Struktur benötigten Hypothesen u.U. gar nicht erst aufgestellt werden, wie Beispiel 2.7 anhand der durch *Right* aufgestellten Hypothesen zeigt[24].

2.7 a. [Der großartige Wissenschaftler starb.]$_0$
b. Der [großartige Wissenschaftler starb.]$_1$
c. Der großartige [Wissenschaftler starb.]$_2$
d. Der großartige Wissenschaftler [starb.]$_3$

[22] Online unter http://developer.berlios.de/projects/abl4j/

[23] Die Menge aller potentiellen Konstituenten eines Satzes der Länge n beträgt, sofern davon ausgegangen wird, dass Konstituenten stets zusammenhängend auftreten, 2^{n-1}: Eine Sequenz aus n Wörtern enthält $n-1$ Leerstellen, die entweder eine Konstituentengrenze markieren *(1)* oder nicht *(0)* – aus der Analogie zum binären Zahlensystem ergibt sich die genannte Formel.

[24] Die Nominal- bzw. Determiniererphrasen *großartige Wissenschaftler* und *Der großartige Wissenschaftler* werden im Fall von Beispiel 2.7 ansatzbedingt nicht erzeugt.

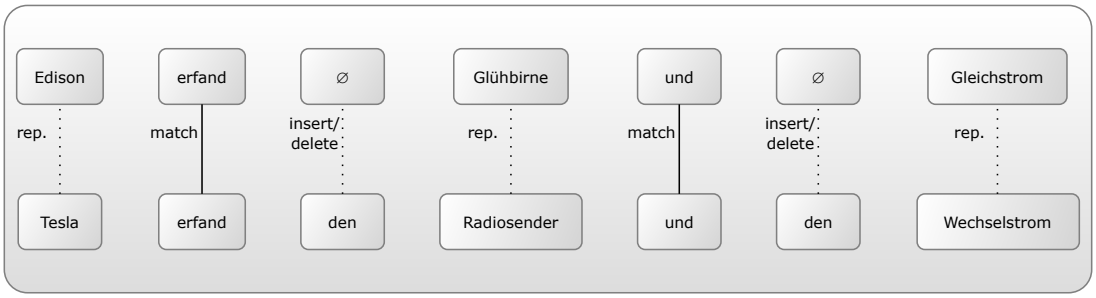

Abbildung 2.1: Berechnung der Edit-Distance zwischen zwei Sätzen. Die punktierten Linien markieren Einfügung, Tilgung und Ersetzung, während die durchgezogenen Linien Elemente verbinden, die in beiden Sequenzen vorkommen.

Zwei der von van Zaanen verwendeten Alignment-Methoden basieren auf dem Wagner-Fischer-Algorithmus, welcher die *Edit Distance* zweier Zeichensequenzen berechnet. Diese ergibt sich aus der Anzahl der Operationen *insert*, *delete* und *replace*, die notwendig sind, um eine Zeichenkette in eine andere Zeichenkette zu überführen (vgl. Wagner & Fischer 1974). Jede der drei Operationen modifiziert eine der beiden Ausgangssequenzen, indem entweder ein Zeichen eingefügt, gelöscht oder ersetzt wird – die kostenminimale[25] Menge dieser Operationen bezeichnet die Distanz beider Sequenzen. Das Verfahren lässt sich grundsätzlich auf beliebige Symbolfolgen anwenden, so etwa auf Wortsequenzen, wie in Abbildung 2.1 veranschaulicht.

Werden die notwendigen Operationen protokolliert, so kann aus der finalen Operatormenge u.a. auf die Symbole, die nicht modifiziert werden mussten, geschlossen werden – übertragen auf ABL handelt es sich bei diesen Symbolen um gemeinsame Kontexte, aus denen wiederum Hypothesen über Konstituenten generiert werden können. Wie Beispiel 2.8 anhand des Alignments der drei dargestellten Sätze zeigt, können die so erzeugten Hypothesen die Gemeinsamkeiten und Unterschiede der alignierten Sätze relativ gut abbilden.

2.8 a. [Der [großartige]$_1$]$_3$ Wissenschaftler starb [heute.]$_2$
b. [Der [mittelmäßige]$_1$]$_3$ Wissenschaftler starb [in New Jersey.]$_2$
c. [Ein großartiger]$_3$ Wissenschaftler starb [gestern.]$_2$

Nachteilig ist jedoch die Laufzeit des Verfahrens: Zum einen müssen sämtliche Satzpaare des Korpus miteinander verglichen werden (was bei n Sätzen zu einer Laufzeit

[25] Im einfachsten Fall werden jedem Operationstyp Kosten von *1* zugewiesen, so dass die Menge der Operationen den Kosten der Überführung entspricht. Dies kann jedoch modifiziert werden, so dass bspw. der Austausch eines Zeichens durch ein anderes (der auch als iterative Anwendung der Operationen *delete* und *insert* interpretiert werden kann) größere Kosten verursacht als die übrigen Operationen.

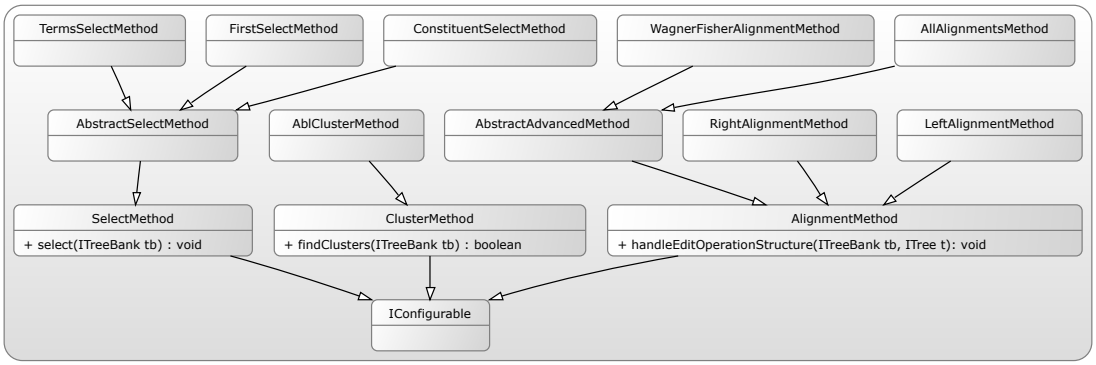

Abbildung 2.2: UML-Diagramm von *Align*, *Cluster* und *Select* in ABL4J. Die objektorientierte Umsetzung ermöglicht eine einfache Integration neuer Verfahren. Zwecks Übersichtlichkeit werden nur ausgewählte Methoden dargestellt.

von $O(n^{\frac{n+1}{2}})$ führt), zum anderen muss für jedes Satzpaar die Wagner-Fischer-Distanz berechnet werden, so dass der Algorithmus (unter der Annahme, dass jeder Satz m Wörter enthält), etwa $O(n^2m^2)$ Schritte benötigt.

Auch in der *Select*-Phase von ABL können verschiedene Algorithmen angewendet werden, wodurch das Auswahlverfahren bei überlappenden und damit widersprüchlichen Strukturhypothesen beeinflusst wird. Neben einem naiven Ansatz, in dem die Hypothese ausgewählt wird, die als erste aufgestellt wurde, sind zwei weitere Strategien implementiert, die jeder Hypothese eine Bewertung zuweisen. Im Fall der *terms*-Methode wird beispielsweise die Anzahl aller Vorkommen einer Konstituente mit der Anzahl aller Hypothesen in Relation gesetzt, so dass häufig als Konstituente markierte Wortsequenzen eine höhere Bewertung erhalten als Hypothesen über selten vorkommende Wortsequenzen.

Neben der Möglichkeit, Algorithmen individuell zu konfigurieren, indem verschiedene Parameter modifiziert werden (vgl. Anhang B.5.1 bis B.5.3), bietet sich der modulare Aufbau des Programms dafür an, sowohl neue Algorithmen zu implementieren als auch weitere Verarbeitungsschritte hinzuzufügen, welche die existierenden Module *Align*, *Cluster* und *Select* wahlweise ergänzen oder ersetzen können – etwa, um die generierten Hypothesen stärker an die im ersten Teil dieses Kapitels beschriebenen Anforderungen der Diskursanalyse zu binden, oder um aus den aufgedeckten Strukturen Transformationsregeln im Sinne der Transformationellen Analyse zu extrahieren. Nicht zuletzt aus diesem Grund wurde ABL daher als Basis für die Implementation eines Alignment-Frameworks ausgewählt, denn alternative Ansätze (oder einzelne Teile davon), wie das in Schwiebert (2005) vorgestellte System *Self Organizing Graph* könnten so in das Framework eingebunden werden, was dem in Kapitel 1 gestellten Anspruch der Vergleichbarkeit der Algorithmen entsprechen würde.

Dies setzt jedoch voraus, dass neue Komponenten entweder mit den von ABL4J verwendeten Datenstrukturen arbeiten, oder dass, falls dies nicht möglich oder nicht gewünscht ist, Kompatibilität bezüglich des Ein- und Ausgabeformats erreicht wird, oder aber dass, falls auch dieser Ansatz nicht ausreichend sein sollte, eine flexiblere Möglichkeit gefunden wird, Alignment-Verfahren zu evaluieren. Dies wird in den restlichen Abschnitten dieses Kapitels diskutiert.

2.2.2 Datenstrukturen und Dateiformat

Wie im vorherigen Abschnitt skizziert, kann es sinnvoll sein, ABL4J um weitere Algorithmen für die *Align*-, *Cluster*- und *Select*-Phase ebenso wie um zusätzliche Phasen zu ergänzen – etwa, um zunächst einen *Align*-Algorithmus zu entwickeln, der (im Gegensatz zu Beispiel 2.7) sämtliche potentiellen Konstituenten zu einem Satz generiert, die anschließend in einer zusätzlichen Phase vor oder nach der *Cluster*-Phase gefiltert werden, so dass die Menge der zu analysierenden Hypothesen möglichst gut eingeschränkt wird und effizient verarbeitet werden kann.

Grundsätzlich bieten sich dafür zwei Möglichkeiten an: Zum einen könnte das von ABL und ABL4J interpretierte Dateiformat gelesen, in eine geeignete interne Darstellung konvertiert, verarbeitet und wieder ABL-konform exportiert werden. Diese Möglichkeit ist zwar sehr flexibel, da sie bspw. nicht auf die Verwendung einer speziellen Programmiersprache beschränkt ist, jedoch ist sie u.U. nicht sonderlich effizient, da viele der notwendigen Informationen nur implizit vorliegen, wie im Folgenden anhand von Listing 2.1 erläutert wird:

```
Der groß artige Wissenschaftler starb heute. @@@(0,5,[0])(1,2,[1])(4,5,[2])(0,2,[3])
Der mittelmäß ige Wissenschaftler starb in New Jersey. @@@ (1,2,[1])(4,7,[2])(0,7,[0])(0,2,[3])
Ein groß artiger Wissenschaftler starb gestern. @@@ (0,2,[3])(4,5,[2])(0,5,[0])
```

Listing 2.1: Datenformat von ABL, dargestellt an Beispiel 2.8.

Ermittelte Konstituenten sind, durch *@@@* vom jeweiligen Satz separiert, als durch runde Klammern zusammengefasste Tripel markiert. Die ersten zwei Zahlen pro Tripel definieren dabei Anfang und Ende der Hypothese im Satz, wobei die erste Zahl die Position des ersten Wortes, das zur Konstituente gehört, definiert, während die letzte Zahl die Position des ersten Wortes, das nicht mehr zur Konstituente gehört, bezeichnet.[26] Darauf

[26] Die Nummerierung folgt dabei der in vielen Programmiersprachen üblichen Konvention, dass das erste Element einer Liste über den Index *0* referenziert wird.

folgend werden in eckigen Klammern alle Kategorien, denen die jeweilige Konstituente entspricht, aufgelistet (in obigem Beispiel ist jede Konstituente genau einer Kategorie zugeordnet worden, weshalb dort nur einelementige Listen auftauchen).

Mit Hilfe dieser Informationen kann zwar eine Datenstruktur erzeugt werden, die für den zu implementierenden Algorithmus effiziente Zugriffsmöglichkeiten bietet (wie den Zugriff auf alle Konstituenten gleicher Kategorie, was andernfalls nur mit einer linearen Suche über alle Sätze möglich wäre), allerdings erfordert dies relativ großen Entwicklungsaufwand, da zunächst die gewünschte Datenstruktur ebenso wie Parser und Generator für das proprietäre ABL-Format implementiert werden müssen. Zudem unterliegt obiges Format Einschränkungen, die einen flexiblen Umgang mit ABL verhindern: Zum einen beziehen sich Start- und Endposition einer Konstituente auf Wörter, was problematisch ist, falls andere linguistische Einheiten (bspw. Morpheme) untersucht werden sollen, oder falls abstraktere Repräsentationen der untersuchten Einheiten, wie etwa POS-Tags, aligniert werden sollen. Zum Anderen ist das Format nur bedingt erweiterbar – die Integration bspw. einer Bewertung von Konstituenten (wie sie das in Abschnitt 2.2.1 vorgestellte Wagner-Fischer-Alignment theoretisch liefern könnte; vgl. auch Abschnitt 5.3.1 für ein Verfahren, dass eine derartige Bewertung umsetzt) in das ABL-Format würde eine Modifikation der vorhandenen IO-Komponenten erfordern.

Innerhalb einer Programmiersprache kann eine derartige Erweiterung u.U. leicht umgesetzt werden – Abbildung 2.3 zeigt die von ABL4J verwendeten Datenstrukturen, bei denen es sich (mit Ausnahme der Klasse `NonTerminal`) um (Java-)Interfaces handelt, so dass Entwickler diese wahlweise reimplementieren oder bereits vorhandene Implementationen durch Vererbung erweitern und somit an die Anforderungen neuer Verfahren anpassen können. Die Methoden, die von den jeweiligen Interfaces bereitgestellt werden, bieten Zugriffsmöglichkeiten, die für die Generierung neuer Hypothesen ebenso wie für die Analyse vorhandener Daten entwickelt wurden, und erleichtern dadurch die Integration neuer Algorithmen.

Allerdings setzt diese Alternative die Verwendung der Programmiersprache Java voraus[27], zudem ist die Erweiterung der von ABL4J vorgegebenen Datenstrukturen weiterhin nur beschränkt möglich, da keine zusätzlichen Informationen gespeichert oder geladen werden können, ohne in den Serialisierungsprozess einzugreifen.

Die Definition einer konkreten Datenstruktur, die für aktuelle ebenso wie für zukünftige

[27] Zwar könnten durch den Einsatz von *Java Native Interfaces* auch andere Programmiersprachen verwendet werden, dies würde jedoch zu erheblichem Mehraufwand bei der Implementation neuer Algorithmen führen.

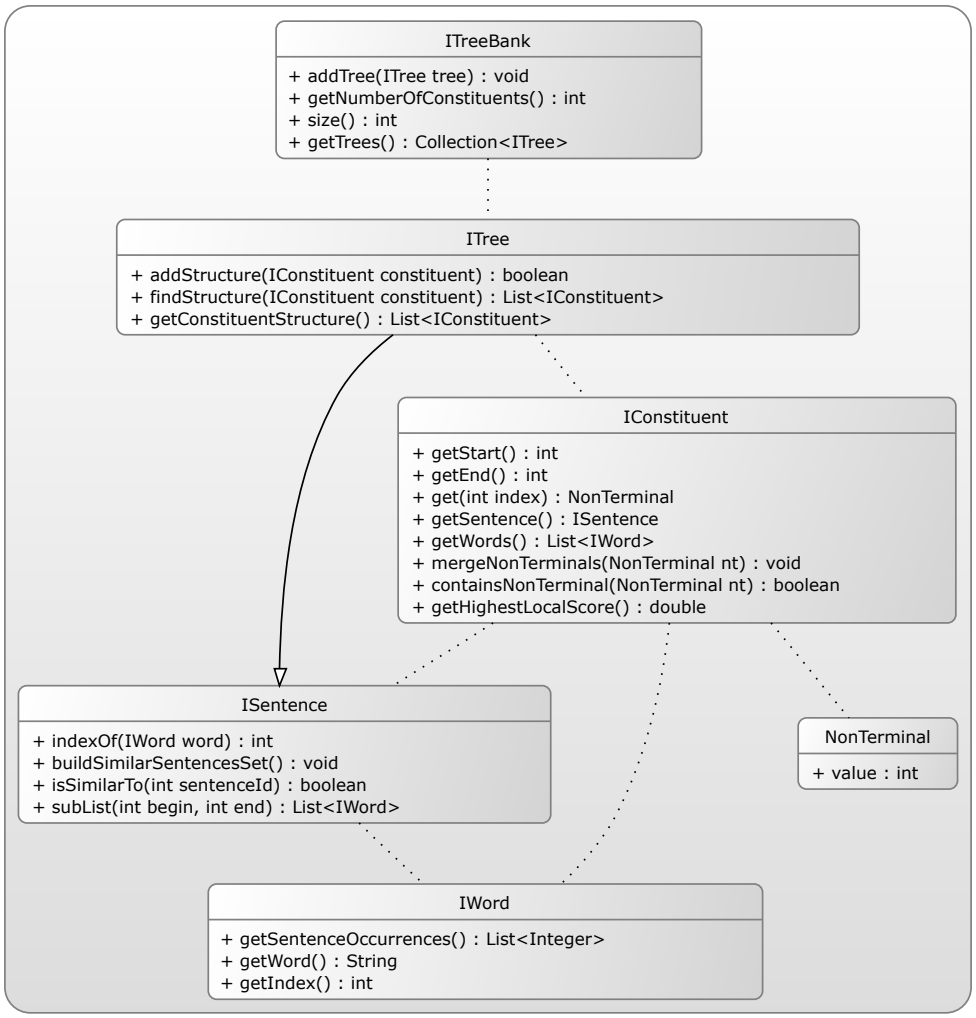

Abbildung 2.3: Vereinfachtes Klassendiagramm der von *ABL4J* verwendeten Datenstrukturen. Methoden wie `ITree.findStructure()` oder `ISentence.isSimilarTo()` verdeutlichen, dass komplexe Datenstrukturen nicht ausschließlich aus Aggregationen einfacher Datentypen bestehen müssen, sondern auch zusätzliche Funktionalität bieten können, die deren Verwendung vereinfacht (vgl. Abschnitt 2.3 ab Seite 38).

Alignment-Verfahren einsetzbar wäre, ist nicht nur aufwändig, sondern müsste zudem als Standard etabliert werden, was im Rahmen einer individuell verfassten Dissertation nicht zu leisten ist. Daraus ergibt sich, dass die Vergleichbarkeit oder Austauschbarkeit von Alignment-Verfahren nur über ein flexibles Ein- und Ausgabeformat erreicht werden kann. Dieses muss abstrakt genug sein, um von beliebigen Verfahren eingesetzt und erweitert werden zu können, und zugleich so konkret definiert werden, dass es dem Anspruch an die Vergleichbarkeit gerecht wird. Im folgenden Abschnitt soll diese Anforderung präzisiert werden.

2.3 Anforderungen an ein linguistisches Komponentensystem

Auch ohne Evaluation der Ergebnisse von ABL (hierzu sei auf Kapitel 5 verwiesen) hat die in Abschnitt 2.2.2 durchgeführte Analyse gezeigt, dass das Verfahren nicht den qualitativen Ansprüchen genügt, die zur Umsetzung der in Kapitel 2.1 skizzierten Theorien erfüllt werden müssen. Insbesondere die automatische Generierung von Transformationsregeln ist grundsätzlich nicht unproblematisch – nach Habert & Zweigenbaum (2002, S. 218) werden Transformationsregeln in aktuellen Ansätzen, wenn überhaupt, lediglich manuell erstellt und, wie im Fall des MLP (vgl. Seite 26), für das Parsen von Texten verwendet. Verfahren zur automatischen Strukturerkennung lassen sich nur eingeschränkt für die Generierung von Transformationen einsetzen, da i.d.R. lediglich Teilstrukturen aufgedeckt werden.

Grundsätzliche Aussagen über das Potential von automatischen Strukturaufdeckungsverfahren sind jedoch schwierig zu treffen. So ist bspw. für das Englische die vom *Right* verfolgte Strategie vielversprechend, weil es sich um eine nach rechts verzweigende Sprache handelt (vgl. auch van Zaanen 2000a, Abschnitt 4 sowie Abschnitt 5.2 dieser Arbeit). Der Erfolg von Edit-Distance-Algorithmen wie der in Abschnitt 2.2.1 vorgestellten Wagner-Fischer-Distanz, ist hingegen stark abhängig von den untersuchten Korpora, wie die Analyse in Kapitel 5 zeigen wird.

Aus leicht variierenden Beispielen, wie in 2.9 dargestellt, ließen sich theoretisch Folgen von terminalen Symbolen als Folgen nonterminaler Symbole (Kategorien) beschreiben, die wiederum verwendet werden könnten, um (einfache) Transformationsregeln zu erzeugen.

2.9 a. Der [[schlaue]$_X$ Physiker]$_Y$ starb.
b. Der [[serbische]$_X$ Physiker]$_Y$ starb.
c. Der [Physiker]$_Y$ starb.
d. Der [Wissenschaftler]$_Y$ starb.

e. Der [[[schlaue]$_X$, [serbische]$_X$]$_X$ Physiker]$_Y$ starb.

Aus diesem Beispiel ließe sich eine formale Definition einfacher Adjektiv-Nomen-Kombinationen extrahieren, die etwa in Form der Regeln $Y \rightarrow XY$ und $X \rightarrow XX$ repräsentiert werden könnte. Die korrekte Extraktion einiger Transformationsregeln aus unstrukturierten Korpora ist also theoretisch möglich. Um den korrekten Aufbau der rekursiven Struktur Y allerdings fehlerfrei zu erlernen (ohne also Rückschlüsse aus ähnlichen Konstruktionen auf Y zu übertragen), müssten sämtliche Beispielsätze im analysierten Korpus vorkommen: Um die Substituierbarkeit von *schlaue* und *serbische* zu erkennen, werden die Sätze 2.9.a und 2.9.b benötigt; 2.9.c ist notwendig, um die Optionalität beider Adjektive zu erkennen, und aus dem Vergleich von 2.9.c und 2.9.d ließe sich schließen, dass *Physiker* und *Wissenschaftler* substituierbar sind – dies lässt jedoch noch keinen eindeutigen Schluss darauf zu, dass die in Kombination mit *Physiker* verwendeten Adjektive auch mit *Wissenschaftler* einsetzbar sind. Aus dem letzten Satz lässt sich zwar auf die Rekursivität der Konstruktion X schließen, allerdings bliebe dabei der Aspekt der Akzeptierbarkeit, bzw. die semantische Qualität der Regel, unberücksichtigt: *Der schlaue, serbische Physiker* wäre ebenso wie *Der schlaue, serbische, schlaue, serbische Physiker* ein mögliches Ergebnis einer Transformation. Unabhängig davon bleibt zudem das Sparse Data Problem existent – es ist nicht zu erwarten, dass die Sätze aus Beispiel 2.9 tatsächlich in einem Korpus auftauchen. Eine Online-Suche[28] nach *Der serbische Physiker* ergab ca. 55 Treffer, die Wortkette *Der serbische Physiker starb* wurde jedoch bereits nicht mehr gefunden.

Demnach kann das hier skizzierte Verfahren dem zu Beginn dieses Kapitels aus Lyons (1971, S. 160) zitierten Anspruch des Strukturalismus, die Grammatik einer Sprache ausschließlich und fehlerfrei aus der Analyse von Korpora abzuleiten (vgl. Abschnitt 2.1), nicht gerecht werden. Jedoch stellt sich die Frage, inwieweit dennoch eine Näherung an diesen Anspruch gelingen kann. So könnte analysiert werden, ob durch Vorgabe einer Menge von Grundstrukturen und Basistransformationen die bezüglich Beispiel 2.9 geschilderten Probleme behoben oder zumindest reduziert werden können. Ebenfalls wäre zu untersuchen, welche strukturellen Eigenschaften mit akzeptabler Qualität detektiert werden können und welche Verfahren sich besonders gut zur Detektion von Konstituentengrenzen eignen (ohne die generierten Kategorien zu berücksichtigen). Auch könnte evaluiert werden, ob Bewertungsheuristiken die Qualität der Ergebnisse positiv beeinflussen, oder ob zusätzliche Filter-Funktionen korrekte und inkorrekte Hypothesen besser trennen können.

[28] Durchgeführt über `http://www.google.de` am 17.10.2011.

Im Bereich der Vorverarbeitung sind ebenfalls Modifikationen möglich: So kann bspw. die Anzahl unterschiedlicher Wörter im Korpus reduziert werden, indem etwa relativ zuverlässig detektierbare Kategorien wie Eigennamen, Orts- oder Zeitangaben durch abstrakte Symbole ersetzt werden oder indem POS-Tags an Stelle von Wörtern verwendet werden (wie u.a. von Klein & Manning (2002, vgl. Abschnitt 2) im Kontext eines selbstlernenden Verfahrens zur Extraktion syntaktischer Strukturen genutzt); auch sind Kombinationen beider Mechanismen denkbar. Die Art der Evaluation ist ebenfalls variierbar: Neben einer Syntax-bezogenen Analyse ist es naheliegend, die extrahierten Strukturen hinsichtlich semantischer Merkmale zu analysieren und zu prüfen, ob bzw. inwieweit das hier beschriebene Verfahren zur Detektion assoziativer Relationen geeignet ist. Schließlich ist auch zu überlegen, ob Alignment-Verfahren durch weitere, ebenfalls auf Harris' Arbeiten zurückzuführende Verfahren (wie das zu Beginn von Abschnitt 2.1.4 erwähnte HAL) ergänzt oder unterstützt werden können.

In Kapitel 5 wird diesen Ansätzen nachgegangen – dazu ist es jedoch notwendig, dass das Framework, in dem Alignment-Verfahren evaluiert werden, flexibel genug ist, neben komplexen Datenstrukturen, wie in Abbildung 2.3 für ABL4J dargestellt, auch abstraktere Konzepte (wie etwa Transformationsregeln oder semantische Ähnlichkeit) verwalten zu können, denn andernfalls ist die Integration weiterführender Komponenten, die bspw. neben Strukturhypothesen auch Transformationsregeln generieren, nur mit reduziertem Funktionsumfangs realisierbar.

Die Diskussion der Methoden und Datenstrukturen von ABL4J hat gezeigt, dass durch die Erweiterung einer existierenden Komponente neue Problemlösungsansätze untersucht werden können, sie hat gleichzeitig jedoch auch die sich daraus ergebenden Einschränkungen verdeutlicht, da Algorithmen, die intern völlig andere Datenstrukturen verwenden, ebenfalls integrierbar sein müssen. Zusätzlich muss die Vergleichbarkeit von unterschiedlichen Implementationen ähnlicher Problemlösungsstrategien gegeben sein. Diese Anforderung sollte nicht ausschließlich auf Alignment-Verfahren beschränkt, sondern für sämtliche Komponenten innerhalb eines Frameworks gelten, beispielsweise auch für Parser wie den *Stanford Parser*, dessen Datenstrukturen in Abbildung 2.4 dargestellt sind.

Ein Vergleich der in Abbildung 2.3 und 2.4 dargestellten Datenstrukturen zeigt eine grundsätzliche Analogie zwischen ABL4J und dem *Stanford Parser*: Beide Komponenten detektieren hierarchische Strukturen und beide Komponenten verwenden Konzepte wie *Baum* und *Konstituente*, auch wenn diese in ihrer Bedeutung ebenso wie in ihrer Implementation deutlich voneinander abweichen, und die Funktionalität der Datenstrukturen

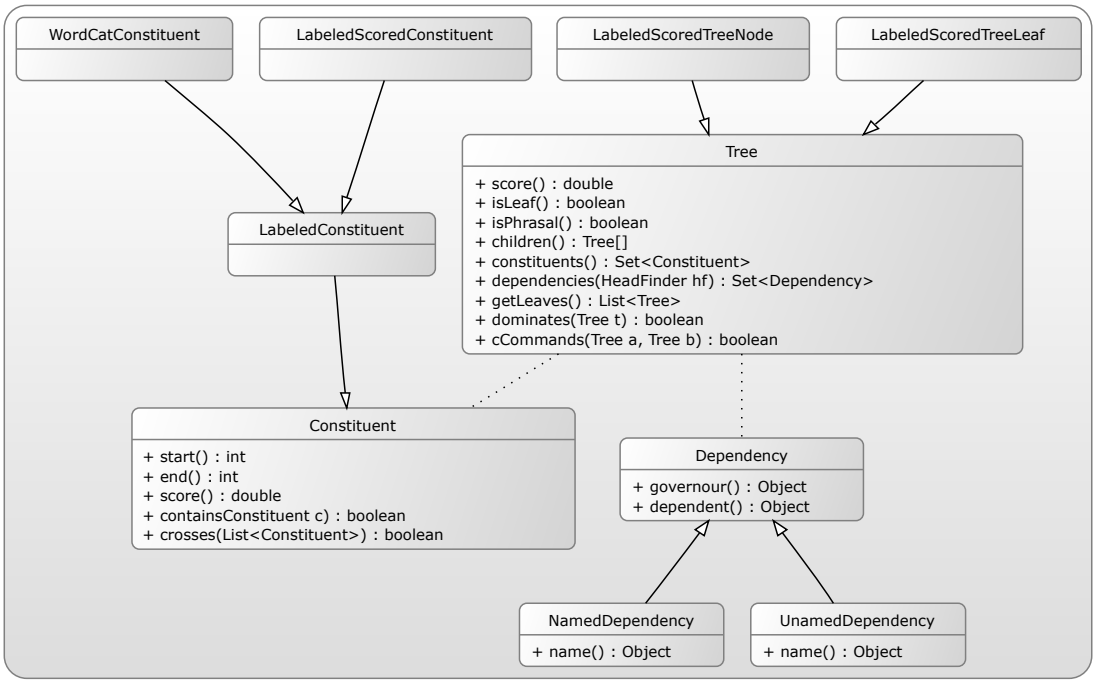

Abbildung 2.4: Klassendiagramm eines Teils der vom *Stanford Parser* genutzten Datenstrukturen. Zwecks Übersichtlichkeit werden nur ausgewählte Methoden dargestellt.

des *Stanford Parsers* mit Methoden wie `Tree.cCommands(Tree other)`[29] deutlich über die von ABL4J hinaus geht. Beide Komponenten so zu modifizieren, dass sie die gleichen Datenstrukturen oder ein kompatibles Ausgabeformat verwenden, wäre technisch zwar möglich, jedoch nur – wie oben erwähnt – mit erheblichem Aufwand und eventuellem Informationsverlust. Dennoch sollten die Komponenten vergleichbar sein, um bspw. die Qualität beider Verfahren anhand eines manuell annotierten Referenzkorpus zu evaluieren (siehe auch Kapitel 5).

Schließlich machen die Datenstrukturen des *Stanford Parsers* eine weitere Anforderung deutlich, die anhand der von ABL4J verwendeten Klassen und Interfaces nicht unmittelbar auffiel: Nicht nur primitive Datentypen und Kombinationen aus diesen – wie etwa eine Konstituente, die im einfachsten Fall als Kombination von zwei `Integern` (zur Markierung von Anfangs- und Endposition im Text) und eines `Strings` (zur Definition

[29] Unter dem Begriff C-Kommando wird eine spezielle Relation zwischen zwei Knoten in einem Baum verstanden: Ein Knoten K_1 c-kommandiert einen Knoten K_2, wenn

- der erste verzweigende Knoten über K_1 auch K_2 dominiert (d.h. auf dem Pfad von K_1 bzw. K_2 zur Wurzel des Baumes liegt) und
- weder K_1 K_2 dominiert, noch umgekehrt.

der Kategorie der Konstituente) modelliert werden könnte – sollten von einem Komponentenframework abgebildet werden können; es muss auch gewährleistet werden, dass Methoden, die von den Datenstrukturen angeboten werden (wie `Tree.cCommands(Tree other)` oder `Set<Constituent> dependencies(HeadFinder hf)` im Fall des *Stanford Parsers* oder `boolean isSimilarTo(int sentenceId)` bei ABL4J), bei der Evaluation einer Komponente mit berücksichtigt werden können.[30] Zwar könnten Relationen wie etwa C-Kommando ebenfalls mit Hilfe primitiver Datenstrukturen (etwa in Form eines Querverweises) abgebildet werden, doch scheitert ein solcher Ansatz beispielsweise dann, wenn die Menge der Relationen nicht-linear zur Menge der generierten Strukturen steigt, wie im Falle von `isSimilarTo`. Um eine Evaluierung von Alignment-Verfahren zu ermöglichen, muss ein linguistisches Komponentenframework also

- die Speicherung beliebig komplexer Datenstrukturen erlauben, um Informationsverluste zu verhindern (vgl. Abschnitt 2.2.2).
- native Zugriffe auf die Methoden komplexer Datenstrukturen ermöglichen (s.o.).
- eine Abstraktionsschicht bieten, die zwischen proprietären Datenstrukturen vermittelt, um so die Vergleichbarkeit und Wiederverwertbarkeit von Komponenten sicherzustellen (vgl. Abschnitt 2.2.1).
- uneingeschränkte Möglichkeiten zur Erweiterung bieten, damit neue Ansätze integriert und unter gleichen Kriterien evaluiert werden können (vgl. Kapitel 2.1 sowie Abschnitt 2.2.1).
- Möglichkeiten der Evaluation bieten, so dass Maße wie Precision und Recall ebenso wie sonstige Aussagen über die Qualität eines Verfahrens von Dritten überprüft werden können (vgl. Kapitel 1).
- die Komplexität, die sich aus den obigen Anforderungen ergibt, so reduzieren, dass auch Anwender ohne detaillierte Kenntnisse der verfügbaren Komponenten die Anwendung verwenden können.

Der letzte Punkt wurde bisher nicht diskutiert, ergibt sich aber aus den übrigen Anforderungen und den bisher genannten Beispielen: Einem Anwender sollte nicht zugemutet werden, sich mit den Details der Implementation eines Verfahrens auseinandersetzen zu

[30] Dies würde zudem die Wiederverwertbarkeit einer solchen Komponente erhöhen, da andere Komponenten auf die zusätzliche Funktionalität zurückgreifen könnten. In Kapitel 4 wird dieser Aspekt erneut aufgegriffen und vertieft.

müssen (also etwa die Implementation des Interfaces `HeadFinder`, welches in den Datenstrukturen des *Stanford Parsers* verwendet wird (s.o.), kennen und verstehen zu müssen), da das Framework andernfalls nur von Experten, die sowohl über die theoretischen als auch über die programmatischen Kenntnisse der vorhandenen Komponenten verfügen, benutzbar wäre. Dies würde den Kreis potentieller Anwender jedoch nicht nur stark reduzieren, sondern auch dem Anspruch der Nachvollziehbarkeit von Ergebnissen widersprechen.

Nachdem die Anforderungen an ein linguistisches Komponentenframework, in das komplexe, experimentelle Ansätze integriert werden können, in diesem Kapitel definiert wurden, werden in Kapitel 3 aktuelle Frameworks, die in den letzten Jahren entwickelt wurden, bezüglich dieser Anforderungen untersucht. Dabei – dies sei vorweggenommen – wird sich zeigen, dass keines der untersuchten Frameworks obige Anforderungen zufriedenstellend erfüllen kann. Kapitel 4 wird daher mit Tesla einen alternativen Ansatz vorstellen, der maßgeblich im Rahmen dieser Arbeit entwickelt und implementiert wurde.

In Kapitel 5 wird das hier vorgestellte ABL4J schließlich wieder aufgegriffen, um zu demonstrieren, wie sich in Tesla vom Strukturalismus inspirierte Algorithmen zur Strukturaufdeckung evaluieren und zu experimentellen Versuchsaufbauten kombinieren lassen. Es ist nicht zu erwarten, dass ein Alignmentverfahren Ergebnisse liefern wird, die denen eines Parsers oder gar einer manuellen Analyse auch nur annähernd entsprechen – wie bereits in Kapitel 1 erwähnt, liegt der Fokus dieser Arbeit vielmehr darauf, ein Framework für computerlinguistische Komponenten zu entwerfen und zu implementieren, mit dem auch komplexe Versuche durchgeführt werden können, in denen zahlreiche Komponenten flexibel kombiniert und konfiguriert wurden. ABL dient dabei lediglich als Referenz, anhand welcher die Möglichkeiten und Grenzen von Sequenzalignment in der Computerlinguistik untersucht werden. Zudem wird Kapitel 5 zeigen, wie die im Rahmen des Tesla-Projekts entwickelten Konzepte genutzt werden können, um eine Plattform zu schaffen, mit der eine einfache Sequenzalignment-Prozesskette sukzessive um elaboriertere Verfahren erweitert werden kann, und die letztlich in einer Anwendung enden könnte, die den theoretischen Anforderungen zu Beginn dieses Kapitels ebenso wie den hier aufgeführten, praktischen Anforderungen entsprechen würde.

3 Linguistische Komponentensysteme

> Let the future tell the truth, and evaluate each one according to his work and accomplishments. The present is theirs; the future, for which I really worked, is mine.
>
> *(Nikola Tesla)*

In diesem Kapitel wird ein Überblick über die Entwicklung linguistisch motivierter Komponentensysteme gegeben. Anhand verschiedener Konzepte werden exemplarisch die Systeme *GATE* (Kapitel 3.1), *UIMA* (Kapitel 3.2) und *TextGrid* (Kapitel 3.3) beschrieben und verglichen, wobei die Systeme insbesondere hinsichtlich der Umsetzung der in Kapitel 2 definierten Anforderungen analysiert werden. Auf spezialisierte Frameworks wie *Heart of Gold* [31] oder das Data Mining-Framework *Weka*[32] wird nicht weiter eingegangen, da diese nicht ohne weiteres für die Verarbeitung unstrukturierter Daten verwendet werden können. Im Folgenden soll zunächst eine kurze Einführung in den Untersuchungsgegenstand sowie eine allgemeine Beschreibung der betrachteten Konzepte gegeben werden.

Voraussetzung für die Analyse eines Komponentensystems ist trivialerweise die Definition einer Komponente. Szyperski *et al.* (1998, S.41) definieren sie (unabhängig vom konkreten System) als „unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties“ – insbesondere muss eine Komponente von einem Framework geladen und instantiiert werden können, ohne dass diese (oder von ihr verwendete Bibliotheken) bei der Kompilierung des Frameworks verfügbar waren (vgl. ebd, S. 36f). Szyperski *et al.* (1998) unterscheiden das Konzept einer Komponente von dem eines Objekts und eines Moduls: Während Komponenten über keinen extern beobachtbaren Status verfügen (sollten) und eher mit einem Schema oder einer Klasse verglichen werden können (die Autoren verwenden hierfür den Begriff „immutable plan“), handelt es sich bei Objekten (analog zur Definition innerhalb objektorientierter Programmiersprachen) um Instanzen, die einem Lebenszyklus unterliegen – sie werden zunächst erzeugt

[31] vgl. Schäfer 2006, siehe auch http://heartofgold.dfki.de/index.html.
[32] Online unter http://www.cs.waikato.ac.nz/ml/index.html, vgl. auch Witten *et al.* 2005.

und initialisiert, um anschließend (beobachtbare, aber ggfs. nicht-deterministische) Aktionen auszuführen und schließlich wieder freigegeben zu werden. Bei der Verwendung einer Komponente kann eine Vielzahl von Objekten benutzt werden, jedoch speichert die Komponente keine Referenzen auf diese und kann ebensowenig auf die Objekte zugreifen. So ist es möglich, das Prinzip von Komponenten auch in nicht objektorientierten Programmiersprachen umzusetzen.

Die Begriffe Komponente und Modul unterscheiden sich in erster Linie bezüglich der Flexibilität der Kombination und Austauschbarkeit von Komponenten: Als Modul bezeichnen Szyperski *et al.* (1998, S. 39) logische Gruppen von Klassen, bspw. Bibliotheken mathematischer Funktionen[33], die u.U. auf weitere Module zugreifen und diese statisch referenzieren, was dem Konzept einer Komponente widerspricht:

> For components, such static dependencies on component-external implementations are allowed but not recommended. Static dependencies should be limited to contractual elements, including types and constants. Dependencies on implementations should be relegated to the object level by preferring indirect over direct interfaces in module dependencies to enable flexible compositions using multiple implementations of the same interface. (Szyperski *et al.*, 1998, S. 40)

Universell einsetzbare Frameworks wie die *Java Enterprise Edition* oder das *Spring Framework* unterscheiden zwar zwischen verschiedenen Typen von Komponenten, durch die das MVC-Paradigma (*Model-View-Controller*, vgl. bspw. Gamma *et al.* 1995, S. 9) sowie architekturspezifische Konzepte umgesetzt werden können (wie von Shannon 2003, S. 6 beschrieben), doch wird der Komponenten-Begriff dort weniger strikt gefasst und als eine Implementation eines der MVC-Bestandteile interpretiert. Smeets & Ladd (2007, S. 1) und Shannon (2003, ebd.) bezeichnen beispielsweise Datenbank, graphische Benutzeroberfläche und Anwendungslogik als drei Komponenten, die von einem J2EE-Server verwaltet werden. Da solche Frameworks dazu entwickelt wurden, jede denkbare Art von Anwendungslogik zu realisieren (auch wenn häufig sog. *Enterprise-Anwendungen* fokussiert werden), ist eine feingliedrigere Abstufung des Komponenten-Konzepts auch nicht *per se* möglich, so dass unter Verwendung der Terminologie von Szyperski *et al.* (1998) hier der Begriff des Moduls benutzt werden müsste.

[33] Das Konzept des Moduls ist in Java nicht umgesetzt worden, wäre aber mit der Definition innerer Klassen in einer *Modulklasse* vergleich- bzw. emulierbar.

Im Gegensatz dazu können linguistische Komponentensysteme, die als Spezialisierungen universeller Ansätze angesehen werden können, weitergehende bzw. stärker spezialisierte Komponentenschnittstellen definieren oder aber die Definition solcher Schnittstellen ermöglichen. So wird vermutlich jedes linguistische Framework, das die Prozessierung natürlichsprachlicher Texte unterstützt, die Funktionalität eines *Taggers* benötigen und weiteren Verarbeitungsprozessen innerhalb des System zur Verfügung stellen – dies kann über eine Komponentenschnittstelle nach obiger Definition geschehen.

Allen linguistischen Komponentensystemen ist gemein, dass sie das Prinzip der Komposition für linguistische Analysen unterstützen, während bspw. die Möglichkeiten der Speicherung von Daten oder der Skalierung nicht in jedem System erweiterbar sein müssen. Doch auch wenn jedes der vorgestellten Systeme ein Komponenten-Konzept realisiert, zeigen sich teils deutliche Unterschiede in der Art der Kommunikation zwischen einer Komponente und dem System oder dem Aufbau einer Komponente. Diese Unterschiede und die sich daraus ergebenden Vor- und Nachteile sollen in den folgenden Kapiteln näher untersucht werden.

Aufgabe eines linguistisch motivierten Komponentensystems ist die Anreicherung von unstrukturiert vorliegenden Daten, wobei dies durch Zugriff auf strukturierte Daten (wie etwa Lexika), durch Algorithmen (bspw. statistische Zeichenkettenanalyse) oder durch eine Kombination von beidem (z.B. trainierbare Klassifikationsverfahren) geschehen kann. Auch wenn ein solches System konzeptuell nicht auf die Verarbeitung textueller Daten beschränkt sein muss (sondern bspw. auch Audio-Daten verarbeiten könnte), liegt der Fokus der hier vorliegenden Analyse auf der Verarbeitung unstrukturierter Texte, die mit zusätzlichen Informationen angereichert werden. Jedes der im Folgenden vorgestellten Systeme bietet dazu sowohl eine Zugriffsmöglichkeit auf Textkorpora als auch eine als *Annotation* bezeichnete Datenstruktur, in der Informationen abgelegt werden können. Die individuellen Implementationen des Annotations-Konzeptes unterscheiden sich jedoch in ihrer Flexibilität und damit in ihrem Einsatzbereich ebenso wie in den Möglichkeiten zum Zugriff auf diese (bspw. die Suche nach Annotationen mit bestimmten Merkmalen) oder hinsichtlich der Speicherung von Annotationen (in bspw. Datenbanken oder lokal vorliegenden Dateien), weshalb ein weiterer Schwerpunkt auf dem Annotationsmodell des jeweiligen Systems und den damit verbundenen Konzepten (wie etwa Persistenz) liegen soll. In Bezug auf Kapitel 2.1 wird dabei insbesondere untersucht, ob bzw. wie sich komplexe, nicht hierarchisch darstellbare Beziehungen zwischen einzelnen Elementen eines Textes abbilden lassen.

Die *Laufzeitumgebung* der Systeme wird ebenfalls genauer analysiert, d.h. die Art und

Weise, wie ausgewählte Komponenten ausgeführt werden. Schließlich wird die *Usability* der Systeme verglichen, d.h. die Anforderungen an Anwender und Entwickler sowie deren Unterstützung bei der Verwendung und Implementation von Komponenten.

3.1 GATE

Das Konzept der *Software Architecture for Language Engineering* (SALE) wurde in Cunningham (2000) eingeführt. Cunningham untersucht u.a. verschiedene Frameworks für maschinelle Sprachverarbeitung sowie deren Schwachstellen, und versucht, mit SALE einen Gegenentwurf zu etablieren, der die aufgedeckten Probleme vermeidet. Als Referenzimplementation entwickelte Cunningham das Komponentenframework GATE (*General Architecture for Text Engineering*), welches seitdem kontinuierlich weiterentwickelt wurde. In den folgenden Abschnitten sollen Komponenten- und Annotationsmodell, die für Entwickler relevante Laufzeitumgebung sowie die graphische Benutzeroberfläche für Endbenutzer vorgestellt werden.

3.1.1 Komponentenmodell

In Cunningham (2000) und Cunningham & Bontcheva (2006) wird zwischen *Language Resource* (LR) und *Processing Resource* (PR) unterschieden. Unter Language Resource werden reine Datenquellen verstanden (bspw. Lexika oder Korpora), während der Schwerpunkt von Processing Resources auf algorithmischen Bestandteilen liegt, wobei durchaus auf LRs zurückgegriffen werden kann. Während diese Unterscheidung auf den ersten Blick zweckmäßig erscheint, ergeben sich bei der genaueren Betrachtung von LRs Probleme, die nicht ohne weiteres lösbar sind. So stellen Cunningham & Bontcheva (2006, S. 14) fest: „[...] each resource has its own representation syntax and corresponding programmatic access mode (e.g. SQL for Celex, C or Prolog for WordNet)“. Zwar existieren Standardisierungen, die den Umgang mit verschiedenen LR-Typen vereinfachen sollen, diese werden jedoch nicht von jeder LR angeboten; zudem konkurrieren verschiedene Standards miteinander. Da sich dieses Problem nicht universell lösen lässt, bleibt somit die Entwicklung einer Schnittstelle zwischen einem Framework und einer LR (bzw. eines der von dieser unterstützten Standards) notwendig. Eine von Cunningham *et al.* (2000) vorgeschlagene Zugriffsschnittstelle für LRs setzt dies mit Hilfe einer zusätzlichen Abstraktionsschicht um, die als Schnittstelle zwischen dem proprietären Datenformat der Datenquelle und dem (zwar ebenfalls proprietären, aber innerhalb des Systems standardisierten) Datenformat von GATE dient. Eine in diesem Zusammenhang entwickelte zweite Schnittstelle ist

für die Bereitstellung der Daten innerhalb eines Netzwerks verantwortlich. So wird laut Cunningham & Bontcheva (2006, S.14) ein zweites zentrales Problem von LRs gelöst:

> „[...] resources must generally be installed locally to be usable, and how this is done depends on what operating systems are available, what support software is required, etc., which varies from site to site. [...] Also, there is no way to 'try before you buy': no way to examine an LR for its suitability for one's needs before licensing it *in toto*."

Während die Autoren hier übersehen, dass eine LR keinesfalls zwingend lokal vorliegen muss[34], können beide erwähnten Probleme auch PRs betreffen: Soll bspw. eine bereits existierende Standalone-Software in ein Framework eingebunden werden, muss für diese ggf. zunächst eine geeignete Umgebung geschaffen werden.[35] Ob eine 'try before you buy'-Option vorliegt, hängt zudem grundsätzlich von der Art der Lizensierung bzw. der Verfügbarkeit einer Testversion einer Ressource ab.[36] Zudem ist die Integration einer lokalen Datenquelle in ein vernetztes System häufig zwar technisch machbar, jedoch durch Nutzungsbedingungen oder urheberrechtliche Ansprüche untersagt (vgl. dazu bspw. Rehm *et al.* 2007). Im von Cunningham *et al.* (2000) vorgeschlagenen Ansatz wird anhand von vier Ressourcen versucht, eine Hierarchie von (Java-) Klassen zu definieren, die auf oberster Ebene sämtliche Merkmale der Datenquellen vereinen. Dabei stellen die Autoren allerdings fest, dass „each resource often has its own specific data structure with unique attributes and value sets that it does not share with other resources" (Cunningham *et al.*, 2000, Abschnitt 4), und schlagen daher vor, Terminologie und Taxonomie eines existierenden Standards zu verwenden und zu implementieren (vgl. ebd). Dieses Vorgehen hat jedoch mehrere Nachteile: Da, wie bereits erwähnt, nicht jede Ressource sämtliche Merkmale abbildet, enthält eine derartige generische Datenstruktur zwangsläufig leere Felder, was bei ihrer Verwendung zu unerwarteten Ergebnissen führen kann, falls ein An-

[34] Wie die Webservices des *Projekt Deutscher Wortschatz* (http://wortschatz.uni-leipzig.de/Webservices/) oder die von der *Wikipedia* bereitgestellten Texte, die zwar nicht unmittelbar mittels einer Programmierschnittstelle angesprochen, jedoch als Kopie auf einem Datenbankserver installiert werden können.

[35] Dies ist auch innerhalb der SALE-Referenzimplementation GATE für zahlreiche Komponenten der Fall, wie bei der Integration des *Tree Taggers*, eines an der Universität Stuttgart entwickelten multilingualen POS-Taggers (http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/). Zwar ist die Verwendung des Taggers kostenlos, eine Redistribution ist jedoch nur mit expliziter Einwilligung des Autors zulässig.

[36] Bedingt durch die Tatsache, dass die Erstellung von Korpora, Lexika o.ä. traditionell eher eine Domäne kommerzieller Verlage ist bzw. war, während wissenschaftliche Software häufig im universitären Umfeld entwickelt und unter einer offenen Lizenz zur Verfügung gestellt wird, mag das geschilderte Problem bei Language Resources häufiger auftreten.

wender sich nicht zuvor mit den von der Datenquelle bereitgestellten Informationen und den Anforderungen verwendeter Komponenten auseinandersetzt – dies stellt jedoch die Nützlichkeit der Vereinheitlichung von Datenquellen (aus Sicht des Anwenders) in Frage. Weiterhin ist problematisch, dass der vom System verwendete Standard von Anwendern nicht modifiziert werden kann, um bspw. für individuelle Eigenschaften einer weiteren Sprache angepasst zu werden.

Zwar ist eine Unterscheidung von Datenquellen und verarbeitenden Komponenten sinnvoll, jedoch kann das mit SALE vorgeschlagene Konzept nicht überzeugen, zumal die Referenzimplementation GATE keine konkrete Umsetzung bietet: Anders als in SALE postuliert benötigen zahlreiche Komponenten lokal installierte Zusatzsoftware, und auch wenn mit GATE 2 laut Cunningham & Bontcheva (2006, S. 9) externe Komponenten unterstützt werden, die über *Remote Method Invocation* (RMI)[37] verwendet werden können, ergab eine Untersuchung des GATE-Quellcodes[38], dass dies nur rudimentär implementiert wurde: Lediglich für Zugriffe auf das in Java implementierte Suchmaschinen-Framework *Lucene*[39] bietet GATE mit dem Interface `gate.creole.annic.apache.lucene.search.Searchable` RMI-Unterstützung, so dass ein Suchindex nicht zwingend lokal vorliegen muss. Für Komponenten wird hingegen keine universelle RMI-Schnittstelle angeboten; GATEs *Ontology Service* nutzt als einzige der standardmäßig vorhandenen Komponenten einen auf RMI basierenden Mechanismus.

Ressourcen werden in GATE geladen, indem sog. CREOLE-Repositories (*CREOLE* steht dabei für *Collection of REusable Objects for Language Engineering*) durchsucht werden. Mit Hilfe von dort bereitgestellten Metadaten, die sämtliche für die Verwendung einer Ressource benötigten Informationen enthalten, können die notwendigen Komponenten geladen und initialisiert werden. Neben der Möglichkeit, diese durch eine separate XML-Datei zur Verfügung zu stellen, wie in Listing 3.1 gezeigt, existiert seit Ende August 2008[40] zudem die Möglichkeit, Metadaten mittels Java Annotationen direkt im Quellcode unterzubringen. Eine minimale CREOLE-Definition innerhalb eines Repositories wird jedoch weiterhin benötigt, um den Pfad zur Bibliothek anzugeben.

Neben Informationen, die lediglich für die Repräsentation einer Ressource in der graphischen Benutzeroberfläche benötigt werden bspw. `ICON`, `NAME` und `COMMENT`) enthält

[37] Dabei handelt es sich um ein Kommunikationsprotokoll in Java, welches den Aufruf von Methoden über Netzwerkverbindungen definiert, so dass Java-Programme, die auf physikalischen oder virtuellen Maschinen ausgeführt werden, auf transparente Art miteinander kommunizieren können.

[38] Analysiert wurde der Quellcode von GATE 6 am 14.08.2011.

[39] Online unter http://lucene.apache.org/java/docs/index.html.

[40] Laut Protokoll des Subversion-Versionierungssystems von Sourceforge, siehe http://gate.svn.sourceforge.net/viewvc/gate/gate/trunk/src/gate/creole/metadata/.

```
<CREOLE-DIRECTORY>
        <CREOLE>
        <RESOURCE>
              <NAME>WordNet 1.6</NAME>
              <CLASS>gate.wordnet.IndexFileWordNetImpl</CLASS>
              <INTERFACE>gate.wordnet.WordNet</INTERFACE>
              <COMMENT>Princeton WordNet 1.6.</COMMENT>
              <HELPURL>http://gate.ac.uk/cgi-bin/userguide/sect:wn</HELPURL>
              <PARAMETER NAME="propertyUrl" SUFFIXES="xml"
                         COMMENT="Property File">java.net.URL</PARAMETER>
              <ICON>controller.gif</ICON>
        </RESOURCE>
            ...
        </CREOLE>
</CREOLE-DIRECTORY>
```

Listing 3.1: Creole-Definition einer Language Resource

ein `RESOURCE`-Eintrag unter `CLASS` den vollqualifizierten Namen der Klasse, die zur Prozessierung benötigt wird, sowie beliebig viele `PARAMETER`-Einträge, die der Konfiguration der Komponente dienen. Nicht formal definiert werden hingegen Anforderungen an die Eingabe, wie etwa die Art der Annotationen, die von einer Komponente benötigt werden. So lässt sich beim Aufbau einer GATE-Applikation nicht feststellen, ob diese überhaupt ausführbar ist – dies kann von einer Komponente zur Laufzeit überprüft werden, um ggf. eine Fehlermeldung auszugeben, die den Anwender Rückschlüsse auf die Fehlkonfiguration ziehen lässt (vgl. Listing 3.2). Diese aus Anwendersicht eher unkomfortable Art der Rückmeldung bei Konfigurationsfehlern lässt sich auf das in GATE verwendete Annotationsmodell zurückführen, welches im folgenden Abschnitt vorgestellt werden soll.

```
gate.creole.ExecutionException: No sentences or tokens to process!
Please run a sentence splitter and tokeniser first!
       at gate.creole.POSTagger.execute(POSTagger.java:202)
```

Listing 3.2: Beispiel einer GATE-Fehlermeldung bei falsch konfigurierter *Processing Pipeline*. Da Abhängigkeiten von verwendeten Komponenten nicht analysiert werden, treten Fehler erst zur Laufzeit auf.

3.1.2 Annotationsmodell

In GATE werden Annotationen in Form eines Annotationsgraphen verwaltet, der stark am in Bird *et al.* (2000) vorgestellten ATLAS-Annotationskonzept orientiert ist: Jede Annotation wird durch Positionsangaben mit einem Bereich im Text verankert und bietet eine *FeatureMap*, in der Attribut/Wert-Paare abgelegt werden können. Komponenten kommu-

nizieren ausschließlich über eine als *Gate Document Manager* (GDM) bezeichnete Schnittstelle, die den Annotationsgraphen kapselt und Zugriffsmöglichkeiten auf Annotationen bereitstellt. Die Menge aller Annotationen wird in einem sog. AnnotationSet abgelegt. Dieses Datenmodell wurde entwickelt, um die Vorteile des im Rahmen von TIPSTER[41] entwickelten Modells mit dem vom *Linguistic Data Consortium* (LDC) vorgeschlagenen Modell[42] zu kombinieren (vgl. Cunningham 2000, S. 98ff für eine Beschreibung der Vor- und Nachteile beider Ansätze). GATEs graphbasiertes Datenmodell entspricht im Wesentlichen dem von TIPSTER; während jedoch TIPSTER nur Grunddatentypen als Werte einer *FeatureMap* zulässt (sowie Listen und durch eindeutige Ids auch Verweise auf Annotationen), wurde diese Beschränkung für das GATE-Datenmodell aufgehoben. Zudem wurde die Anzahl von Verankerungen im Text auf zwei reduziert, so dass unzusammenhängende Bereiche nicht in einer Annotation, sondern nur über Verweise zwischen Annotationen referenziert werden können. Aus LDC wurde die Möglichkeit, mehr als ein AnnotationSet pro Dokument zu benutzen, übernommen. Die Vorteile eines graphbasierten Annotationsmodells begründen Cunningham & Bontcheva (2006, S. 19) wie folgt:

> Texts may appear to be one-dimensional, consisting of a sequence of characters, but this view is incompatible with structures like tables, which are inherently two-dimensional. Their representation and manipulation is easier in a referential model like TIPSTER than in an embedding one like SGML where markup is stored in a one-dimensional text string.

Die Art der Implementation des Annotationsgraphen in GATE bringt verschiedene Vor- und Nachteile mit sich. So kann die *FeatureMap* einer Annotation von unterschiedlichen Komponenten angereichert werden – dadurch lässt sich bspw. der Output einer Komponente nachträglich durch eine weitere Komponente verbessern, allerdings können so auch versehentlich Features überschrieben werden, zumal aus einem *FeatureMap*-Eintrag kein Rückschluss darauf gezogen werden kann, welche Komponente den Eintrag hinzugefügt hat. Das Konzept des GDM stellt eine einzige Schnittstelle für sämtliche IO-Operationen auf dem Annotationsgraphen bereit, was den Vorteil mit sich bringt, dass keine individuellen Schnittstellen für verschiedene Komponenten entwickelt werden müssen (vgl. Cunningham 2000, 127ff.), gleichzeitig jedoch auch die Flexibilität in der Entwicklung von

[41] Das bis Ende 1998 durchgeführte TIPSTER-Programm war ein Verbundprojekt mehrerer US-Behörden (u.a. des Verteidigungsministeriums und der CIA), in dem u.a. Verfahren zur Informationsextraktion und automatischen Zusammenfassung von Texten entwickelt und verbessert wurden (siehe http://www.itl.nist.gov/iaui/894.02/related_projects/tipster/).

[42] vgl. Bird & Liberman 2001.

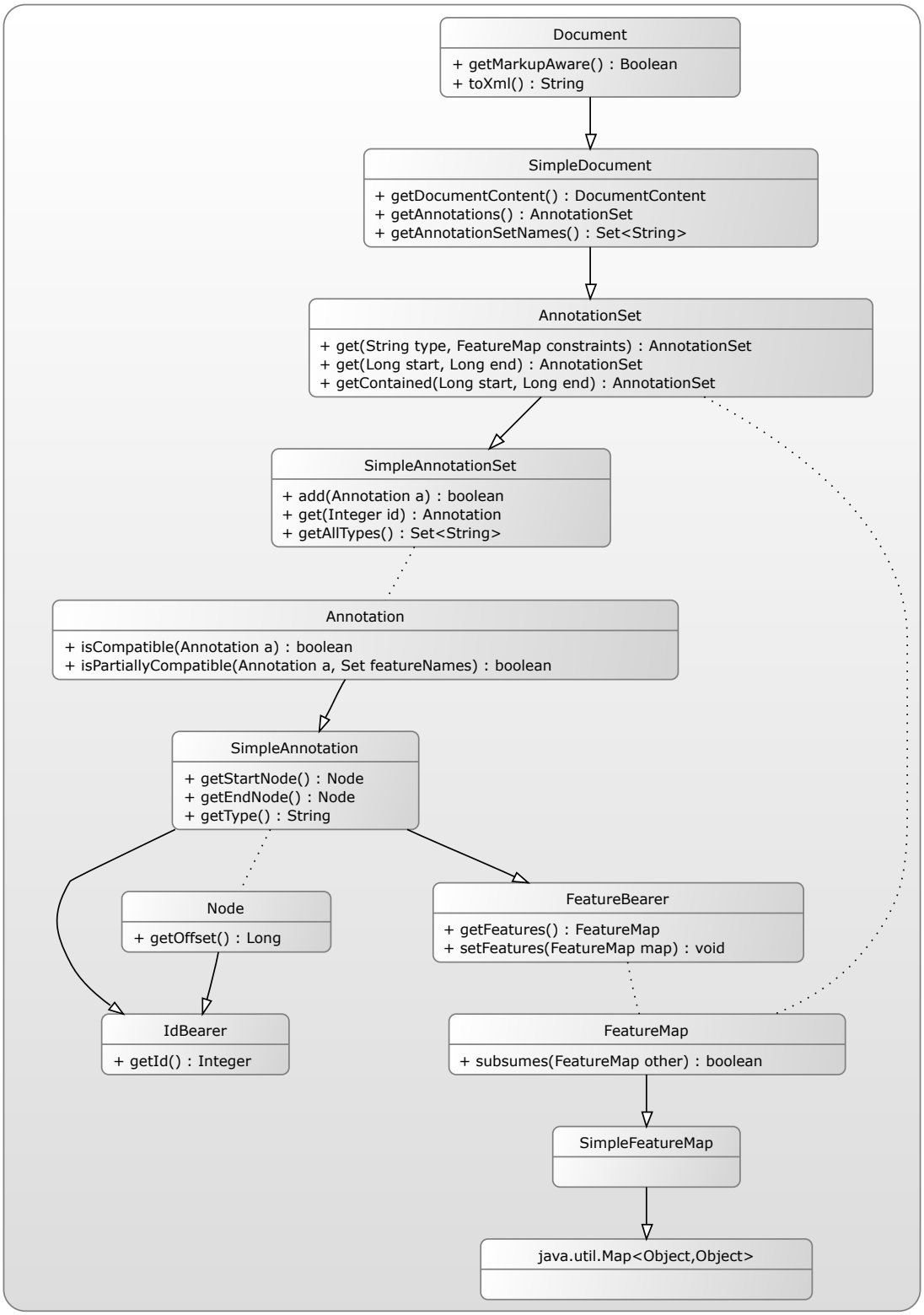

Abbildung 3.1: Klassendiagramm der im Annotationsgraphen von GATE verwendeten Interfaces. Zentrales Element ist `Annotation`, welches mit Hilfe von `Node`-Instanzen am analysierten Text verankert werden kann, und Zugriff auf eine `FeatureMap` bietet, in der beliebige Attribut-Wert-Paare abgelegt werden können. Das Interface `Document` bietet zusammen mit seinen Super-Interfaces Zugriffsmethoden auf Annotationen.

Komponenten einschränkt, da Erweiterungen oder Optimierungen der GDM-API Änderungen an GATE erfordern.

Da die in einer *FeatureMap* gespeicherten Attribute und Werte keinen Einschränkungen unterliegen[43], lassen sich zwar beliebig komplexe Datentypen verwenden, ohne vom Entwickler einer Komponente zusätzlichen Programmieraufwand zu erwarten, doch bedeutet der Einsatz solcher Datentypen auch, dass die Query-Möglichkeiten innerhalb des Annotationsgraphen eingeschränkt werden, weil dem System die genaue Struktur zwangsläufig unbekannt ist. Die Klasse AnnotationSet bietet zwar die Möglichkeit, mittels einer als Constraint interpretierten *FeatureMap* die Menge aller Annotationen zu filtern, doch basiert diese Filterung auf einer Überprüfung der Gleichheit zweier Objekte[44] – dies erfordert nicht nur Mehraufwand für Komponentenentwickler, sondern unterbindet auch eine Filterung auf Basis ausgewählter Eigenschaften eines Objekts. Werden bspw. morphologische Eigenschaften objektorientiert modelliert, enthält die resultierende Klasse vermutlich einzelne Felder wie `Numerus` oder `Kasus`, von denen u. U. nur wenige für eine Filterung relevant sind – dies lässt sich mit GATE jedoch nicht realisieren.

Ein weiterer Nachteil der *FeatureMap* liegt darin, dass die Unterstützung beliebiger Datentypen nicht konsequent eingehalten wird, denn weder in der graphischen Benutzeroberfläche noch beim XML-Export werden komplexe Datenstrukturen unterstützt: „[...] features of annotations and documents in GATE may be any virtually any [sic!] Java object; serialising arbitrary binary data to XML is not simple; instead we serialise them as strings, and therefore they will be re-loaded as strings.“ (Cunningham *et al.*, 2005, Fußnote auf S. 53)[45] – die zuvor beschriebenen Vorteile eines Annotationsgraphen gehen beim Export der Daten also verloren. Umgekehrt betrachtet bedeutet dies, dass Daten, die exportiert werden sollen, nicht objektorientiert modelliert werden können, sondern in Form primitiver Attribut/Wert-Paare in der FeatureMap abgelegt werden müssen: Die Einschränkungen des Export-Mechanismus wirken sich somit unmittelbar auf die Konzeption und das Design einer Komponente in GATE aus.

Somit eignet sich das Annotationsformat von GATE nur sehr bedingt für die Verwendung der in Kapitel 2.2.2 und 2.3 skizzierten Baumstrukturen: Da diese nicht persistiert

[43] Wie Abbildung 3.1 zeigt, akzeptiert das zugrundeliegende Interface `Map` Objekte beliebiger Klassen als Attribute ebenso wie als Werte.

[44] vgl. die Methode `subsumes()` der Klasse `gate.util.SimpleFeatureMap`, die intern die Methode `Object .equals(Object other)` verwendet.

[45] Dies trifft allerdings nicht grundsätzlich zu, vielmehr werden komplexe Features (zumindest in GATE 6.1) beim Export u.U. vollständig übersprungen (vgl. die Methode `writeFeatures` der Klasse `gate.corpora.DocumentStaxUtils`, online unter http://gate.ac.uk/gate/src/gate/corpora/DocumentStaxUtils.java)

oder dargestellt werden können, müssen sie entweder verlustbehaftet in Aggregationen primitiver Datentypen konvertiert oder unmittelbar nach der Ausführung von weiteren Komponenten verarbeitet werden. Während der erste Fall die in 2.3 bereits ausführlich dargelegten Nachteile bezüglich der in komplexen Objekten definierten Methoden mit sich bringt, führt der zweite Fall dazu, dass Ergebnisse einer Analyse nicht uneingeschränkt weiterverarbeitet werden können, da temporär verfügbare Informationen bei Visualisierung oder Export der Daten verloren gehen.

3.1.3 Laufzeitmodell

GATE ist als Desktop-Anwendung konzipiert – zwar wird die graphische Oberfläche nicht zwingend benötigt, um Anwendungen auszuführen, doch waren bis zur Veröffentlichung von Gate 6 (im November 2010) keine Möglichkeiten vorhanden, um Prozessierung und Darstellung auf getrennten Systemen ablaufen zu lassen oder um mehrere Systeme für die Prozessierung einer GATE-Anwendung zu nutzen (in Form eines Rechner-Clusters). Die Ausführungsschicht von GATE ist allerdings durch das Interface `gate.Controller` erweiterbar gehalten, so dass dies in zukünftigen Versionen implementiert werden könnte (wobei davon auszugehen ist, dass dies Änderungen am Annotationsgraph mit sich bringen würde). Auch die Aufteilung von Komponenten auf separate Threads ist in GATE nur rudimentär implementiert. So kann eine Processing Resource in einem separaten Thread ausgeführt werden, doch bietet das System in diesem Fall keine weitere Unterstützung für Entwickler oder Anwender: „Responsibility for the semantics of the interleaving of data access (who has to write what in what sequence in order for the system to succeed) is a matter for the user“ (Cunningham & Bontcheva, 2006, S. 10). Mit *GateCloud*[46] steht inzwischen eine – kostenpflichtige – Möglichkeit bereit, Daten innerhalb eines Cloud-Services prozessieren zu lassen. Der Zugriff ist jedoch nicht über die Desktop-Anwendung möglich; vielmehr muss eine Prozessierungspipeline zunächst exportiert und anschließend über ein Web-Interface an den Cloud-Service übermittelt werden.

3.1.4 GUI

Die graphische Oberfläche von GATE ist übersichtlich gehalten und bietet dem Anwender im Wesentlichen die Möglichkeiten, Anwendungen zu konfigurieren, auszuführen und Ergebnisse anzeigen zu lassen. Letzteres geschieht bspw. dadurch, dass ein prozessiertes Dokument als Fließtext, in dem Annotationen farblich hinterlegt sind, dargestellt wird

[46] Siehe https://gatecloud.net/.

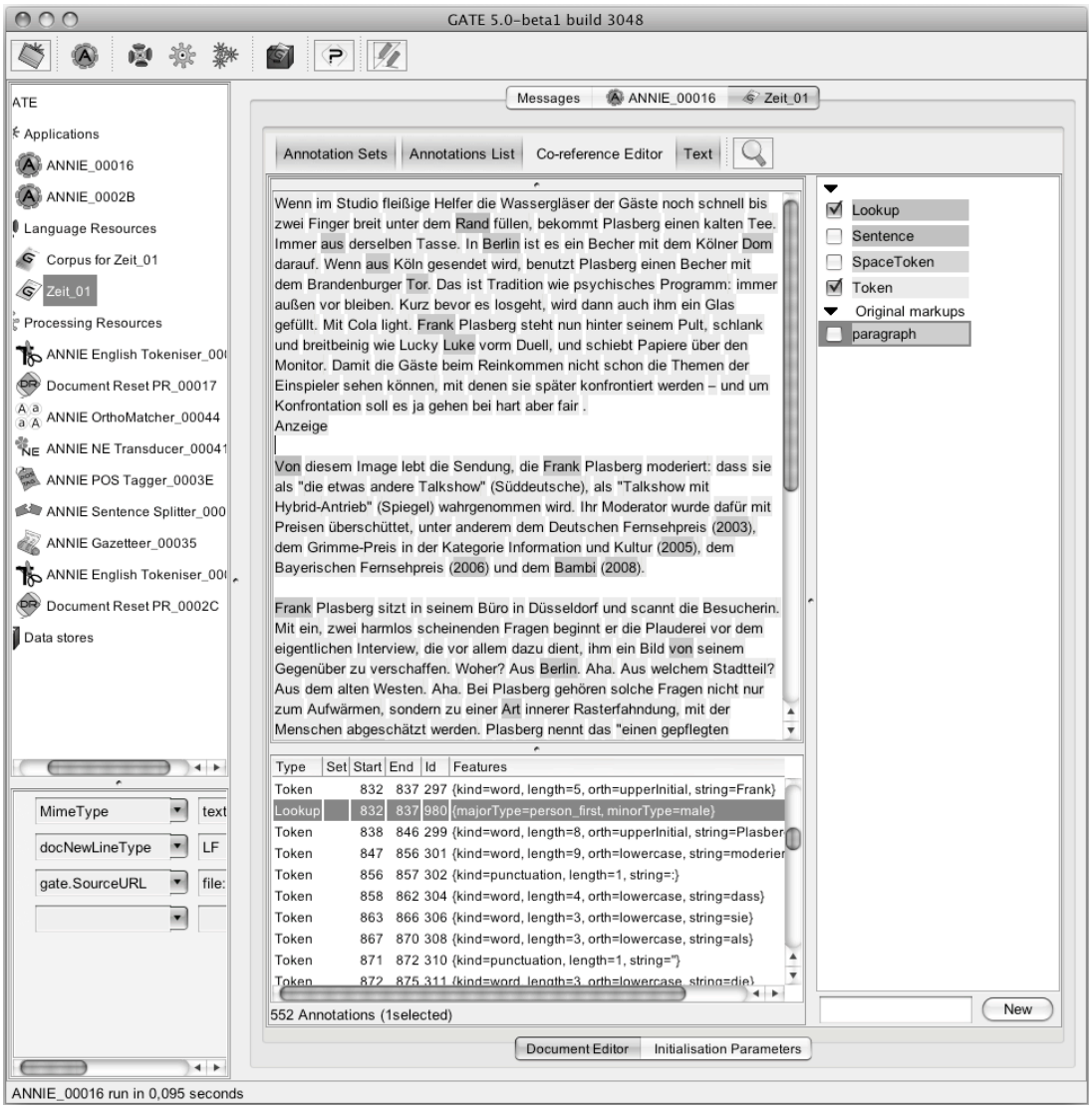

Abbildung 3.2: Graphische Benutzeroberfläche von GATE. Auf der linken Seite sind verfügbare Komponenten dargestellt, das Hauptfenster zeigt das Ergebnis einer Analyse sowohl durch Hervorhebungen im Text als auch darunter durch eine tabellarische Auflistung sämtlicher Annotationen.

(vgl. Abbildung 3.2). Unterstützt wird diese Darstellung durch eine Liste aller Annotationen sowie durch ein Overlay, in dem Informationen zum aktuell ausgewählten Bereich angezeigt werden.

Die graphische Oberfläche erlaubt es, die im Annotationsgraph enthaltenen Daten manuell zu modifizieren, eine echte Interaktion zwischen Anwender und Komponenten (etwa der Form, dass eine Komponente auf Eingaben eines Anwenders reagiert und bspw. ein Parser nach Änderung einer POS-Annotation automatisch einen neuen Syntaxbaum generiert) ist jedoch nicht vorgesehen.

Komponenten werden in Form einer *Processing Pipeline* zusammengestellt, konfiguriert und in sequentieller Folge ausgeführt. Je nach Komponente bzw. der von dieser verwendeten CREOLE-Definition können unterschiedliche Parameter modifiziert werden – so kann bspw. der *ANNIE-POS-Tagger*[47], der u.a. Satz- und Token-Annotationen als Input benötigt und mit Standard-Einstellung die Werte *Sentence* und *Token* erwartet, für die Verwendung alternativer Bezeichnungen konfiguriert werden.

Als eigenständiges Programm bietet GATE keine IDE zur Entwicklung neuer Komponenten, bietet jedoch einen Wizard, mit dem aus wenigen Benutzerangaben eine Creole-Beschreibung sowie eine Basisklasse generiert werden, die innerhalb einer beliebigen Java-Entwicklungsumgebung verwendet werden können.

3.1.5 Diskussion

GATE ist als eines der wichtigsten computerlinguistisch motivierten Komponentensysteme zu betrachten, was nicht zuletzt daran liegt, dass es sich um eines der ersten Komponentensysteme handelt, das in einer Vielzahl von Projekten verwendet wurde.[48]

Insgesamt zeigt sich jedoch, dass Cunninghams *Software Architecture for Language Engineering* bezüglich der Möglichkeiten zur Komponentenentwicklung relativ eingeschränkt ist. Während die Plattform GATE für einfachere Komponenten noch ausreicht, ist dies bei komplexeren Komponenten nicht mehr der Fall. Die Implementation als Desktop-Anwendung verhindert die Verarbeitung großer Korpora und die Nutzung ressourcenintensiver Algorithmen; die Anpassung existierender Lösungen an das System ist aufgrund fehlender Schnittstellen nur bedingt möglich. Eine der in Kapitel 2.3 formulierten Anforderungen an ein Komponentensystem bestand darin, komplexe Komponenten wie beispielsweise den *Stanford Parser* integrieren zu können, was in GATE auch umge-

[47] Siehe http://gate.ac.uk/sale/tao/splitch6.html#sec:annie:tagger
[48] vgl. die Liste der externen Projekte unter http://gate.ac.uk/projects.html.

setzt wurde, allerdings mit verschiedenen Einschränkungen[49]: So ist es u.a. notwendig, dass vor der Verwendung des Parsers ein Tagger ausgeführt wird, welcher Sätze, Wörter und Satzzeichen mit den Zeichenketten `Sentence` und `Token` annotiert, was bedeutet, dass die Komponente nicht ausführbar ist, wenn alternative Bezeichnungen verwendet werden. Dies kann jedoch leicht der Fall sein, etwa dann, wenn ein Tokenizer bei der Auszeichnung terminologisch zwischen Wörtern, Zahlen und Satzzeichen unterscheidet. Zwar ist es relativ einfach, eine Konvertierung der verwendeten Terminologie vorzunehmen – allerdings nur, solange eine isomorphe Abbildung der verwendeten Terme möglich ist (vgl. auch Cunningham 2000, S. 90), was im skizzierten Beispiel nicht der Fall ist. Die in GATE verwendete uneingeschränkte bzw. nicht-standardisierte Art der Definition von Bezeichnungen und Werten innerhalb einer FeatureMap ist auch unter dem Aspekt der in Abschnitt 3 diskutierten Abgrenzung zwischen Komponente und Modul nach Szyperski *et al.* (1998) problematisch: Dadurch, dass das System keinen gemeinsamen Kontrakt bezüglich der Ein- und Ausgabeschnittstellen der Komponenten erfordert, sondern im Gegenteil Abhängigkeiten zwischen konkreten Implementationen verschiedener Komponenten nahelegt, wird die Austauschbarkeit funktional äquivalenter oder verwandter Komponenten deutlich erschwert – das in GATE verwendete Komponentenmodell entspricht somit nicht der Definition nach Szyperski *et al.* (1998).

Weiterhin müssen die vom *Stanford Parser* generierten Datenstrukturen (vgl. Abbildung 2.4 auf Seite 41) in eine FeatureMap-Repräsentation überführt werden, um visualisiert oder exportiert werden zu können, und um weiteren Komponenten, die GATEs Möglichkeiten zur Suche nach Merkmalen innerhalb einer FeatureMap verwenden, zur Verfügung zu stehen.

Während hier ebenso wie beim ANNIE-POS-Tagger willkürliche Zeichenketten zur Bezeichnung einzelner Features verwendet werden, die die Austauschbarkeit verschiedener Parser-Implementationen erschweren (vgl. etwa Zeilen 17 bis 23 in Listing 3.3), findet gleichzeitig ein Informationsverlust statt, da die Logik der vom *Stanford Parser* verwendeten Datenstrukturen nicht vollständig in GATEs Datenmodell abgebildet werden kann. Nicht zuletzt aus diesem Grund ist der Aufwand zur Integration einer komplexen Komponente in GATE vergleichsweise hoch und beträgt im Fall des *Stanford Parsers* ca. 1000 Zeilen Code (einschließlich Kommentaren), von denen ca. 300 Zeilen auf die Modellierung der Datenstrukturen und ca. 700 Zeilen auf die Konvertierung von Ein- und Ausgabeformat sowie die Ausführung des Parsers entfallen.

[49] Untersucht wurden die am 29.11.2010 aktuellen Versionen der Klassen, die für die Einbindung des *Stanford Parsers* in GATE benötigt werden (Revision 11821 im GATE-Repository).

```
1 private Annotation annotatePhraseStructureConstituent(Long startOffset, Long
2   endOffset, String label, List<Integer> consists, int depth) {
3
4    Annotation phrAnnotation = null;
5    Integer phrID;
6    try {
7      String cat;
8      if (useMapping && mappingLoaded) {
9        cat  = translateTag(label);
10     }
11     else {
12       cat = label;
13     }
14     if (addConstituentAnnotations) {
15       String text = document.getContent().getContent(startOffset, endOffset).toString();
16       FeatureMap fm = gate.Factory.newFeatureMap();
17       fm.put(PSG_TAG_FEATURE, cat);
18       fm.put("text", text);
19       /* Ignore empty list features on the token-equivalent annotations. */
20       if (consists.size() > 0) {
21         fm.put("consists", consists);
22       }
23       phrID  = annotationSet.add(startOffset, endOffset, OUTPUT_PHRASE_TYPE, fm);
24       phrAnnotation = annotationSet.get(phrID);
25       recordID(annotationSet, phrID);
26     }
27     if ( addPosTags && (depth == 1) ) {
28       ...
29     }
30   }
31   catch (InvalidOffsetException e) {
32     e.printStackTrace();
33   }
34   return phrAnnotation;
35 }
```

Listing 3.3: Ausschnitt aus der für die Integration des Stanford-Parsers verantwortlichen GATE-Komponente `gate.stanford.Parser`.

Die Diskussion zeigt, dass vor allem das *FeatureMap*-Konzept nicht ausreichend für einen flexiblen Datenaustausch ist, wenn die Komplexität von Datenstrukturen zunimmt und über einfache Key-Value-Paare hinausgeht. Daher kann die Entwicklung des Komponentenframeworks *UIMA*, welches im folgenden Abschnitt vorgestellt wird, auch als Konsequenz aus den genannten Einschränkungen interpretiert werden.

3.2 UIMA

Die *Unstructured Information Management Architecture* (UIMA) wurde ursprünglich von *IBM Research* entwickelt. Ziel war es, eine einheitliche Plattform bereitzustellen, die unterschiedlichen Projekten mit Bezug auf das Management unstrukturierter Daten als gemeinsame Basis dienen sollte (vgl. Ferrucci & Lally 2003), um so die Wiederverwertbarkeit von

Entwicklungen zu erhöhen, die Notwendigkeit von Weiterbildungen zu verringern und die Entwicklungsdauer neuer Produkte zu reduzieren (vgl. Ferrucci & Lally 2004, S. 328). Nach vier Jahren Entwicklungszeit wurde im August 2005 beschlossen, das Framework als Open Source Software freizugeben[50], ein Jahr später wurde es an die *Apache Software Foundation* übergeben, die es seitdem kontinuierlich weiterentwickelt[51]. UIMA greift einige Grundideen von GATE auf, bietet jedoch mehr als nur eine einzelne Schnittstelle zur Erweiterung des Systems an, so dass sich deutlich flexiblere Einsatz- und Erweiterungsmöglichkeiten ergeben. Dies begründen Ferrucci & Lally (2004, S. 329) wie folgt:

> A primary design point for UIMA is that it should admit the implementation of middleware frameworks that provide a component-based infrastructure for developing UIM applications suitable for vastly different deployment environments.

Auf diese Schnittstellen wird in den Abschnitten 3.2.1 und 3.2.3 näher eingegangen, das als *Common Analysis Structure* (CAS) bezeichnete Annotationsmodell wird in Abschnitt 3.2.2 vorgestellt.

3.2.1 Komponentenmodell

Analog zu GATE wird in UIMA zwischen Ausgangsdaten und verarbeitenden Komponenten unterschieden. Zugriffe auf Daten werden durch *Collection Reader* realisiert. Verarbeitende Komponenten werden in zwei Gruppen unterteilt: Zum einen *Text Analysis Engines* (TAEs), die konzeptuell GATEs *Processing Resources* entsprechen und einzelne Dokumente annotieren, zum anderen *CAS Consumer*, die Relationen zwischen mehreren Dokumenten verarbeiten können und bspw. dazu verwendet werden können, Lexika o.ä. zu erzeugen, oder eine CAS zu persistieren[52]. Ferrucci & Lally (2004) begründen die Trennung zwischen beiden Komponententypen unter anderem technisch: So kann die Ausführung von TAEs ohne zusätzlichen Implementationsaufwand parallelisiert werden[53],

[50] vgl. Pressemitteilung vom 8.8.2005 unter http://www-03.ibm.com/press/us/en/pressrelease/7822.wss.

[51] Siehe http://uima.apache.org/news.html.

[52] Auf der UIMA-Website werden CAS Consumer zur Persistierung von Daten mit Hilfe von *Lucene* und der Lucene-Erweiterung *Solr* (http://lucene.apache.org/solr/) zum Download angeboten (siehe http://uima.apache.org/sandbox.html, eine Abbildung der CAS auf relationale Datenbanken ist ebenfalls möglich). Zusätzliche Persistenzmechanismen können durch Implementation eines entsprechenden CAS Consumers hinzugefügt werden.

[53] Eine Parallelisierung der Ausführung einer TAE kann durch Konfiguration des *Collection Processing Managers* (siehe Abschnitt 3.2.3), der die Anwendung von TAEs auf Dokumente steuert, erreicht werden.

während dies bei *CAS Consumer*-Komponenten nicht ohne weiteres möglich ist. Wörterbücher und andere externe, strukturierte Daten werden über eine separate Schnittstelle (*Knowledge Source Adapter*) in das System eingebunden und können anschließend von Komponenten verwendet werden.

Um eine TAE in Java zu implementieren, muss das Interface `Annotator` implementiert werden, das die Methoden *initialize*, *process* und *destroy* fordert (wobei eine abstrakte Basisklasse den Aufwand auf die Implementation der process-Methode reduziert). Listing 3.4 zeigt einen Ausschnitt eines POS-Tagger-Wrappers, der die oben beschriebene Datenstruktur verwendet.

```
1   ...
2   Sentence sentence = (Sentence) sentenceIterator.next();
3   FSIterator tokenIterator = tokenIndex.subiterator(sentence);
4   while (tokenIterator.hasNext()) {
5      Token token = (Token) tokenIterator.next();
6      tokenList.add(token);
7      tokenTextList.add(token.getCoveredText());
8   }
9   List tokenTagList = tagger.tag(tokenTextList);
10  for (int i = 0; i < tokenList.size(); i++) {
11     Token token = (Token) tokenList.get(i);
12     String posTag = (String) tokenTagList.get(i);
13     POSTag pos = null;
14     try {
15        pos = (POSTag) AnnotationTools.getAnnotationByClassName(aJCas, postagset);
16        pos.setBegin(token.getBegin());
17        pos.setEnd(token.getEnd());
18        pos.setValue(posTag);
19        pos.setComponentId(COMPONENT_ID);
20        pos.addToIndexes();
21     } catch (...Exception e1) {
22        ...
23     }
24     FSArray postags;
25     if (token.getPosTag() == null) {
26        postags = new FSArray(aJCas, 1);
27        try {
28           postags.set(0, pos);
29        } catch (CASRuntimeException e) {
30           ...
31        }
32        postags.addToIndexes();
33        token.setPosTag(postags);
34     }
35     else {
36        postags = JulesTools.addToFSArray(token.getPosTag(), pos, 1);
37     }
38  }
39  ...
```

Listing 3.4: Ausschnitt der Klasse *POSTagAnnotator*, die ebenso wie Listing 3.6 der am JulieLab entwickelten UIMA-Komponente *OpenNLP POS Tagger* (siehe Abschnitt 3.2.2) entnommen wurde.

Listing 3.4 zeigt, dass die automatisch generierten Klassen bereits eine deutliche Vereinfachung gegenüber dem in GATE verwendeten *FeatureMap*-Modell (vgl. Listing 3.3 auf Seite 59) darstellen: Die dort erforderliche, systemorientierte Art der Entwicklung wird hier durch parametrisierte Methodenaufrufe ersetzt, wodurch sowohl die Lesbarkeit des Codes erhöht als auch die Anfälligkeit für Fehler reduziert wird.[54]

UIMA integriert zudem eine als *semantic search* bezeichnete Query-Schnittstelle, die im Vergleich zu GATE (vgl. Abschnitt 3.1.2) flexibleres Suchen und Filtern ermöglicht, wie Ferrucci & Lally (2004, S. 334) festhalten:

> The key feature of the query interface is that it supports queries that may be predicated on nested and overlapping structures [...] and tokens in addition to Boolean combinations of tokens and annotations.

3.2.2 Annotationsmodell

Das Konzept der Annotation in UIMA ähnelt dem von GATE: Ein prozessiertes Dokument wird zusammen mit seiner Analyse in einer als *Common Analysis Structure* (CAS) bezeichneten Struktur abgelegt, wobei einzelne Annotationen über Positionsangaben mit dem Dokument verbunden werden und den annotierten Bereich so um zusätzliche Metadaten anreichern. Bei der CAS handelt es sich um ein (u.a. in Form von XML oder Binärdaten serialisierbares) Annotationsformat, das inzwischen von der *Organization for the Advancement of Structured Information Standards* (OASIS) standardisiert wurde[55]. Durch die Verwendung von XML als Austauschformat ist es grundsätzlich möglich, Komponenten in beliebigen Programmiersprachen zu entwickeln.[56] Zudem kann UIMA dadurch flexibel in Datenverarbeitungsprozesse eingebunden werden und bspw. auch in produktiv genutzten Umgebungen für die Extraktion und Aufbereitung von Informationen genutzt werden.

Im Gegensatz zu GATE muss der Entwickler einer Komponente zunächst sowohl die Struktur der produzierten Analyse definieren als auch ggf. zulässige Datentypen angeben (vgl. Ferrucci & Lally 2004, S. 331), bevor die Komponente neue Informationen in der CAS ablegen kann. Listing 3.5 zeigt eine solche Datenstruktur anhand der Definition ei-

[54] Dies ist allerdings nur der Fall, wenn eine Komponente eine domänenspezifische CAS verarbeitet: Wird hingegen, wie von Verspoor *et al.* (2009) vorgeschlagen, ein abstrakteres Typsystem verwendet, kann dieser Vorteil gegenüber dem FeatureMap-Modell nicht ausgenutzt werden.

[55] vgl. http://www.oasis-open.org/news/oasis-news-2009-03-19.php.

[56] Unter der Voraussetzung, dass die Sprache von der UIMA-Laufzeitumgebung unterstützt wird, vgl. Kapitel 3.2.3.

```
<?xml version="1.0" encoding="UTF-8"?> <typeSystemDescription
    xmlns="http://uima.apache.org/resourceSpecifier">
  <name>julie-morpho-syntax-types</name>
  <description>...</description>
  <version>2.1</version>
  <vendor/>
  <types>
    ...
    <typeDescription>
      <name>de.julielab.jules.types.POSTag</name>
      <description>...</description>
      <supertypeName>de.julielab.jules.types.Annotation</supertypeName>
      <features>
        <featureDescription>
          <name>tagsetId</name>
          <description>Every POS tagset (see subtypes) has an identifier, C</description>
          <rangeTypeName>uima.cas.String</rangeTypeName>
        </featureDescription>
        <featureDescription>
          <name>value</name>
          <description>The value of POS (NN, JJ etc.)</description>
          <rangeTypeName>uima.cas.String</rangeTypeName>
        </featureDescription>
      </features>
    </typeDescription>
    ...
  </types>
</typeSystemDescription>
```

Listing 3.5: Definition einer CAS-Annotation am Beispiel eines POS-Tags. Der Ausschnitt wurde dem am *JULIE Lab* entwickelten *UIMA Type System* entnommen.

nes POS-Tags[57]: Hier wird der Annotationstyp *POS* definiert, welcher aus den Features *tagsetId* und *value* besteht. Durch die CAS wird verhindert, dass eine Komponente beliebige Features produziert und zugleich sichergestellt, dass die IO-Schnittstellen einer Komponente im System registriert und validiert werden können. Zudem erleichtert dieses Konzept den Umgang mit der CAS, da aus der Definition des Outputs einer Komponente ein Objekt-Modell generiert werden kann, das von Entwicklern verwendet werden kann, ohne mit der zugrundeliegenden XML-Datenstruktur arbeiten zu müssen. Neben einer Implementation in C++ und Java liegt auch eine Prolog-Schnittstelle vor, die aus der CAS eine Prolog-Wissensbasis erzeugen und diese wieder in die CAS-Spezifikation transformieren kann (vgl. Fodor *et al.* 2008).

Wie Listing 3.6 anhand eines Ausschnitts der aus Listing 3.5 erzeugten Java-Klasse im JCas-Format[58] zeigt, kapselt der generierte Quellcode die Funktionalität, die für Schreib-

[57] Das Beispiel wurde dem am *JULIE Lab* der Universität Jena entwickelten *UIMA Type System 2.6.8* entnommen (siehe https://julielab.de/Resources/Software/NLP+Tools/Download/UIMA+Type+System-p-98.html) und stammt aus der Datei der Datei *julie-morph-syntax-types.xml*.

[58] Dargestellt ist ein Ausschnitt einer vom *OpenNLP POS Tagger* verwendeten Klasse, der unter https://

```
1
2 public class POSTag extends Annotation {
3   ...
4   public String getTagsetId() {
5     if (POSTag_Type.featOkTst && ((POSTag_Type)jcasType).casFeat_tagsetId == null)
6       jcasType.jcas.throwFeatMissing("tagsetId", "de.julielab.jules.types.POSTag");
7     return jcasType.ll_cas.ll_getStringValue(addr,
          ((POSTag_Type)jcasType).casFeatCode_tagsetId);}
8
9   public void setTagsetId(String v) {
10    if (POSTag_Type.featOkTst && ((POSTag_Type)jcasType).casFeat_tagsetId == null)
11      jcasType.jcas.throwFeatMissing("tagsetId", "de.julielab.jules.types.POSTag");
12    jcasType.ll_cas.ll_setStringValue(addr, ((POSTag_Type)jcasType).casFeatCode_tagsetId, v);}
13
14  public String getValue() {
15    if (POSTag_Type.featOkTst && ((POSTag_Type)jcasType).casFeat_value == null)
16      jcasType.jcas.throwFeatMissing("value", "de.julielab.jules.types.POSTag");
17    return jcasType.ll_cas.ll_getStringValue(addr, ((POSTag_Type)jcasType).casFeatCode_value);}
18
19  public void setValue(String v) {
20    if (POSTag_Type.featOkTst && ((POSTag_Type)jcasType).casFeat_value == null)
21      jcasType.jcas.throwFeatMissing("value", "de.julielab.jules.types.POSTag");
22    jcasType.ll_cas.ll_setStringValue(addr, ((POSTag_Type)jcasType).casFeatCode_value, v);}
23 }
```

Listing 3.6: Auszug der aus Listing 3.5 generierten JCas-Klasse, die vom am *JULIE Lab* entwickelten *OpenNLP POS Tagger* verwendet wird.

und Lesezugriffe auf mittels CAS definierte Eigenschaften einer Annotation benötigt werden. Gleichzeitig wird eine grundlegende syntaktische Integrität der Daten sichergestellt, indem die Existenz dieser Eigenschaften geprüft wird. Eine Definition möglicher Wertebereiche (wie etwa eine Einschränkung des Wertebereichs einer Zahl auf positive Zahlen o.ä.) ist allerdings nicht möglich.

Eine weitere, wesentliche Verbesserung gegenüber des in GATE verwendeten Annotationsformats besteht darin, dass die CAS Typhierarchien unterstützt: Eine in CAS definierte Datenstruktur ist entweder direkt von der systemseitig vorgegebenen Struktur `uima.tcas.Annotation` oder von einer ihrer Subtypen abgeleitet und erbt die dort festgelegten Eigenschaften. So ist es bspw. möglich, generische Basistypen zu entwickeln, die abhängig von konkreten Anwendungsszenarien weiter spezialisiert werden, gleichzeitig aber kompatibel zu den Basistypen bleiben. Da diese Hierarchie auch in JCas verwendet (bzw. generiert) wird, kann dies zu einer im Vergleich zu GATE verbesserten Wiederverwertbarkeit von Komponenten führen. Voraussetzung hierfür ist allerdings, dass sämtliche Komponenten das gleiche Typsystem verwenden, wie etwa die durch das *JULIE Lab* an

`julielab.de/Resources/Software/NLP+Tools/Download/UIMA+Analysis+Engine-p-94.html` zum Download angeboten wird.

der Universität Jena entwickelten UIMA-Komponenten[59]. Diese verwenden ein speziell für linguistisch motivierte Annotationen definiertes Typsystem, das verschiedene Anwendungsfälle abdeckt: Buyko & Hahn (2008, Abschnitt 3.3) unterscheiden zwischen Struktur und Metadaten eines Dokuments sowie syntaktischen, morphosyntaktischen und semantischen Auszeichnungsebenen, wobei auf jeder Ebene mit Hilfe einer Typhierarchie eine Spezialisierung der Basistypen umgesetzt werden kann. So können bspw. Gemeinsamkeiten und Unterschiede der Tag-Sets verschiedener POS-Tagger abgebildet werden.

Der in Buyko & Hahn (2008) vorgestellte Ansatz kann jedoch auch kritisiert werden: So argumentieren bspw. Verspoor *et al.* (2009) gegen ein domänenspezifisches Typsystem, und schlagen stattdessen ein leichtgewichtiges, generisches Typsystem als Alternative vor, das für die jeweilige Forschungsdomäne spezifiziert werden kann, wobei diese Spezialisierung nicht durch Subtypen erfolgen soll, sondern durch die Parametrisierung der Basistypen vorgenommen wird. Ein derartiges Metamodell würde die Wiederverwertbarkeit von Komponenten weiter verbessern und eine Inkompatiblität zwischen Komponenten, die auf der Verwendung unterschiedlicher Typsysteme beruht, vermeiden.

3.2.3 Laufzeitmodell

Die Definition einer Analyse wird in UIMA durch eine *Collection Processing Engine* (CPE) repräsentiert – dabei handelt es sich um eine Kombination von *Collection Reader*, *TAE* und *CAS Consumern*, wobei zusätzlich noch eine als *CAS Initializer* bezeichnete Komponente für die Konfiguration des Annotationsgraphen und die Aufbereitung der zu verarbeitenden Daten benötigt wird. Ausgeführt wird eine CPE durch einen *Collection Processing Manager*, der gleichzeitig für Parallelisierung, Fehlerbehandlung u.ä. verantwortlich ist (vgl. Ferrucci & Lally 2004, S. 333).

Das Design der UIMA-Middleware orientiert sich am in der *Java 2 Enterprise Edition* vorgeschlagenen und implementierten Konzept verschiedener Entwickler-Rollen: Dort wird ein Rollenmodell vorgestellt, das unterschiedliche Vorgänge und Anforderungen innerhalb der Entwicklung einer Business-Applikation abbildet, wie etwa die Rolle des *Application Assemblers*, dessen Aufgabe die Kombination einzelner Komponenten zu einer Applikation ist, oder die Rolle des *System Component Providers*, in dessen Verantwortlichkeit die Entwicklung von Schnittstellen zum Umgang mit externen Ressourcen liegt (vgl. Shannon 2003, S. 16ff). In UIMA werden diese Rollen als *Analysis Engine Assembler* respektive *Analysis Engine Deployer* bezeichnet und vom *Annotator Developer*, dessen Aufgabe in

[59] Online unter http://julielab.de/Resources/Software/NLP_Tools.html.

der Entwicklung neuer Komponenten besteht, abgegrenzt. Zweck des Rollenmodells (sowohl in der J2EE als auch bei UIMA) ist es zum einen, verschiedene Aufgaben innerhalb eines Projekts entsprechend der individuellen Spezialisierungen der beteiligten Entwickler zu vergeben und so die Einarbeitungszeit gering zu halten. Die zahlreichen Schnittstellen, die an der Ausführung einer UIMA-Applikation beteiligt sind, ermöglichen zum anderen, die Middleware flexibel zu erweitern, ohne Änderungen am Systemkern vornehmen zu müssen. Ferrucci & Lally (2003, S. 73) nennen beispielhaft die Möglichkeit, den Datenfluss zwischen Komponenten für unterschiedliche Umgebungen (wie etwa innerhalb eines Clusters oder auf mobilen Geräten) anzupassen, ohne dafür Modifikationen an der CAS vornehmen zu müssen.

Zunächst stellte UIMA für derartige Fälle keine wohldefinierte Schnittstelle bereit, so dass individuelle Modifikationen notwendig waren (vgl. Egner *et al.* 2007). Mit der 2008 veröffentlichten Erweiterung *UIMA Asynchronous Scaleout* (UIMA-AS)[60] wurden jedoch die erforderlichen Anpassungen umgesetzt – die Leistungsfähigkeit der sich daraus ergebenden Möglichkeiten, insbesondere eine Echtzeit-Datenverarbeitung, konnte mit dem Anfang 2011 durchgeführten *IBM Jeopardy Challenge* veranschaulicht werden.[61]

3.2.4 GUI

UIMA bietet neben einigen Programmen zur Visualisierung von prozessierten Daten, die in etwa die Funktionalität des GATE-Viewers abbilden, auch mehrere Eclipse-Plugins[62], dank der die Entwicklung neuer UIMA-Komponenten vereinfacht wird.

So zeigt Abbildung 3.3 den Editor zur Modifikation des Typsystems – dargestellt wird das in Abschnitt 3.2.2 erwähnte morphosyntaktische Typsystem des *JULIE Lab*, im linken Bereich des Fensters wird zudem die Hierarchie der generierten JCas-Klasse `POSTag` gezeigt. Hierbei handelt es sich zwar nicht um ein UIMA-spezifisches Element der graphi-

[60] Siehe http://uima.apache.org/doc-uimaas-what.html.

[61] Das von *IBM Research* entwickelte System *Watson* nahm als Konkurrent der bisher erfolgreichsten Spieler an der Quizshow *Jeopardy!* teil, bei der in möglichst kurzer Zeit Fragen zu vorgegebenen Antworten formuliert werden müssen. Watson war als ein Cluster von 90 Servern mit je vier 8-Kern-Prozessoren realisiert, so dass dem System – neben insgesamt 16 Terabyte Arbeitsspeicher – 2.880 Prozessorkerne zur Verfügung standen, um durch Anwendung verschiedener Algorithmen auf unstrukturiert vorliegende Ausgangdaten die gesuchte Frage stellen zu können – Datenverarbeitung und Skalierung wurden auf Basis von UIMA und UIMA-AS realisiert (Siehe http://www-03.ibm.com/innovation/us/watson/). Watson gewann das auf drei Folgen aufgeteilte Quiz mit deutlichem Abstand, was die Leistungsfähigkeit der Architektur (sowohl eindrucksvoll als auch publikumswirksam) demonstrieren konnte.

[62] Eclipse sei hier als eine in Java implementierte, plugin-basierte Entwicklungsumgebung definiert, deren Basisfunktionalität durch eigene Plugins erweitert werden kann. Für eine ausführlichere Beschreibung des Eclipse-Frameworks sei auf Abschnitt 4.1.7 verwiesen.

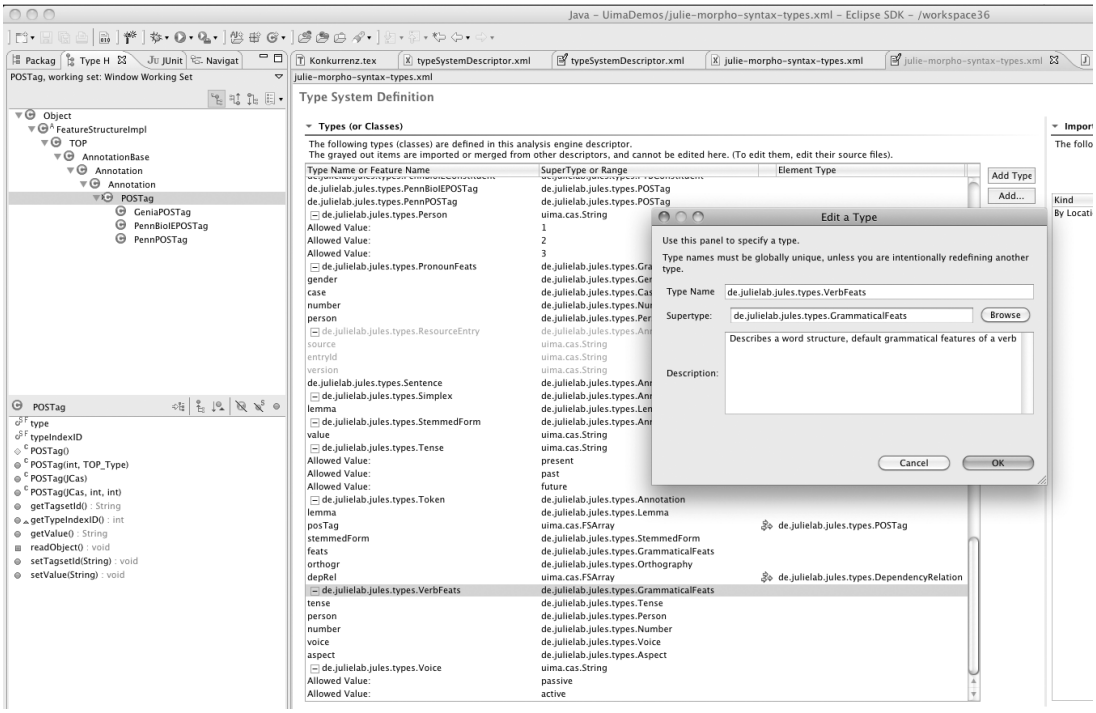

Abbildung 3.3: Der von UIMA verwendete CAS-Editor. Abgebildet ist die Definition morphosyntaktischer Typen des *JULIE Lab*.

schen Oberfläche (sondern um eine durch Eclipse zur Verfügung gestellte Ansicht), doch veranschaulicht dies den Mehrwert, den die Integration von UIMA in Eclipse bereitstellt: Die dort vorhandenen Werkzeuge zur Entwicklung mit Java können grundsätzlich für den Umgang mit JCas-Klassen und damit auch für die Implementation neuer Komponenten verwendet werden.

Die UIMA-IDE unterstützt dabei auch Refactoring, d.h. die Modifikation existierender CAS und JCas – wird bspw. eine bestehende JCas-Klasse manuell modifiziert, so wird diese Modifikation bei der Neugenerierung der Klasse auch dann übernommen, wenn die zugrundeliegende CAS ebenfalls modifiziert wurde. Allerdings ist die Refactoring-Unterstützung unvollständig: Wird beispielsweise ein Element einer Datenstruktur umbenannt und durch *JCasGen* in eine Java-Klasse konvertiert, so ändern sich die entsprechenden *set*- und *get*-Methoden nur innerhalb dieser Klasse, jedoch nicht an anderen Stellen eines Programms – diese müssen anschließend manuell bearbeitet werden.

Neben weiteren Editoren für UIMA-spezifische Konfigurationsdateien werden keine Ergänzungen zur Oberfläche von Eclipse hinzugefügt – zur Ausführung von Komponenten muss eine externe Anwendung gestartet und konfiguriert werden (vgl. Abbildung 3.4).

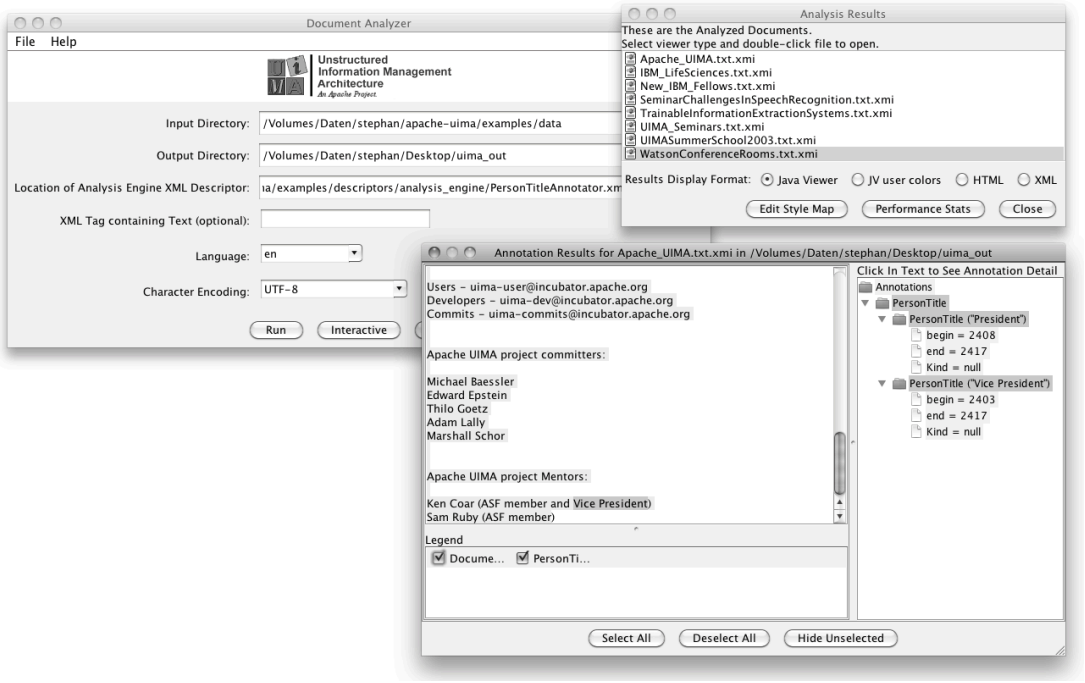

Abbildung 3.4: Teile der graphischen Oberfläche von UIMA. Links ist die Oberfläche zur Ausführung einer Analysis Engine dargestellt, rechts die Übersicht über alle prozessierten Dokumente. Das unterste Fenster zeigt eine detaillierte Ansicht zu einem dieser Dokumente, die der Visualisierung in GATE ähnelt (vgl. Abbildung 3.2).

3.2.5 Diskussion

Die Definition der *Common Analysis Structure* als Austauschformat ist sinnvoll, bleibt aber auf die Struktur der Daten beschränkt: Zwar weisen Ferrucci & Lally (2004, S. 336) darauf hin, dass sich aus der CAS generierte Datenstrukturen erweitern lassen, um bspw. den Umgang mit komplexen Datenstrukturen mittels des *Facade Design Patterns*[63] zu erleichtern, doch sind evtl. hinzugefügte Zugriffsmethoden nicht über die CAS spezifiziert – auch bei UIMA wird ein rein datenorientierter Ansatz verfolgt, der in diesem Punkt und im Vergleich mit GATE lediglich um eine semantisch angereicherte Zwischenschicht erweitert wurde.

Die Erweiterung der Entwicklungsumgebung Eclipse vereinfacht zwar die Implementation neuer Komponenten, ist jedoch u.U. nicht ausreichend, wenn mit häufigen Refactoring-Prozessen zu kalkulieren ist, wie etwa innerhalb des in Kapitel 2 vorgestellten, vergleichsweise experimentellen Einsatzgebietes – neben der in Abschnitt 3.2.1 erwähnten unvollständigen Unterstützung bei der Neugenerierung von JCas-Klassen erschwert auch die Komplexität der Konfiguration von Komponenten eine experimentorientierte Vorgehensweise.

Diese Kritik betrifft allerdings nicht das gesamte System, sondern lediglich die graphische Oberfläche und die damit verbundenen Anforderungen; zudem liegt der Anspruch der UIMA-Entwickler nicht darin, ein umfassendes Framework zur Komponentenentwicklung bereitzustellen, sondern (wie bereits eingangs von Abschnitt 3.2 erwähnt), die Basis eines solchen Frameworks zu realisieren, wie Ferrucci & Lally (2004) klarstellen:

> The UIMA project was initiated at IBM based on the premise that an architectural foundation for developing, integrating and deploying UIM technologies, if adopted, would facilitate reuse and integration of natural language technologies under development at IBM and elsewhere with each other and within IBM products.

Als Schnittstelle für die Integration verschiedener Werkzeuge ist UIMA dank der wohldefinierten Ein- und Ausgabeformate hingegen gut geeignet, und die in Abschnitt 3.2.3 erwähnten Architektur- und Rollenkonzepte vereinfachen eine Verwendung des Frameworks in produktiv genutzten Umgebungen. Die im März 2009 erfolgte Standardisierung der UIMA-CAS-Definition durch OASIS bietet Entwicklern zudem Planungssicherheit,

[63] Zur Definition dieses Entwurfsmusters siehe Gamma *et al.* (1995).

und die Integration in bspw. IBM's *Omnifind*[64] oder die an der *Defense Advanced Research Projects Agency* entwickelte *Global Autonomous Language Exploitation* (GALE)[65] zeigen, dass diese Vorteile gegenüber GATE auch genutzt werden.

Den in Abschnitt 2.3 aufgestellten Anforderungen an ein Framework, das experimentelles Arbeiten mit in der Entwicklung befindlichen Verfahren, Nachvollziehbarkeit von Zwischenergebnissen und einfache Änderungen an komplexen Datenstrukturen sowie den Zugriff auf diese ermöglicht, wird UIMA jedoch nur eingeschränkt gerecht. Insbesondere die Verwendung der CAS, welche in produktiv eingesetzten Workflows von großem Vorteil ist, kann in diesem Zusammenhang als ein Nachteil angesehen werden, der sowohl Refactoring-Prozesse erschwert als auch die Modularität von Komponenten einschränkt, da zwischen Komponenten auszutauschende Daten stets in ein zwar standardisiertes und wohldefiniertes, aber u.U. ineffizientes Format konvertiert werden müssen.

3.3 TextGrid

TextGrid ist ein im Rahmen der *D-Grid*-Initiative im Februar 2006 begonnenes Verbundprojekt, an dem verschiedene Institutionen aus Wissenschaft und Wirtschaft beteiligt sind. Ziel der durch das Bundesministerium für Bildung und Forschung initiierten D-Grid-Initiative ist es, eine digitale Infrastruktur aufzubauen, durch die der Austausch großer Datenmengen (wie sie beispielsweise in Klimaforschung oder Astrophysik anfallen) vereinfacht und kollaboratives Arbeiten auch für geographisch getrennte Forschungs- und Rechenzentren ermöglicht wird[66]. Mit TextGrid soll die Verfügbarkeit einer solchen Infrastruktur auch auf geisteswissenschaftliche Anwendungen und Forschungsfelder erweitert werden – so berichten Gietz *et al.* (2006, S. 135) davon, dass zu Testzwecken eine mehr als 4 Terabyte Rohdaten umfassende multimediale Edition der Werke des Schriftstellers Jean Paul verwendet wird. Kuster *et al.* (2007) verweisen zudem darauf, dass nur durch kollaborative Strategien eine moderne Edition umfangreicher oder umfangreiches Fachwissen erfordernder Werke erstellt werden kann[67].

Wie die Beispiele bereits vermuten lassen, steht – anders als bei GATE und UIMA – in der aktuellen Entwicklungsphase von TextGrid weniger eine Prozessierungsmöglichkeit im Vordergrund als vielmehr die Aufgabe, Daten zu verteilen und kollaborativem Arbeiten

[64] Online unter http://www-01.ibm.com/software/data/enterprise-search/.
[65] Siehe Website unter http://www.darpa.mil/Our_Work/I2O/Programs/Global_Autonomous_Language_Exploitation_%28GALE%29.aspx.
[66] vgl. die Website des D-Grid Projekts unter http://www.d-grid.de/.
[67] Als Beispiel wird dort die *Capitularia Benedicti Levitae* genannt, eine aus über 1.700 Kapiteln bestehende mittelalterliche Fälschung, deren letzte Edition über 150 Jahre zurückliegt.

zugänglich zu machen. Zwar erwähnen Gietz *et al.* (2006), dass es wünschenswert wäre, die Modularisierung bestehender Komponentensysteme auf TextGrid zu übertragen, weisen jedoch darauf hin, dass dies bei einem u.a. auch geographisch weit verteiltem System nicht ohne weiteres möglich sei. Hinzu kommt, dass TextGrid-Komponenten die Interaktion mit Anwendern unterstützen sollen, wie Kuster *et al.* (2007, S. 508) hervorheben:

> In our point of view it is crucial, though, to emphasize that these ecosystems are populated by human and digital inhabitants alike and that they are only created [...] through the *interactions* between both human and computer-based agents [...].

Diese Anforderung erschwert allerdings die Definition einer Komponentenschnittstelle, da eine Komponente in diesem Fall während der Ausführung evtl. Eingaben eines Nutzers benötigt. Entsprechend existiert bisher noch kein konkret definiertes Komponentenmodell – Gietz *et al.* (2006, S. 137) erwähnen jedoch, dass zusätzliche *Web Services* implementiert werden können, um TextGrid zu erweitern.

Das in TextGrid eingesetzte Annotationsmodell basiert auf den Vorschlägen der *Text Encoding Initiative* (TEI), welches in Abschnitt 3.3.1 vorgestellt und diskutiert wird.

Da TextGrid als verteilte Infrastruktur angelegt ist, die von verschiedenen Forschergruppen parallel genutzt wird, ohne dass jedem Anwender Zugriff auf sämtliche Ressourcen und Workflows gestattet werden soll, wurde in TextGrid eine Schnittstelle zur Verwaltung von Zugriffsrechten integriert. So können verschiedene Ressourcen unterschiedlichen Benutzergruppen zugeteilt werden – hier wird zwischen Projektleiter, Administrator, Bearbeiter und Betrachter unterschieden. Wurde ein Anwender bezüglich einer Ressource in keine der genannten Gruppen aufgenommen, wird diese Ressource automatisch ausgeblendet. Ein TextGrid-kompatibles Komponentensystem könnte diese Funktionalität verwenden, um die Verwaltung von Korpora zu vereinfachen und die in Kapitel 1 erwähnte Berücksichtigung von Copyright und Urheberrecht (vgl. auch Abschnitt 4.1.3) ebenso wie die notwendige Infrastruktur zur Distribution großer Textmengen mit TextGrid zu realisieren.

3.3.1 Annotationsmodell

Die Text Encoding Initiative wurde im Jahr 1987 als nicht-kommerzielle Organisation gegründet, deren Ziel es ist, ein lizenzfreies, plattformunabhängiges Format zur Annotation digital vorliegender Texte zu entwickeln und zu pflegen (vgl. Vanhoutte 2004, S. 14). Die

inzwischen in fünfter Revision[68] vorliegenden und mehr als 1000 Seiten umfassenden *TEI Guidelines* definieren ein XML-basiertes Datenformat, das dazu dienen soll, den Datenaustausch zu vereinfachen, applikationsunabhängige Prozessierung zu ermöglichen und bei der Erstellung von Texten und der Extraktion von Daten zu helfen (vgl. Abschnitt iv.1.2 (Seite xxvi) des Vorworts von Burnard & Bauman 2008). Gleichzeitig wurde versucht, die TEI Guidelines möglichst allgemein zu halten:

> These Guidelines apply to texts in any natural language, of any date, in any literary genre or text type, without restriction on form or content. They treat both continuous materials ('running text') and discontinuous materials such as dictionaries and linguistic corpora. (Burnard & Bauman, 2008, S. xxiii)

Um die Komplexität der Auszeichnungen möglichst gering zu halten – nach Vanhoutte (2004, S. 11) werden bereits in der vierten Revision der TEI Guidelines mehr als 600 verschiedene Elemente definiert – ist das XML-Schema modular aufgebaut, so dass ein Basis-Modul um zusätzliche Spezifikationen (bspw. für die Annotation von Prosa, Lyrik oder Dramen, von Handschriften oder aber zur linguistischen Beschreibung natürlichsprachlicher Texte) erweitert werden kann. Zudem lassen die TEI Guidelines meist mehr als eine Möglichkeit zu, Text zu annotieren, und ermöglichen es so, flexibel auf spezifische Anforderungen einzugehen.

Die Unabhängigkeit linguistischer Annotation von zugrundeliegender Theorie und analysierter Sprache kann beispielsweise dadurch erreicht werden, dass, wie Listing 3.7 zeigt, in der Analyse der Struktur eines Satzes auf beliebige, selbst definierte Klassen verwiesen wird. Dies verhindert, dass ein in sich geschlossenes und damit im Zweifelsfall unvollständiges Tagset benutzt werden muss.

Da es sich bei XML um ein streng hierarchisches Format handelt, werden in den TEI Guidelines verschiedene Möglichkeiten vorgestellt, um Querverweise, wie sie bspw. für die Annotation von Pronominalreferenzen oder Alignment benötigt werden, innerhalb einer Strukturbeschreibung umzusetzen (vgl. Burnard & Bauman 2008, S. 611ff): Durch Redundanz (in Form identischer Annotation), Markierungen mit leeren Elementen (um bspw. Satzanfang und -ende zu markieren, können Elemente wie `<beginSentence/>` und `<endSentence/>` eingefügt werden), Fragmentierung und Rekonstruktion (Aufteilung überlappender Strukturen, die bei der Interpretation wieder zusammengefügt werden müssen) oder durch *Stand-Off Markup* (Trennung von Text und Auszeichnungen).

[68] Das erste *Proposal* der TEI Guidelines wurde bereits 1990 veröffentlicht.

```
<!-- Definition einer Nominalphrase -->
<phr ana="#n">
        <phr ana="#gn">
                <w ana="#AT0">The</w>
                <w ana="#NN1">victim</w>
                <m ana="#POS">‘s</m>
        </phr>
        <w ana="#NN2">friends</w>
</phr>

<!-- (Theoriespezifische) Interpretationsregeln fuer Part-of-Speech-Tags -->
<interpGrp type="POS">
        <interp xml:id="AT0">Definite article</interp>
        <interp xml:id="NN1">Noun singular</interp>
        <interp xml:id="NN2">Noun plural</interp>
        <interp xml:id="POS">Genitive marker</interp>
</interpGrp>

<!-- (Theoriespezifische) Interpretationsregeln fuer Konstituenten -->
<interpGrp type="constituentFunction">
        <interp xml:id="n">nominal</interp>
        <interp xml:id="gn">genitive</interp>
</interpGrp>
```

Listing 3.7: Auszug der linguistischen Annotation des Satzes *The victim's friends told police that Kruger drove into the quarry and never surfaced* (aus Burnard & Bauman 2008, S. 539).

Durch ausschließliche Verwendung von XML (bzw. XSD) als Sprache zur Formatbeschreibung werden die TEI Guidelines von zahlreichen Programmen, wie etwa den XML-Editoren *XML Spy*[69] und *Oxygen*[70], unterstützt, zudem existieren Tools, die bspw. die Office-Suite *OpenOffice*[71] um eine TEI-konforme Exportmöglichkeit erweitern.[72] Ferner bieten die TEI Guidelines mit dem Konzept der *Feature Structures* die Möglichkeit, sowohl binäre Daten als auch primitive Datentypen zu komplexen Strukturen zusammenzufassen, die bei Bedarf auch im Wertebereich eingeschränkt und bezüglich struktureller Korrektheit validiert werden können (vgl. Burnard & Bauman 2008, S. 563ff). Dies gilt jedoch nur für hierarchische Strukturen – andernfalls ist eine Validierung nicht möglich, wie Burnard & Bauman (2008, S. 611) ebenfalls anmerken:

> Non-nesting information poses fundamental problems for any XML-based encoding scheme, and it must be stated at the outset that no current solution combines all the desirable attributes of formal simplicity, capacity to repre-

[69] Siehe http://www.altova.com/products/xmlspy/xmlspy.html.
[70] Online unter http://www.oxygenxml.com/.
[71] Siehe http://www.openoffice.org.
[72] Einen Überblick über TEI-kompatible Werkzeuge bietet das Wiki des Konsortiums unter http://wiki.tei-c.org/index.php/Category:Tools.

> sent all occurring or imaginable kinds of structures, suitability for formal or mechanical validation. The representation of non-hierarchical information is thus necessarily a matter of trade-offs among various sets of advantages and disadvantages.

Grundsätzlich bietet TextGrid durch die Umsetzung der TEI Guidelines ein Annotationsmodell, das in vielen Punkten von vergleichbarer Flexibilität und Mächtigkeit wie das Konzept der CAS ist, durch die strikte Fokussierung auf (geistes-)wissenschaftliche Textanalyse jedoch elaboriertere Strukturierung und Formalisierung ermöglicht. Allerdings wurden in den bisher veröffentlichten Aufsätzen zu TextGrid noch keine Schnittstellen zu Programmiersprachen erwähnt, die – analog zu UIMAs JCAS-Generator – den Umgang mit TEI-konformen Dokumenten vereinfachen. Die von TextGrid verwendeten TEI-Spezifikationen sind bisher[73] nicht für einen Einsatz in linguistischen oder computerlinguistischen Domänen ausgelegt: So findet sich lediglich für die Konstruktion *Morphem* eine einfache Definition, die eine Auszeichnung von Präfix, Wurzel und Suffix vorsieht.

3.3.2 GUI

Da in geisteswissenschaftlichen Disziplinen (im Vergleich mit Naturwissenschaften) die Verwendung von unterstützender Fachsoftware aufgrund der rechnerisch schwerer zu fassenden Forschungsgegenstände weniger verbreitet ist, definieren Gietz *et al.* (2006, S. 136) die folgenden Anforderungen an TextGrid:

- be easy to install and use (user interfaces and publication platform);
- offer flexible support (user defined workflows and data structures, extensibility and modularity).

Um diesen Anforderungen seitens der Benutzeroberfläche gerecht zu werden, wurde die Benutzeroberfläche von TextGrid als *Rich Client* auf Basis des Eclipse-Frameworks[74] entwickelt. Nach Beantragung eines Accounts kann das als *TextGridLab* bezeichnete Frontend mit einem TextGrid-Server verbunden werden. Die Benutzeroberfläche stellt anschließend verfügbare Korpora und Werkzeuge (wie etwa eine Möglichkeit zur Suche in Wörterbüchern, vgl. Abbildung 3.5) dar, während ein graphischer Editor für die Erstellung neuer

[73] Untersucht wurde der Quellcode von TextGrid 1.0 am 13.08.2011.
[74] Vgl. Kapitel 4.1.7, in dem Eclipse in Zusammenhang mit Tesla vorgestellt wird.

Workflows verwendet werden kann – dies gelang in der zunächst untersuchten Beta-Version von TextGrid (Revision 3408) jedoch nicht. Da der Editor in der finalen Version 1.0 nicht mehr enthalten ist, kann hier nur spekuliert werden, ob die zugrundeliegende Funktionalität zur Zeit überarbeitet wird, oder ob sie permanent entfernt wurde.

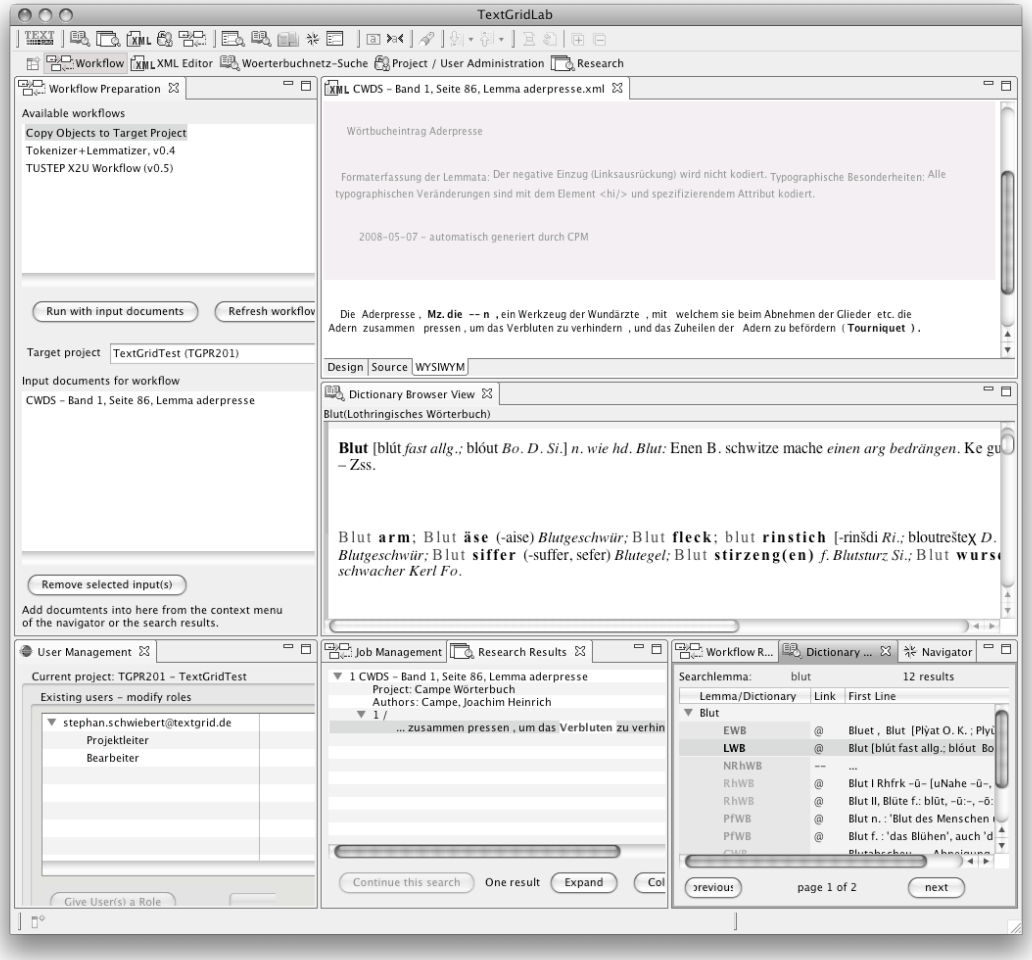

Abbildung 3.5: Graphische Benutzeroberfläche von TextGrid mit einer individuellen Zusammenstellung verschiedener *Views*.

Abbildung 3.5 zeigt eine Auswahl der Views des TextGridLabs, die Zugriff auf verschiedene Funktionen bieten. Im Gegensatz zu Systemen wie GATE oder UIMA wird hier keine universelle Visualisierungs-Schnittstelle verwendet, vielmehr werden die Ergebnisse in auf die jeweilige Aufgabe angepassten Views dargestellt. Dies ermöglicht zwar eine optimale Aufbereitung der anzuzeigenden Daten, erfordert jedoch zusätzlichen Programmieraufwand, wenn neue Dienste in die Oberfläche integriert werden sollen.

3.3.3 Diskussion

In der aktuellen Phase des TextGrid-Projekts zeigt sich deutlich, dass der Fokus der Entwicklungsarbeit auf der Implementation eines Systems zur kontrollierten Distribution textueller Daten liegt, das potentiell dazu in der Lage ist, beliebig viele geisteswissenschaftliche Forschergruppen unabhängig vom Standort der einzelnen Mitglieder zu vernetzen. Bedingt durch die Rechteverwaltung bleibt die Oberfläche von TextGrid übersichtlich, unabhängig von der Anzahl der Projekte, Ressourcen, Komponenten oder Forschergruppen, da diese nur dann dargestellt werden, wenn entsprechende Zugriffsrechte vorliegen. Allerdings fehlt so auch die Möglichkeit, auf Projekte zu stoßen, die evtl. für eine Zusammenarbeit oder einen wissenschaftlichen Austausch von Interesse wären.

Die Verwendung des von der *Text Encoding Initiative* vorgeschlagenen Formats ist einerseits aufgrund der Standardisierung und Etablierung des XML-Dialekts positiv zu beurteilen, andererseits ist dieses Format nur eingeschränkt als Austauschformat zwischen Komponenten innerhalb eines Systems einsetzbar: Die in Abschnitt 3.3.1 bereits angeführten Probleme bei der Auszeichnung nicht hierarchischer Relationen zwingen zu Workarounds, die nur bedingt syntaktisch validierbar und zudem semantisch proprietär sind, so dass die Integration einer Datenabstraktionsschicht, die eine solche Funktionalität bietet, notwendig wäre, um diesbezüglich mit der Leistungsfähigkeit der CAS oder eines ähnlichen Formats konkurrieren zu können. Die (von den Entwicklern erwünschte und unter dem Gesichtspunkt der Theorieunabhängigkeit auch notwendige) Flexibilität der TEI bei der Definition neuer Tags erschwert die Verwendung innerhalb eines Komponentensystems zusätzlich, da Komponenten nur dann austauschbar sind, wenn ihre IO-Schnittstellen identisch sind – anders als bei UIMA (vgl. Abschnitt 3.2.2) liegt bisher jedoch keine explizite Schnittstellen-Definition vor. TEI wird zudem permanent weiterentwickelt, so dass nicht ausgeschlossen ist, dass zukünftige Versionen nicht rückwärtskompatibel zur aktuellen Version sein werden – dies war etwa im November 2007 bei der Umstellung des Standards von Version P4 zu P5 der Fall.[75]

Die Ausrichtung auf geisteswissenschaftliches Arbeiten, die sich beispielsweise im Rechtemanagement bezüglich der – oftmals urheberrechtlich geschützten – Korpora, aber auch in der Fokussierung auf die Mensch-Maschine-Interaktion zeigt (und sich damit von naturwissenschaftlichen Disziplinen wie der Bioinformatik oder Meteorologie unterscheidet), sowie die Integration des TextGrid-Clients in Eclipse und die damit verbundene Erweiterbarkeit um zusätzliche domänenspezifische Funktionen machen das System je-

[75] vgl. http://www.tei-c.org/Guidelines/P5/migrate.xml.

doch zu einem (innerhalb der Geisteswissenschaften) potentiell universell einsetzbaren Framework, auch wenn die Komponenten-Schnittstelle auf Basis von Webservices bisher deutlich unterspezifiziert ist.

Letzteres kann jedoch auch als Vorteil gesehen werden, da so eine flexible Schnittstelle zur Verfügung steht, die von anderen Komponentensystemen, die im Gegensatz zu TextGrid eine vollautomatische Datenverarbeitung, jedoch keine Rechte- oder Gruppenverwaltung anbieten, bedient werden könnte.

3.4 Zusammenfassung

Keines der hier vorgestellten Frameworks genügt den in Abschnitt 2.3 aufgeführten Ansprüchen. Dies kann darauf zurückgeführt werden, dass in jedem der drei untersuchten Ansätze eine starke Fokussierung auf den Datenexport vorliegt, die sich restriktiv auf die Annotationsmöglichkeiten auswirkt. Als ein Grund kann dabei die Entscheidung, XML als Austauschformat zu nutzen, betrachtet werden, da sich diese nur bedingt für die Auszeichnung komplexer Objektgraphen eignet:

XML wurde 1996 vom *World Wide Web-Consortium* (W3C) spezifiziert und seitdem kontinuierlich weiterentwickelt und in nahezu sämtlichen Anwendungsbereichen der IT eingesetzt: So wurden bspw. XML-Dialekte für Vektorgrafiken (*Scalable Vector Graphics* (SVG)) ebenso wie komplexe Office-Dokumente (*Open Document* und *Office Open XML*) definiert, Schnittstellen für einen plattformunabhängigen Zugriff auf Internet-Dienste mittels XML realisiert (*WSDL*, *SOAP* oder *RSS*) und Container für Metadaten (wie sie bspw. von den Frameworks *Spring* und *Hibernate*, siehe Abschnitt 4.2 dieser Arbeit, zu Konfigurationszwecken benötigt werden) geschaffen.

Da XML lediglich syntaktische Eigenschaften eines Dokuments berücksichtigt, ist es möglich, Dialekte zu definieren, die flexibel auf Anforderungen und Eigenschaften des jeweiligen Einsatzgebietes eingehen können. Kombiniert mit der Vielzahl von Anwendungen und Werkzeugen, die XML verarbeiten und modifizieren können, wird XML so zu einem geeigneten Speicherformat verschiedenster Anwendungen und kann insbesondere zum Datenaustausch zwischen diesen verwendet und in automatisierte Workflows[76] eingebunden werden.

[76] Im Rahmen dieser Arbeit wurde bspw. ein Großteil der Grafiken durch das Programm *Graphviz* generiert, im SVG-Format exportiert, mit einem XSLT-Stylesheet nachbearbeitet und mit Hilfe von *Inkscape* in PDFs umgewandelt, die schließlich in das vorliegende Dokument eingebettet werden konnten. Dieser Prozess wurde zudem automatisiert, so dass Änderungen an einer Datei keine weiteren, manuell durchzuführenden Arbeitsschritte erforderten.

Insbesondere der letzte Punkt ist dabei im Kontext von UIMA hervorzuheben, da dieses Framework für derartige Einsatzbereiche entwickelt wurde. Auch innerhalb von TextGrid ist die Verwendung von XML als gemeinsam genutztes Format sinnvoll: Schwerpunkt dieser Plattform ist der Austausch zwischen verschiedenen Forschergruppen, denen eine gemeinsame Infrastruktur angeboten werden soll, was den Einsatz eines flexibel nutzbaren und dennoch hoch strukturierten Datenformats erfordert. Beide Anwendungsfälle entsprechen den bei der Entwicklung von XML definierten Zielen, wie der Abschnitt *Origins and Goals* der W3C-Webseite zu XML[77] zusammenfasst – die Intention der Entwickler lag darin, den Dokumentenaustausch über das Internet zu standardisieren und zu vereinfachen. Es wurde hingegen nicht unter dem (unerfüllbaren) Anspruch entwickelt, ein universelles Format für sämtliche Formen von Daten zu definieren.[78]

Dies kann in Bezug auf korpuslinguistische Komponentensysteme und die in Abschnitt 2.3 beschriebenen Anwendungsfälle und Datenstrukturen hinderlich sein, was sich bspw. daran zeigt, dass

1. Annotationen nicht notwendigerweise dokumentbezogen sind, sondern auch Inhalte verschiedener Dokumente in Relation zueinander gesetzt oder aggregiert werden können[79],

2. nicht jede Form der Annotation speicher- und laufzeiteffizient innerhalb einer hierarchischem (Baum-)struktur abgebildet werden kann (wie Graphen oder hochdimensionale Vektoren und Matrizen), und

3. Algorithmen, die auf Annotationen operieren, nicht oder nur unter Einschränkungen[80] implementiert werden können.

Derartige Anforderungen müssen nicht notwendigerweise von einem Komponentensystem erfüllt werden: Wie bereits erläutert ist dies abhängig vom angestrebten Einsatzbereich eines Systems. Allerdings verhindert die Fokussierung auf XML eine experimentelle,

[77] Siehe http://www.w3.org/TR/REC-xml/#sec-origin-goals.

[78] So existiert bspw. kein XML-Format für Bitmap-Grafiken, da die Kodierung einzelner Bits mit XML den benötigten Speicherplatz um ein vielfaches anwachsen lassen würde, während die Verarbeitungsgeschwindigkeit bedingt durch den zusätzlichen Parsing-Aufwand sinken würde, ohne dass eine solche Kodierung gegenüber einem (standardisierten) binären Format Vorteile bringen würde.

[79] Dies lässt sich zwar dadurch emulieren, dass Dokumente und deren Inhalt über Links, Anker oder IDs referenziert werden, erfordert jedoch eine zusätzliche Verarbeitungslogik (bspw. *XPath*), und verursacht somit u.a. einen erhöhten Ressourcenbedarf.

[80] Bspw. durch Verwendung der Prozessierungsmöglichkeiten, die XSL und andere in XML definierte Programmiersprachen bieten.

fein granulierte Komponentendefinition, da es für den Austausch großer, nicht-hierarchisch organisierter Datenmengen ungeeignet ist.

Das in GATE genutzte *FeatureMap*-Konzept erlaubt hingegen einen objektorientierten Datenaustausch zumindest während der Ausführung eines Workflows – die Definition einer Austausch-Schnittstelle im Sinne von Szyperski (vgl. Seite 45 ff. dieser Arbeit) ist jedoch aufgrund fehlender Möglichkeiten zur Restriktion komplexer Datentypen nicht umsetzbar, zudem fehlt dem System ein Persistenzmechanismus, der für eine Speicherung derartiger Datentypen verwendet werden kann, so dass sämtliche Daten im Arbeitsspeicher vorgehalten werden müssen.

Zusammenfassend lässt sich daher festhalten, dass die in diesem Kapitel untersuchten Frameworks in verschiedenen Domänen der Korpuslinguistik unterschiedlich gut einsetzbar sind. Die Architektur von UIMA erlaubt eine flexible Integration in produktive Umgebungen ebenso wie die Integration existierender Werkzeuge, das Komponenten- und Annotationsmodell ist für tokenbasierte, datenorientierte Verarbeitung geeignet. Letzteres gilt auch für GATE, wobei hier die Typisierung, die in UIMA durch Verwendung der CAS möglich wird, fehlt – dafür ist die Entwicklung neuer Komponenten in GATE deutlich einfacher als in UIMA. TextGrid wiederum eignet sich für den Einsatz in klassischen geisteswissenschaftlichen Domänen, um dort den wissenschaftlichen Austausch zu vereinfachen und um domänenspezifische Dienste zu erweitern. Den in Kapitel 2 aufgeführten Anforderungen an eine Umgebung für die Entwicklung und Evaluation experimenteller Verfahren entspricht jedoch keines der vorgestellten Frameworks – dies ist größtenteils auf die Art der Datenverwaltung, aber auch auf die Form der Definition von Komponentenschnittstellen zurückzuführen.

Im folgenden Kapitel wird daher das Framework *Tesla*, welches ebendies ermöglicht und somit eine Alternative zu den beschriebenen Frameworks bietet, in Konzept und Implementation vorgestellt. U.a. wird dabei gezeigt, dass der Einsatz eines standardisierten Austauschformates (wie CAS oder TEI) innerhalb eines Komponentenframeworks nicht zwingend benötigt wird, und dass ein objektorientierter, programmatischer Ansatz die oben geschilderten Unzulänglichkeiten umgehen kann. Zwar wird auch in Tesla XML verwendet – dies beschränkt sich jedoch auf geeignete Einsatzgebiete, wie etwa die Kodierung von Konfigurationsparametern, oder den Export von Daten zur Weiterverarbeitung in anderen Systemen.

4 Tesla

> The little engine labors and grows, performs more and more involved operations, becomes sensitive to ever subtler influences and now there manifests itself in the fully developed being [...].
>
> *(Nikola Tesla)*

Nachdem im letzten Kapitel gezeigt wurde, dass der in Frameworks wie GATE oder UIMA verfolgte Ansatz nicht zur Integration komplexer Komponenten geeignet ist, wird in diesem Kapitel das *Text Engineering Software Laboratory* (Tesla) vorgestellt, dessen Konzepte und Architektur dies (u.a.) ermöglichen. Tesla wird seit 2005 an der Sprachlichen Informationsverarbeitung am Institut für Linguistik der Universität zu Köln entwickelt. Ein Ziel der Entwicklung bestand unter anderem darin, die in Abschnitt 3.4 zusammengefassten Unzulänglichkeiten zu vermeiden und eine Plattform zur Verfügung zu stellen, in die beliebig komplexe Komponenten, Datenstrukturen und Zugriffsmethoden integriert werden können, ohne dass auf Framework-spezifische Datenmodelle zurückgegriffen werden muss. Zu diesem Zweck musste ein alternatives Konzept entwickelt werden, welches ohne derartige Einschränkungen auskommt, während gleichzeitig der Umstand berücksichtigt werden musste, dass der Implementationsaufwand bei der Entwicklung einer neuen Komponente möglichst gering bleibt. Insbesondere der letzte Punkt ist dabei relevant, wie auch Götz & Suhre (2004), Mitentwickler des UIMA-Frameworks, beobachten:

> At one extreme data modeling could be left entirely to the individual components, and data could simply be transferred between components without any knowledge about the structure of the data. This is the most flexible approach because there are no restrictions on the kind of data that can be modeled. But there are drawbacks. The framework can offer no support for handling the data. Moreover, APIs and tools for dealing with the data must be provided by the individual components.

Als Konsequenz aus dieser Überlegung zogen Götz & Suhre (2004) den Schluss, dass ein Komponentenframework primitive Datenstrukturen anbieten müsse, die zu komplexeren Datenstrukturen kombiniert werden können (wie in UIMA der Fall) – doch auch wenn ein solcher Ansatz im Vergleich zu GATE eine größere Flexibilität bei der Modellierung eigener Datenstrukturen ermöglicht, können die Möglichkeiten, die von einer modernen, objektorientierten Programmiersprache angeboten werden, nicht vollständig ausgeschöpft werden (vgl. Abschnitt 3.2.5).

Werden jedoch die Techniken, die eine Programmiersprache wie Java bietet, genauer analysiert, und wird gleichzeitig der Anspruch darauf, ein Framework zu entwickeln, welches sämtliche Programmiersprachen unterstützt, zurückgestellt, so zeigt sich, dass es grundsätzlich möglich ist, eine Architektur zu implementieren, die es Entwicklern erlaubt, beliebige Datenstrukturen und Zugriffsmethoden zu realisieren. Als Beispiel aus einer anderen Domäne sei hier das Persistenz-Framework *Hibernate*[81] genannt, das demonstriert, wie *plain old java objects* (POJOs) auf eine relationale Datenbank abgebildet werden können, indem Java-Klassen durch *Java Annotationen* ausgezeichnet werden. Allerdings zeigt dieses Beispiel auch, dass der von Götz & Suhre (2004) erwähnte Mehraufwand u.U. relativ groß ist, da die Verwendung von Hibernate Kenntnisse über relationale Datenbanken erfordert und beispielsweise die Optimierung von Datenbankabfragen eine intensive Einarbeitung in SQL und die *Hibernate Query Language* (HQL) voraussetzt. Alternative Persistenzframeworks, wie beispielsweise die Objekt-Datenbank *db4o*, sind zwar einfacher zu nutzen, verlangen von Entwicklern jedoch ebenfalls die Verwendung frameworkspezifischer Konzepte und Techniken. Auch wenn der Mehraufwand hier geringer als bei Verwendung von Hibernate ist, bleibt er höher als bei Einsatz eines ins Framework integrierten Persistenz- und Abfragemechanismus, wie im Falle von GATE.

In Bezug auf die innerhalb eines linguistischen Komponentenframeworks anfallenden Daten ergibt sich bei der Wahl eines geeigneten Persistenzframeworks zudem das Problem, dass eine große Varianz bezüglich der Komplexität einzelner Datenstrukturen ebenso wie bezüglich der Quantität dieser Daten berücksichtigt werden muss: Während beispielsweise ein Tokenizer eine hohe Zahl sehr einfacher Annotationen produziert, verhält es sich bei einer Clustering-Komponente umgekehrt[82] – je nach Art einer Komponente bietet jedes Persistenzframework unterschiedliche Vor- und Nachteile für die Speicherung von und die Suche nach Daten. Aus diesem Grund wurde die IO-Schnittstelle von Tesla in Form *lin-*

[81] vgl. auch Abschnitt 4.2.1.1. Eine ausführliche Diskussion des Frameworks würde den Rahmen dieser Arbeit sprengen, für eine Einführung sei deshalb auf Minter & Linwood (2005) verwiesen.
[82] vgl. auch Abschnitt 4.1.4 und 4.2.1.

guistischer Rollen implementiert, wodurch von konkreten Persistenzmechanismen ebenso wie von konkreten Datentypen abstrahiert werden konnte. Dies wird zusammen mit weiteren Konzepten von Tesla in Abschnitt 4.1 diskutiert. Der restliche Aufbau dieses Kapitels ähnelt strukturell der Analyse der in Kapitel 3 vorgestellten Systeme: Im Anschluss an Abschnitt 4.1 wird in Abschnitt 4.2 die Implementation von Komponenten aus technischer Sicht diskutiert, um in Abschnitt 4.3 zentrale technische Aspekte der Tesla-Architektur vorzustellen.

4.1 Konzepte eines virtuellen Labors

Wie das Akronym *Tesla* bereits vermuten lässt, ist das Labor-Konzept ein wesentliches Element des Systems: Im Gegensatz zu produktiven Umgebungen, in denen fest definierte computerlinguistische Werkzeuge wiederholt auf (neue) Daten angewendet werden, wird hier eine laborative, experimentelle Vorgehensweise fokussiert. Während Frameworks wie UIMA oder TextGrid hauptsächlich zur Integration existierender und etablierter Werkzeuge entwickelt wurden (vgl. Abschnitt 3.2 und 3.3), werden in Tesla sowohl die Entwicklung neuer Komponenten als auch die Analyse experimenteller Versuchsanordnungen in den Vordergrund gestellt[83]. Ein weiteres Ziel der Entwicklung war, eine Umgebung zu schaffen, die dem Anspruch einer Nachvollziehbarkeit durch wissenschaftlichen Austausch gerecht wird.

Während eine Laborumgebung in den Naturwissenschaften als unverzichtbar gilt (u.a. aufgrund möglicher Gefahren, die von den untersuchten oder verwendeten Substanzen ausgehen, und der Anforderungen an die Umgebungsbedingungen), ist die Notwendigkeit einer solchen Umgebung in der Domäne der Computer- und Korpuslinguistik weniger offensichtlich: Algorithmen werden nicht durch externe Einwirkungen beeinflusst, ihre Ausführung ist weder mit Gefahren noch mit hohen Kosten verbunden, und die digital vorliegenden Untersuchungsgegenstände sind verlustfrei reproduzierbar. Zudem gibt es für eine Forschergruppe keinen Grund, ein Experiment erneut auszuführen, um die Ergebnisse zu verifizieren (so lange die verwendeten Algorithmen deterministisch arbeiten). Dennoch gibt es einige Vorzüge naturwissenschaftlicher Laborumgebungen, die auch in die hier untersuchte Domäne übernommen werden könnten:

- Naturwissenschaftliche Experimente werden (soweit möglich) unter Verwendung standardisierter oder etablierter Werkzeuge und Prozeduren ausgeführt, was den Versuchsaufbau vereinfacht.

[83] Der Begriff *Experiment* im Kontext von Tesla wird in Abschnitt 4.1.2 genauer definiert.

- Ein Labor wird von einer Vielzahl von Forschern benutzt, wodurch die Kosten der Infrastruktur sinken, während der wissenschaftliche Austausch gesteigert wird.

- Das Protokoll (oder Laborbuch) und die Beschreibung des Versuchsaufbaus erlauben den Austausch von Wissen und Techniken innerhalb der jeweiligen Domäne und können verwendet werden, um ein Experiment zu revidieren.

Sowohl Standards als auch etablierte Werkzeuge werden selbstverständlich auch in der Softwareentwicklung eingesetzt, und ebenso wie in jeder anderen wissenschaftlichen Domäne werden auch in der Computerlinguistik Ergebnisse veröffentlicht. Dennoch können Experimente (oder deren Analysen) oftmals nicht oder nur schwer durch Dritte revidiert werden, da die Beschreibung eines Versuchsaufbaus für diesen Zweck oftmals zu vage oder zu komplex für eine Reimplementation ist, oder da die Auswahl der analysierten Daten nicht nachvollziehbar ist (vgl. auch Abschnitt 4.1.1). Durch Kombination verschiedener Ansätze, die Korpora, Versuchsaufbau und Standardisierung betreffen und in den folgenden Abschnitten vorgestellt werden, wird in Tesla versucht, den wissenschaftlichen Austausch zu vereinfachen und die entsprechenden Konzepte aus der naturwissenschaftlichen Forschung zu übernehmen. Dazu gehört bspw., dass die von einer Komponente generierten Daten nach erfolgter Speicherung nicht weiter modifiziert werden dürfen, da dies die Nachvollziehbarkeit der Ergebnisse beeinträchtigen würde.

Die Kosten einer Infrastruktur, die in der Computerlinguistik verwendet wird, sind zwar relativ gering (verglichen mit der in naturwissenschaftlichen Labors benötigten Ausstattung), doch stellen manche Algorithmen Anforderungen an Arbeitsspeicher und/oder Rechenleistung, die von Desktop-PCs nicht erfüllt werden können. Tesla wurde daher als ein Client-Server-System entwickelt, in welchem Experimente clientseitig definiert, aber von einem zentralen Server ausgeführt werden, so dass ein einzelner leistungsfähiger Rechner von mehreren Wissenschaftlern geteilt werden kann. Die zugrundeliegende Architektur wird zusammen mit weiteren Implementationsdetails in Abschnitt 4.2 erläutert.

4.1.1 Open Access

Ein infrastrukturelles Problem, das sowohl naturwissenschaftliche, aber auch geisteswissenschaftliche Forschung erschwert, betrifft den Zugang zu Publikationen und Forschungsergebnissen. So sind bspw. Veröffentlichungen in Fachzeitschriften und -büchern häufig daran gebunden, dass die entsprechenden Artikel nicht auf anderen Wegen zugänglich gemacht werden dürfen – die Beschaffung einer (möglicherweise für die eigene Forschung

irrelevanten) Publikation ist dadurch häufig mit hohen Kosten verbunden[84] – gleichzeitig werden die betroffenen Forschungsprojekte oftmals mit öffentlichen Mitteln finanziert, wodurch die Akzeptanz dieses Modells sinkt. Zwar stehen zunehmend alternative Publikationsformen zur Verfügung, um dieses Problem zu beheben[85]; der Zugang zu ausgewerteten Daten oder genauen Beschreibungen der angewandten Verfahren wird auf diese Weise jedoch nicht vereinfacht. Pedersen (2008) veranschaulicht dies in einem lesenswerten (und frei zugänglichem) Artikel anhand eines fiktiven Taggers, der einschließlich hervorragender Ergebnisse in einer Publikation beschrieben wird, jedoch trotz intensiver Bemühungen weder zu beziehen noch zu reimplementieren ist. Pedersen (2008, S. 470) schließt daraus:

> If we believe in empirical methods and the value of comparisons and experimental studies, then we must also believe in having access to the software that produced those results as a necessary and essential part of the evidentiary process. Without that we are asked to re-implement methods that are often too complicated and underspecified for this to be possible, or to accept the reported results as a matter of faith. [...R]eleasing software that makes it easy to reproduce and modify experiments should be an essential part of the publication process, to the point where we might one day only accept for publication articles that are accompanied by working software that allows for immediate and reliable reproduction of results.

Bei der Entwicklung von Tesla wurde diese Kritik aufgegriffen und es wurde versucht, die Architektur des Frameworks so aufzubauen, dass sowohl einzelne Komponenten als auch individuelle Zusammenstellungen von Komponenten in Experimenten zwischen Anwendern ausgetauscht werden können, wie in den folgenden Abschnitten (insbesondere in 4.1.2 und 4.1.4) beschrieben wird. Damit ist Tesla anschlussfähig an und integrierbar in Community-Plattformen wie *MyExperiment* und *Taverna*[86], die auf den Austausch von *Workflows* zwischen Forschergruppen und Anwendern spezialisiert sind.[87]

[84] Der Preis der PDF-Version der ersten Publikation über das hier vorgestellte Framework in Hermes & Schwiebert (2010) beträgt bspw. knapp 25 Euro; vor der Veröffentlichung mussten die Autoren sämtliche Rechte an dem Artikel an den Verlag abtreten. Es stehen jedoch mit Hermes (2011) und dieser Arbeit zwei kostenlos verfügbare Publikationen bereit, die im Umfang weit über den erwähnten Artikel hinausgehen, ein Kauf des Artikels ist daher unnötig.

[85] Wie beispielsweise die auf http://www.open-access.net/ nach Fächergruppen kategorisierten Portale – vgl. auch Hermes 2011, Kapitel 2.3 für eine ausführlichere Diskussion dieses Problems im Kontext eines Frameworks zur Textprozessierung.

[86] Siehe http://www.myexperiment.org/ bzw. http://www.taverna.org.uk/.

[87] Eine Integration wurde jedoch bisher nicht angestrebt, da sich Tesla zum einen noch in der Entwicklung befand, zum anderen aber auch die Akzeptanz dieser Plattformen bisher noch relativ gering ist (so

4.1.2 Experimente

Zweck eines Experimentes ist es i.d.R., Zusammenhänge zwischen Variablen aufzudecken, Hypothesen zu testen und ggfs. zu widerlegen, oder neue Kenntnisse über die untersuchte Domäne zu erlangen. In Bezug auf Korpuslinguistik kann diese Beschreibung weiter spezifiziert werden: Hier werden Experimente entweder durchgeführt, um Eigenschaften neuer Korpora mit etablierten Methoden aufzudecken, um neue Methoden zu entwickeln oder existierende Methoden zu verbessern, oder um die aus Veröffentlichungen entnommenen Schlussfolgerungen nachzuvollziehen und somit u.U. die Etablierung neuer Analysemethoden voranzutreiben. Im Gegensatz zu Experimenten in bspw. der Sozialwissenschaft sind Experimente hier häufig deterministisch, in jedem Fall aber unbeeinflusst durch kulturelle Unterschiede oder sonstige äußere Einflüsse, so dass sie bei jeder Ausführung zu gleichen oder (im Falle von nicht-deterministischen Verfahren) sehr ähnlichen Ergebnissen führen sollten.

Innerhalb von Tesla definiert ein Experiment sowohl den Versuchsaufbau des virtuellen Labors als auch das Protokoll der Ausführung: Analysierte Korpora, verwendete Komponenten und deren individuelle Konfiguration werden im Protokoll festgehalten, so dass – bedingt durch obige Beobachtung und unter der Voraussetzung, dass die verwendeten Komponenten und Korpora zur Verfügung stehen – sämtliche Ergebnisse und damit auch die Schlussfolgerungen aus dem Experiment rekonstruiert werden können. Ein Tesla-Experiment kann als ein Workflow-Diagramm betrachtet werden, in dem die einzelnen Komponenten auf Basis ihrer individuellen Anforderungen, aber auch auf Basis der zu untersuchenden Hypothese verbunden sind, wie Abbildung 4.1 zeigt.

Wird ein Experiment ausgeführt, werden die Abhängigkeiten, wie in der Abbildung dargestellt, aufgelöst, und die einzelnen Komponenten werden gestartet, sobald die von ihnen benötigten Daten verfügbar sind. Dies führt u.a. zu einer einfachen Form von paralleler Verarbeitung, da Komponenten, zwischen denen keine Abhängigkeiten bestehen (wie *Berkeley Parser* und *ABL Align* in Abbildung 4.1), automatisch parallel ausgeführt werden. Ist die Prozessierung einer Komponente abgeschlossen, so wird der Abhängigkeitsgraph erneut analysiert, bis schließlich sämtliche Komponenten verarbeitet worden sind (Abschnitt 4.1.5 bietet eine vertiefende Darstellung des Lebenszyklus und der Ausführung von Komponenten).

Ein deutlich größerer Geschwindigkeitszuwachs entsteht jedoch dann, wenn ein Experiment mit leicht variierten Parametern oder um zusätzliche Komponenten erweitert

waren bspw. am 21. Juni 2011 bei *MyExperiment* ca 4.700 Nutzer mit insgesamt 1.800 Workflows registriert), so dass dieser Punkt bisher nur eine geringe Priorität hatte.

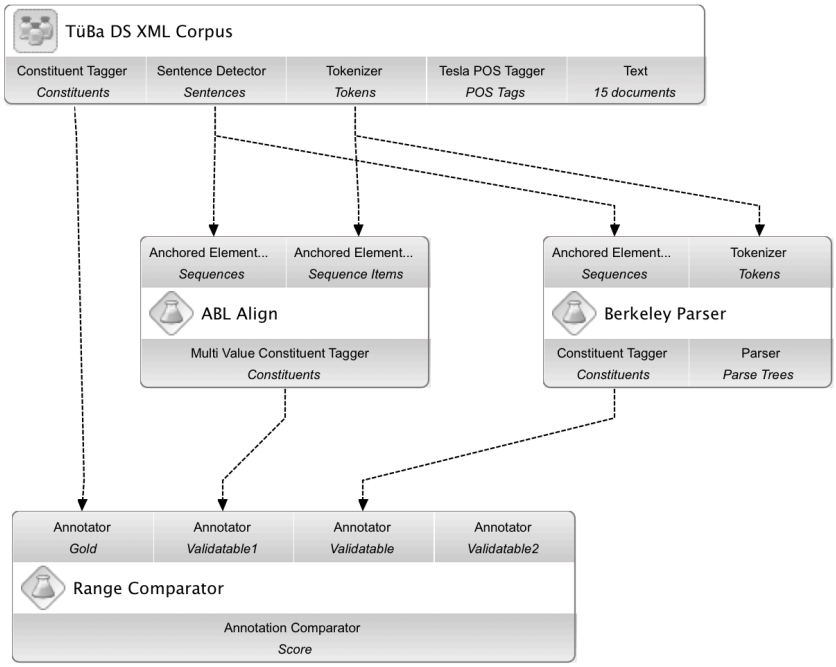

Abbildung 4.1: Beispiel eines Versuchaufbaus in Tesla (Screenshot des graphischen Editors). Die Komponenten *Berkeley Parser* und *ABL Align* erzeugen u.a. (Hypothesen über) syntaktische Strukturen, die hier von der Komponente *Range Comparator* anhand der manuellen Auszeichnungen im *TüBa-D/S*-Korpus evaluiert werden (vgl. auch Kapitel 5).

ausgeführt wird: In diesem Fall werden nur die Komponenten, deren Konfiguration geändert wurde, sowie Komponenten, die von diesen abhängig sind, erneut prozessiert. Eine Änderung der Konfiguration der *ABL Align*-Komponente in Abbildung 4.1 würde bspw. dazu führen, dass bei einer erneuten Ausführung des Experiments nur *ABL Align* und *Range Comparator* prozessiert werden müssten. Da die Analyse der Abhängigkeiten von Komponenten experimentübergreifend durchgeführt wird, kann auf die Prozessierung von Komponenten u.U. auch dann verzichtet werden, wenn ein neu erzeugtes Experiment zum ersten Mal verarbeitet wird.

Dieses Vorgehen setzt voraus, dass sämtliche von einer Komponente produzierten Daten vollständig persistiert werden, was als Kritikpunkt des Ansatzes gesehen werden kann, da es dazu führen kann, dass der von einer Komponente benötigte Speicherplatz und die für ihre Ausführung benötigte Laufzeit stark ansteigen – die bei der Vorverarbeitung des *British National Corpus* (vgl. folgendes Kapitel) anfallende Datenmenge liegt bspw. bei ca. 5 GB, was ungefähr dem 9-fachen des (komprimierten) Ausgangsmaterials entspricht. Allerdings ist es unter dem Aspekt der Nachvollziehbarkeit von Experimenten notwendig, die

Daten vollständig vorzuhalten, zudem vermeidet der hier vorgestellte Ansatz redundante Speicherung, so dass die Daten im Idealfall lediglich einmal prozessiert und gespeichert werden. Um die negativen Auswirkungen dieses Vorgehens auf die Ausführungsgeschwindigkeit des Systems zu verringern, wurde der Persistenzmechanismus asynchron implementiert: Von einer Komponente produzierte Datenstrukturen werden zunächst in einem Cache zwischengespeichert, der durch einen parallel ausgeführten Thread in die jeweilige Datenbank übertragen wird, so dass die von einer Komponente benötigte Laufzeit kaum durch den Persistenzmechanismus beeinflusst wird.[88]

4.1.3 Korpora

Wie bereits in Kapitel 1 erwähnt, sind die Korpora, auf deren Basis Experimente durchgeführt werden, für die Nachvollziehbarkeit von Hypothesen und Schlussfolgerungen äußerst relevant – insbesondere dann, wenn die Qualität der Ergebnisse einer Komponente auf korpusspezifische Eigenschaften zurückgeführt werden kann, oder wenn annotierte Korpora verwendet werden, deren Annotationen für die Ausführung eines Experiments benötigt werden. Dies ergibt weitere Anforderungen an ein Komponentensystem: So muss es beliebig große Korpora verarbeiten können, sowohl annotierte als auch nicht ausgezeichnete Dokumente in unterschiedlichen Zeichenkodierungen unterstützen und schließlich eine Möglichkeit bieten, den wissenschaftlichen Austausch auch dann zu unterstützen, wenn die verwendeten Korpora urheberrechtlich geschützt sind und eine Weiterverbreitung der Texte untersagt ist. Techniken wie *Corpus Masking*, durch die Texte derart chiffriert werden, dass der copyright-geschützte Originaltext nicht rekonstruierbar ist, während statistische Merkmale der Texte (teilweise) erhalten bleiben (vgl. Rehm *et al.* 2007), bieten nur in Einzelfällen eine akzeptable Möglichkeit, urheberrechtliche Einschränkungen zu umgehen, denn dadurch, dass verschiedene linguistische Eigenschaften (wie etwa subsymbolische Informationen zu morphologischen Kategorien) verloren gehen, ist eine solche Technik in vielen Fällen ungeeignet. Ferner bieten online verfügbare Korpora oftmals nur eingeschränkten Zugriff auf die zugrundeliegenden Texte, etwa in Form von Dokumentausschnitten, die den Kontext einer Suchanfrage darstellen. Bei einer Untersuchung fünf deutschsprachiger Korpora mussten Duffner & Näf (2006, S. 11) feststellen, dass die Suchergebnisse in allen Fällen auf den unmittelbaren Kontext der Suchanfrage beschränkt waren und nicht

[88] Dies gilt allerdings nur, falls eine Komponente nicht deutlich schneller Daten produziert, als sie von der gewählten Datenbank gespeichert werden können. Andernfalls sorgt das System dafür, dass der Cache geleert wird, bevor neue Daten abgelegt werden können, um so sicherzustellen, dass stets genügend Arbeitsspeicher verfügbar ist. Die Auswahl einer geeigneten Datenbank ist somit ein wesentlicher Punkt bei der Konzeption einer neuen Komponente und wird in Abschnitt 4.2 separat diskutiert.

über wenige Sätze hinausgingen. Während dies für manuelle Suchanfragen oftmals ausreicht (und in solchen Fällen die Benutzbarkeit des Systems vereinfachen kann), sind die zugrundeliegenden Daten innerhalb eines Frameworks für Computerlinguistik nicht universell nutzbar, weil sämtliche dokumentbezogenen Komponenten zwangsläufig an diesen Einschränkungen scheitern müssen.

Da Projekte wie das in Abschnitt 3.3 beschriebene *TextGrid* sich dieses Problems zumindest dahingehend annehmen, dass eine Infrastruktur entwickelt wird, die von individuellen Datenquellen abstrahiert (ohne allerdings die oben erwähnten juristischen Probleme zu lösen, und ohne einen dokumentbasierten Ansatz zu realisieren), wurden bei der Konzeption von Tesla ausschließlich solche Korpora berücksichtigt, bei denen ein direkter Zugriff auf die Dokumente möglich ist. Dabei wird zwischen unterschiedlichen Datenquellen einerseits und unterschiedlichen Datenformaten andererseits unterschieden, wobei das System in beiden Fällen durch Implementation eines Interfaces bzw. Extension einer Basisklasse erweiterbar bleibt und an neue Datenquellen und -formate angepasst werden kann: Durch Implementation des Interfaces `de.uni_koeln.spinfo.tesla.datasource.IDocumentProvider` können, wie im Klassendiagramm in Abbildung 4.2 dargestellt, Zugriffsmöglichkeiten auf weitere Datenquellen umgesetzt werden, um etwa die Inhalte von Onlineressourcen oder proprietäre Datenbanken adressieren zu können. Im Gegensatz zu dem in GATE umgesetzten Ansatz (vgl. Abschnitt 3.1.1) wird hier nicht versucht, proprietäre Eigenschaften verschiedener Datenquellen in einem einheitlichen Format abzubilden, die Schnittstelle ist vielmehr auf allgemeine Dokument-Operationen (wie etwa die Auflistung aller Dokumente oder den Zugriff auf den unprozessierten Inhalt eines Dokuments) beschränkt. Die Interpretation des Inhaltes eines Dokuments wird in einem weiteren Schritt durchgeführt, der in Abschnitt 4.1.3.1 vorgestellt wird.

Mittels einer Konfigurationsdatei können anschließend die Adresse der Datenquelle (bspw. eine URL oder ein relativer Pfad im Dateisystem, siehe auch Listing C.1 in Anhang C) ebenso wie weitere, eventuell notwendige Parameter bestimmt werden, so dass ein DocumentProvider für unterschiedliche Korpora eingesetzt werden kann. Standardmäßig bietet Tesla durch die Klasse `de.uni_koeln.spinfo.tesla.datasource.zip.ZipDocumentProvider` Zugriff auf Dokumente in serverseitig vorliegenden, komprimierten .zip-Dateien (wie etwa den Daten des BNC- und verschiedener Tübinger Baumdatenbanken[89]), und mit der Klasse `de.uni_koeln.spinfo.tesla.datasource.TeslaDocumentProvider` einen DocumentProvider für eine editierbare, eingebaute Da-

[89] Siehe http://www.sfs.uni-tuebingen.de/en/corpora.shtml – in Kapitel 5 werden diese Korpora ausführlicher vorgestellt.

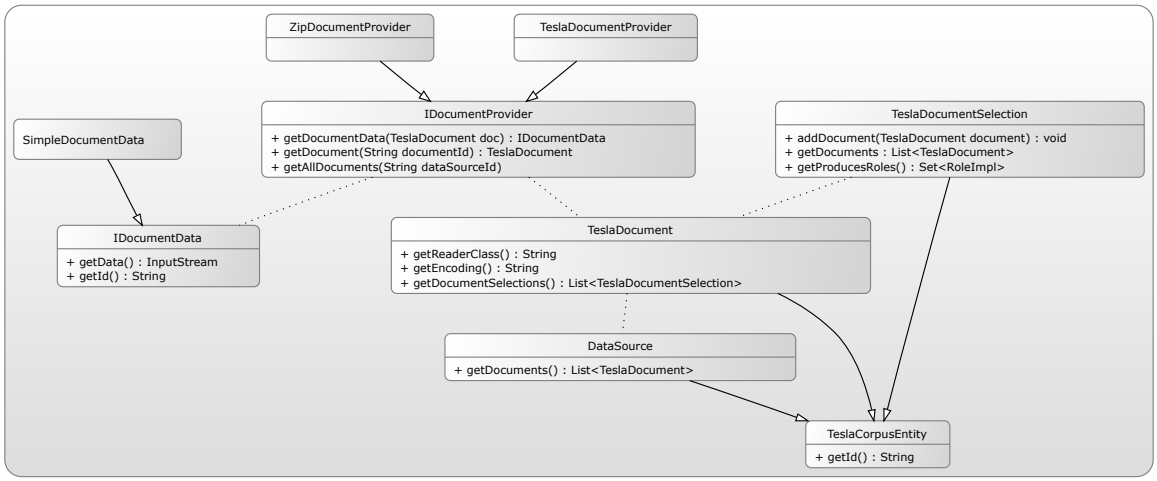

Abbildung 4.2: Klassendiagramm der Korpus- und Dokumentverwaltung in Tesla. Durch Pfeile markiert sind Vererbungsrelationen, punktierte Linien symbolisieren Assoziationen.

tenbank, so dass dem System eigene Dokumente hinzugefügt werden können, ohne zwingend eine neue Implementation des Interfaces erstellen zu müssen.

Jede so definierte Datenquelle kann in einem Experiment verwendet werden; zusätzlich können jedoch auch einzelne Dokumente in einer *Document Selection* gruppiert werden, um so das in einem Experiment prozessierte Korpus den jeweiligen Anforderungen anzupassen.

Abbildung 4.2 zeigt ebenfalls, dass keine direkte Assoziation zwischen Dokumenten und ihrem Inhalt (bzw. den entsprechenden Konstruktionen `TeslaDocument` und `IDocumentData`) besteht: Auf den Inhalt eines Dokuments kann nur über die Methode `DocumentProvider.getDocumentData(TeslaDocument doc)` zugegriffen werden. Diese Trennung ermöglicht es, Verwaltungsoperationen, wie etwa das Erzeugen großer *Document Selections*, mit Hilfe von platzsparenden Datenstrukturen zu implementieren, und gleichzeitig die Anzahl der bei der Implementation neuer DocumentProvider erforderlichen Methoden einzuschränken, da nur die Methoden implementiert werden müssen, die für den Zugriff auf den Inhalt einer Datei benötigt werden. Die Zusammengehörigkeit von Dokument und Inhalt wird mit Hilfe eines gemeinsamen Schlüssels sichergestellt, der wiederum durch den Message-Digest-Algorithmus[90] als Prüfsumme des Dokumentinhaltes berechnet wird. Eine Kollision zweier Prüfsummen, d.h. die Berechnung identischer Prüfsummen bei unterschiedlichen Dokumentinhalten, ist zwar nicht ausgeschlossen, aber

[90] vgl. http://de.wikipedia.org/wiki/Md5.

sehr unwahrscheinlich[91] – der Vorteil dieses Verfahrens liegt jedoch umgekehrt darin, dass die Prüfsummen identischer Dokumente trivialerweise ebenfalls identisch sind: So kann sichergestellt werden, dass bei der Ausführung eines Experiments in einer anderen Tesla-Installation genau die Dokumente prozessiert werden, die auch auf der Ursprungsinstallation verwendet wurden. Zwar löst dieses Vorgehen nicht *per se* das Problem der Nachvollziehbarkeit von Experimenten, jedoch kann es dieses auf ein rein rechtliches Problem reduzieren, denn unter der Voraussetzung, dass der Empfänger eines Tesla-Experiments über die referenzierten Dokumente verfügt, kann das Experiment ohne weitere Einschränkungen ausgeführt werden.

4.1.3.1 Reader

Da Tesla als ein Framework entwickelt wurde, welches nicht auf die Verarbeitung von Texten beschränkt sein soll, ist, wie das Diagramm in Abbildung 4.2 zeigt, der Inhalt eines Dokuments nicht als Sequenz von Zeichen, sondern als `java.io.InputStream` repräsentiert, durch den auf die zugrundeliegenden Bytefolgen zugegriffen werden kann. Diese werden zur Laufzeit von einem *Reader* interpretiert und im Falle von Text-Dateien in eine Darstellung als UTF-8-kodierte Zeichenketten überführt, die anschließend auf generische Art vom Framework verwendet werden können (vgl. auch Abbildung 4.3). Andere Formate, wie etwa MIDI Sequenzen, Video- oder Audiostreams, könnten ebenfalls durch Implementation eines geeigneten Readers verarbeitet werden. Um die Analogie zu dem von Bird & Liberman (2001) als *linguistic signal* bezeichneten Konzept zu verdeutlichen, wird der Inhalt eines Dokuments im Folgenden ebenso wie im Quellcode des Programms als *Signal* bezeichnet.

Reader stehen zwangsläufig am Anfang des in einem Experiment definierten Versuchsaufbaus, da sie für die Extraktion der zu untersuchenden Daten verantwortlich sind. Zusätzlich zu dieser Funktion können Reader allerdings auch als Komponenten (vgl. den folgenden Abschnitt 4.1.5) fungieren und bspw. das von ihnen produzierte Signal um Annotationen anreichern.[92] Neben Annotationen können Reader zudem Dokument-Metadaten[93] generieren, die auch in der graphischen Oberfläche dargestellt werden (wie

[91] Da die generierten Schlüssel jeweils 128 Bit lang sind, kann es bei ihrer Generierung theoretisch zu Kollisionen kommen – in einem solchen Fall würden zwei Dokumenten unterschiedlichen Inhaltes identische Schlüssel zugewiesen. Bei fast 10^{40} möglichen Schlüsseln ist dies jedoch äußerst unwahrscheinlich.

[92] vgl. Abbildung 4.1 – der dort dargestellte `TuebaXmlReader` generiert u.a. *Token*-, *Sentence*- und *POS*-Annotationen.

[93] Das verwendete Metadaten-Set folgt den im *Dublin Core*-Standard (vgl. http://dublincore.org/) vorgeschlagenen Kernauszeichnungen wie etwa *Autor*, *Sprache* oder *Datum*.

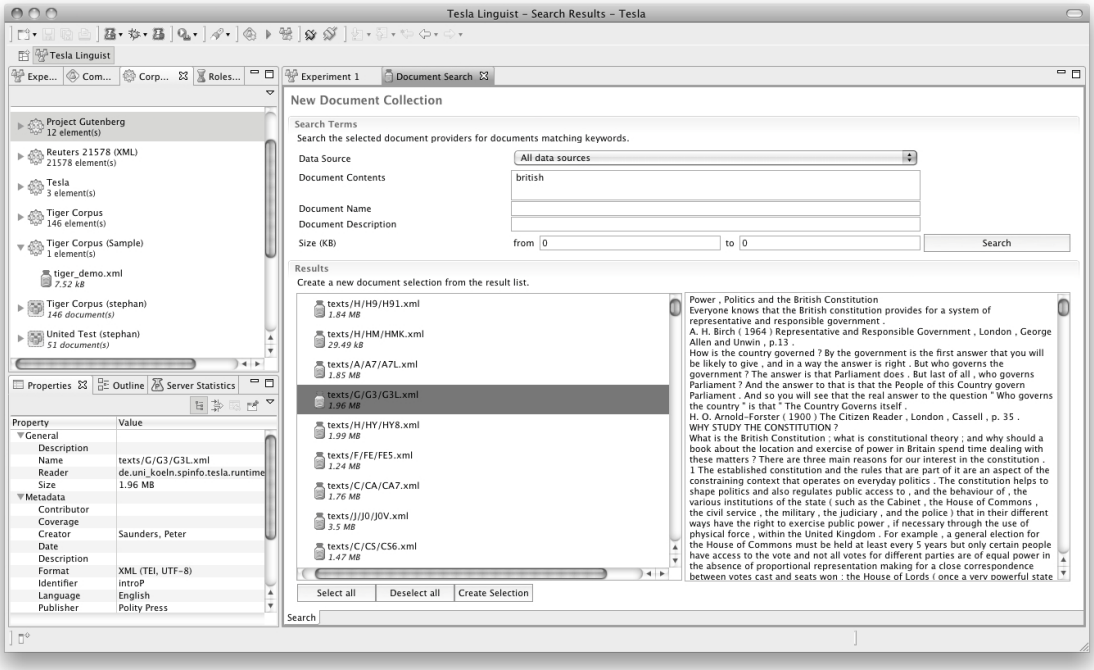

Abbildung 4.3: Screenshot der Korpus-Suchmaske in Tesla. Die Abbildung veranschaulicht die von Readern vorgenommene Trennung von Inhalt und Auszeichnung eines Dokuments am Beispiel des BNC: In der Detailansicht wird ausschließlich der Inhalt eines Dokuments dargestellt, während im Properties-Bereich dokumentspezifische Metadaten angezeigt werden. Die im BNC enthaltenen Annotationen werden nicht dargestellt, stehen jedoch für die Prozessierung durch weitere Komponenten zur Verfügung.

etwa im *Properties*-Bereich in Abbildung 4.3).

Dies ermöglicht sowohl eine Trennung als auch eine Kombination von Daten und ggfs. bereits vorhandenen, proprietären Auszeichnungen, wie sie bspw. im Falle des deutschsprachigen TüBa-D/S-Korpus oder des englischen British National Corpus vorhanden sind. Letzteres besteht aus ca. 100 Millionen Wörtern, die um POS-Tags und Lemmata angereichert wurden[94], während im (mit ca. 360.000 Wörtern deutlich kleineren) TüBa-D/S-Korpus[95] zusätzlich noch syntaktische Relationen abgebildet sind. Trotz der Gemeinsamkeiten unterscheiden sich beide Korpora: Zwar liegen die Dokumente in XML vor, jedoch wurden verschiedene XML- und Zeichenkodierungen verwendet. In Tesla übernehmen Reader die Aufgabe, diese Unterschiede zu eliminieren, indem das Signal von Annotationen bereinigt und letztere stattdessen in ein objektorientiertes Format überführt

[94] vgl. http://www.natcorp.ox.ac.uk/corpus/index.xml.
[95] vgl. http://www.sfs.uni-tuebingen.de/en/tuebads.shtml.

werden, welches von weiteren Komponenten verarbeitet werden kann. So erzeugt bspw. der BNC-Reader neben einer UTF-8-Darstellung des annotierten Textes verschiedene Annotationstypen, wie etwa Tokens, Lemmata und POS-Tags, die von anderen Komponenten weiterverarbeitet werden können. Dies ist insofern hilfreich, als dass, wie in Abbildung 4.1 gezeigt, Komponenten gleicher Funktionalität anhand der durch einen Reader gelieferten Daten evaluiert werden können, und kann ebenfalls von Vorteil sein, falls eine benötigte Komponente noch nicht existiert, ein (manuell) ausgezeichnetes Korpus jedoch bereits vorhanden ist: Die Annotationen des Korpus können in Form einer *Rolle* (das zugrundeliegende Konzept wird im folgenden Abschnitt 4.1.4 erläutert) zugänglich gemacht werden, die anschließend auch von anderen Komponenten erfüllt werden kann. Dies führt dazu, dass – anders als bei den in Kapitel 3 untersuchten Frameworks GATE und UIMA – in Tesla keine strikte Trennung zwischen Komponenten und zu verarbeitenden Daten vorgenommen werden muss, sondern dass ein Reader als eine Spezialisierung einer Komponente angesehen und entsprechend auch implementiert und verwendet werden kann, wie Abschnitt 4.1.5.3 zeigen wird. Zudem können auf diese Weise die in Cunningham & Bontcheva (2006) beschriebenen Probleme bei der Integration unterschiedlicher *Processing Resources* in das GATE-Framework (vgl. Abschnitt 3.1.1) vermieden werden.

4.1.4 Das Tesla Role System

Damit in einem datenorientierten System Informationen zwischen Komponenten ausgetauscht werden können, müssen diese in expliziter Form im verwendeten Annotationsgraphen abgelegt werden. Bei der anschließenden Verarbeitung durch weitere Komponenten müssen diese Daten dann interpretiert werden, was je nach Komplexität der kodierten Informationen unterschiedlich aufwändig ist: So legt bspw. die im Kontext von UIMA vorgestellte Definition eines POS-Tags (vgl. Listing 3.5 auf Seite 63) fest, dass ein POS-Tag die Attribute *tagsetId* und *value* enthalten muss, schränkt die möglichen Werte jedoch lediglich auf den Datentyp `String` ein und erfordert so ihre individuelle Interpretation abhängig vom verwendeten Tag-Set. Müssen komplexere Daten wie Graphen oder mehrdimensionale Arrays im Annotationsgraph gespeichert werden, nimmt nicht nur der notwendige Interpretationsaufwand weiter zu, sondern auch der Implementationsaufwand, der für die Konvertierung der Datenstrukturen in das vom Framework unterstützte Format benötigt wird – dies zeigt bspw. der Umfang der bereits erwähnten GATE-Adaption des *Stanford Parsers* (vgl. Abschnitt 3.1.5).

Zwar könnten Algorithmen, die auf komplexen Daten operieren, unmittelbar in die Komponente integriert werden, die diese Daten produziert. So würde die Notwendigkeit

der Konvertierung zwischen verschiedenen Datenformaten entfallen, jedoch würde dies dem Anspruch einer Wiederverwertbarkeit von Komponenten entgegenstehen. Zudem existiert für jede zu bearbeitende Aufgabe meist mehr als ein Lösungsansatz, wobei sich die unterschiedlichen Ansätze in ihrer Qualität, aber auch bezüglich der Anforderungen an Laufzeit und Speicherbedarf unterscheiden – es ist daher naheliegend, diese Ansätze als eigenständige Komponenten zu implementieren (vgl. die Diskussion des Komponentenbegriffs nach Szyperski *et al.* (1998) ab Seite 45 dieser Arbeit), statt sie in bestehende Komponenten zu integrieren.

Zwar können auch universelle Datenstrukturen, wie UIMAs CAS oder der TEI Standard genutzt werden, um komplexe Daten, bspw. Graphen mit mehreren Millionen Knoten und Kanten, zu speichern. Allerdings können dabei, wie in den vorangegangenen Kapiteln bereits diskutiert, verschiedene Probleme auftreten:

- U.U. ergeben sich Probleme bei Laufzeit und Speicherbedarf, etwa dann, wenn große XML-kodierte Datensätze verarbeitet werden müssen, die weder vollständig in den verfügbaren Arbeitsspeicher geladen noch effizient mittels externer Werkzeuge prozessiert werden können, für die jedoch ein wahlfreier Zugriff benötigt wird (vgl. Abschnitt 3.4).

- Der Zugriff auf implizite Informationen innerhalb der Datenstrukturen (wie etwa Kongruenz-Relationen zwischen Elementen eines Syntaxbaums, Ähnlichkeiten von alignierten Sätzen oder Distanzen zwischen Vektoren) erfordert detaillierte Kenntnisse über die interne Struktur der Daten (vgl. Abschnitt 2.3) und verlangt zudem die Implementation effizienter Query-Algorithmen, die u.U. nicht durch das Framework bereitgestellt werden können.

Bei der Konzeption und Implementation einer neuen Komponente, die derartige Daten verarbeiten soll, muss in einem solchen Fall zunächst die komponentenspezifische Terminologie und deren Semantik analysiert werden, um die notwendigen Abfragen implementieren zu können. Genügt dies nicht (weil etwa die Dokumentation der Komponente in einigen Punkten nicht hinreichend ist), muss zudem eine Analyse der verwendeten Algorithmen durchgeführt werden. Beide Punkte sind insbesondere dann relevant, wenn verschiedene Personen an der Entwicklung der betroffenen Komponenten beteiligt waren oder sind, da sich zusätzlicher Kommunikationsaufwand ergeben kann.

Das *Tesla Role System* (TRS) wurde als Alternative zu den datenorientierten Typisierungsansätzen bestehender Frameworks (vgl. Kapitel 3) entwickelt und in Hermes &

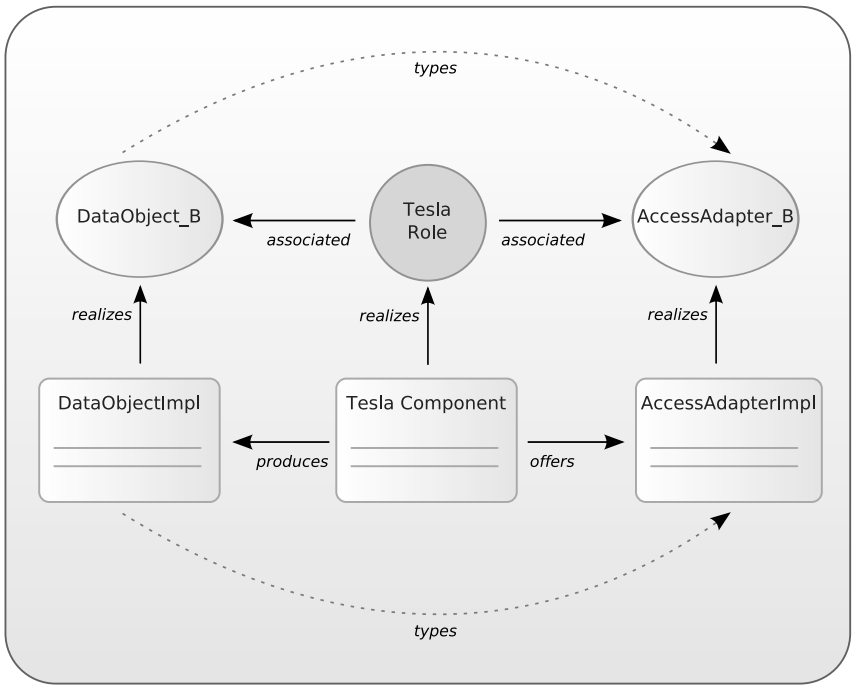

Abbildung 4.4: Schematische Darstellung des Tesla Role System (Grafik aus Hermes & Schwiebert 2010). Eine Rolle assoziiert *Access Adapter*- und *Data Object*-Interface, wobei letzteres gleichzeitig den Adapter typisiert. Eine Komponente repräsentiert eine konkrete Realisierung einer oder mehrerer Rolle(n), wobei die spezifizierten Interfaces auf beliebige Art implementiert werden können.

Schwiebert (2010) erstmals vorgestellt. Anders als bei datenorientierten Frameworks werden Annotationen im TRS nicht in Form von Datenstrukturen, sondern als Schnittstellen beschrieben, die auf beliebige Art implementiert werden können.

Die Definition einer Rolle in Tesla umfasst zunächst die Funktionalität, die bspw. auch durch die *Common Analysis Structure* von UIMA angeboten wird: Durch Aggregation primitiver Datentypen können auch hier komplexe Datentypen definiert und zum Austausch zwischen Komponenten eingesetzt werden, ebenfalls lassen sich Typ-Hierarchien definieren, die beliebig erweitert und spezialisiert werden können. Ein wesentlicher Unterschied zu datenorientierten Ansätzen besteht jedoch darin, dass zur Definition des Austauschformats auf die Möglichkeiten der objektorientierten Programmierung zurückgegriffen wird, indem dieses mit Hilfe eines Java-Interfaces und den dort definierten Methoden spezifiziert wird. Gegenüber einem datenorientierten Ansatz führt dies zu einer deutlich höheren Flexibilität: So ist es bspw. möglich, neben dem Zugriff auf explizit vorliegende Daten auch deren algorithmische Interpretation zu definieren, durch Parametrisierung von Methoden eine API-ähnliche Funktionalität anzubieten und durch Trennung von Funktionalität und zugrundeliegendem Datenmodell eine transparente Optimierung von Speicher- und Laufzeitbedarf zu erreichen und die Weiterverarbeitung von Annotationen zu erleichtern.

Damit stehen Entwicklern bei der Modellierung von Datenstrukturen sämtliche Möglichkeiten zur Verfügung, die von der Programmiersprache Java angeboten werden – je nach Anwendungsfall kann es jedoch zusätzlich erforderlich sein, auch den Zugriff auf diese Daten zu modifizieren und die generischen Methoden, die von einem Framework angeboten werden, um weitere, anwendungsspezifische Methoden zu ergänzen. Zu diesem Zweck definiert eine Rolle in Tesla ein weiteres Java-Interface, das ebendiese Funktionalität spezifiziert.

Das TRS benötigt somit kein zusätzliches Meta-Format, das die syntaktische oder semantische Form von Annotationen festlegt, sondern greift vielmehr auf die Technologien, die für die Programmiersprache Java zur Verfügung stehen, zurück, und kann so bspw. auch *Unit-Tests*, mit denen das Testen der Funktionalität einer Komponente vereinfacht wird, nutzen (vgl. Abschnitt 4.2.3). Zudem werden Entwickler bei der Konzeption neuer Austauschformate und Zugriffsmöglichkeiten nicht durch das TRS eingeschränkt: Die einzige Vorgabe besteht darin, dass die in einer Rolle kombinierten Interfaces (direkt oder indirekt) von den *System-Interfaces* `DataObject` bzw. `IAccessAdapter` abgeleitet werden müssen.

Abbildung 4.4 veranschaulicht den Zusammenhang zwischen den durch eine Rolle spezifizierten Interfaces und einer Komponente, die eine Rolle erfüllt: Die Rolle assoziiert

die beiden Interfaces (in Form einer XML-Spezifikation wie in Listing 4.1), während eine Komponente dafür verantwortlich ist, die konkreten Implementationen der Interfaces zur Verfügung zu stellen.

```
<role_system context="generic_roles">
  ...
  <role id="de.uni_koeln...PosTagger">
    <metadata>
      <name>POS Tagger</name>
      <description>Assigns POS Tags to words</description>
    </metadata>
    <produces>
      <data_object_interface>de....IPartOfSpeech</data_object_interface>
      <input_adapter_interface>de....IPartOfSpeechAccessAdapter</input_adapter_interface>
    </produces>
  </role>
  ...
</role_system>
```

Listing 4.1: Rollendefinition in Tesla. Exemplarisch dargestellt ist die Spezifikation eines POS-Taggers, die aus der Verknüpfung der Interfaces `IPartOfSpeech` und `IPartOfSpeechAccessAdapter` besteht.

Da die Implementationen von DataObject- und AccessAdapter-Interfaces keinen weiteren Beschränkungen unterliegen, und da beide Interfaces beliebig spezialisiert werden können, lässt sich mit Hilfe des TRS jede Art von Komponente realisieren. Allerdings ergibt sich daraus die Frage, wie hoch der Entwicklungsaufwand dieses Ansatzes ist, und ob die eingangs von Kapitel 4 zitierte Beobachtung von Götz & Suhre (2004) (nach der vollständige Freiheit bei der Modellierung von Datenstrukturen dazu führt, dass ein Framework keine Unterstützung für Bearbeitung und Nutzung der Daten bieten kann, siehe Seite 81) akzeptiert werden muss, bzw. welche Möglichkeiten es gibt, um den Implementationsaufwand gering zu halten. So sollte bspw. vermieden werden, dass bei der Entwicklung einer neuen Rolle und der anschließenden Implementation in einer Komponente auch die notwendigen Persistenzmechanismen neu implementiert werden müssen. Da es sich hierbei jedoch weniger um ein konzeptionelles als vielmehr um ein technisches Detail handelt (auch wenn es für die Beurteilung des Ansatzes relevant ist), soll dieser Punkt an dieser Stelle nicht weiter betrachtet, sondern stattdessen in Abschnitt 4.2.1 erneut aufgegriffen werden.

Tesla verwendet, wie auch GATE und UIMA, einen Annotationsgraphen, durch den DataObject-Implementationen mit einem (Teil-)Signal verknüpft werden können. Dies wird mit Hilfe der vom System vorgegebenen Klasse `Annotation` umgesetzt, welche als Referenz auf ein DataObject dient und Methoden bietet, mit denen ein (zusammenhän-

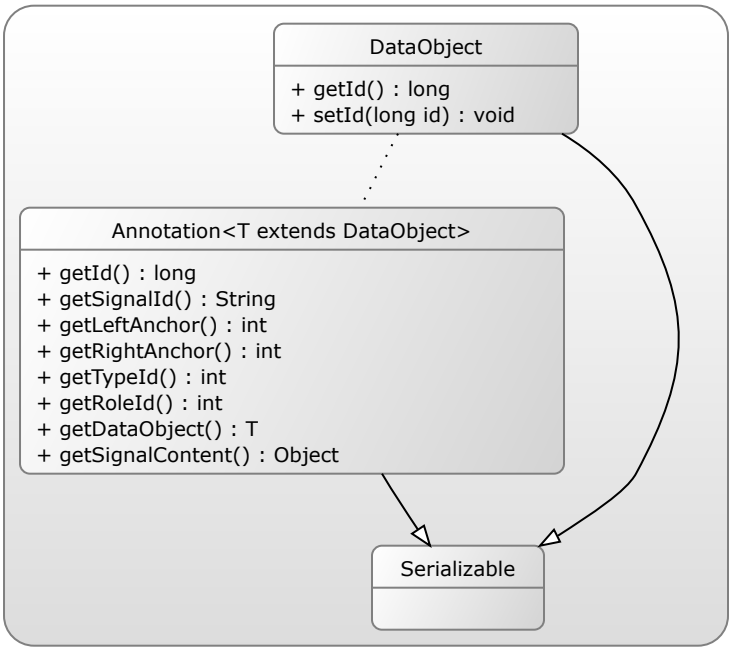

Abbildung 4.5: Assoziation zwischen Annotation und DataObject. Während es sich bei Annotation um eine von Tesla definierte Klasse handelt, kann das Interface DataObject beliebig erweitert werden. Die Assoziation zwischen zwei Instanzen wird durch Verwendung einer identischen Id hergestellt.

gender) Ausschnitt eines Signals adressiert werden kann (vgl. Abbildung 4.5). Im Gegensatz zu den in Kapitel 3 beschriebenen Frameworks ist der Annotationsgraph von Tesla jedoch keine vorgegebene Datenstruktur, sondern lediglich ein Konzept, das entsprechend den Anforderungen einer Rolle unterschiedlich umgesetzt werden kann. Die Klasse `Annotation` dient lediglich dazu, eine gemeinsame Schnittstelle zu definieren und den Umgang mit allgemeinen, auf den Annotationsgraphen bezogenen Methoden zu modellieren. So können bspw. bereichsbezogene Queries, wie der Zugriff auf sämtliche Tokens eines Satzes oder eines Paragraphen, auf generische Art anhand der Klasse `Annotation` implementiert werden, was, wie Abschnitt 4.2.1 zeigen wird, die Wiederverwertbarkeit von `IAccessAdapter`-Implementationen erhöht. Zudem führt dies dazu, dass trotz größtmöglicher Flexibilität bei der Entwicklung neuer Datenstrukturen ein Teil der von Götz & Suhre (2004) geforderten Basisfunktionalität generisch umgesetzt werden kann.

Die innerhalb der Klasse `Annotation` verwaltete Referenz auf ein DataObject wird dadurch realisiert, dass beiden Objekten bei der Speicherung die gleiche Id zugewiesen wird, so dass es nicht zwingend notwendig ist, dass ein Annotationsobjekt stets über eine Objektreferenz (im Sinne der Programmiersprache Java) auf ein DataObject verfügt, sondern dieses bei Bedarf zur Laufzeit geladen werden kann. Wie in Abschnitt 4.2.2 gezeigt

werden wird, führt dies dazu, dass unterschiedliche Umsetzungen des Annotationsgraph-Konzepts zueinander kompatibel bleiben, und insbesondere auch Querverweise zwischen Annotationen, die von unterschiedlichen Persistenz-Frameworks verwaltet werden, möglich sind.

Durch das TRS wird die Austauschbarkeit von Komponenten stark vereinfacht: Neben Signalen können ausschließlich Rollen von einer Komponente konsumiert, d.h. als Eingabe verarbeitet werden (vgl. Abbildung 4.8), und auch die von einer Komponente produzierten Daten müssen durch eine oder mehrere Rollen definiert werden. Da mehrere Komponenten die gleiche Rolle erfüllen können, und da Rollen zudem in einer Ontologie eingeordnet werden können (vgl. den folgenden Abschnitt), lassen sich die Anforderungen einer Komponente an die von ihr konsumierten Rollen auf unterschiedliche Arten erfüllen. Dies zeigt auch das in Abbildung 4.1 (Seite 87) dargestellte Experiment: Hier produzieren sowohl der *TüBa-XML-Reader* als auch die Komponenten *ABL Align* und *Berkeley Parser* die Rolle *Constituent Tagger*. Eine Komponente, die diese Rolle konsumiert (wie der in Abbildung 4.1 dargestellte *Range Comparator*), kann entsprechend mit jeder der genannten Komponenten arbeiten, und beim Aufbau eines Experiments kann, je nach Fragestellung oder Anspruch an Ergebnisqualität, Rechenzeit und Speicherbedarf, die am besten geeignete Komponente ausgewählt werden.

Ein positiver Nebeneffekt, der sich aus der Interface-basierten Definition von Rollen ergibt, liegt im Umgang mit Komponenten und Werkzeugen, die nicht frei verfügbar sind: Da Kommunikation zwischen zwei Komponenten ausschließlich über Interfaces stattfindet, welche die generierten Daten in einer *funktionalen* Art beschreiben, kann jede Rolle mit beliebiger Lizenz (re-)implementiert werden, so dass Anwender nicht auf den Einsatz spezieller Werkzeuge angewiesen sind. Das in Abschnitt 4.1.1 beschriebene Problem der Nicht-Verfügbarkeit von Werkzeugen, auf die in einer Publikation verwiesen wird, kann damit zwar nicht behoben werden, jedoch wird ein Austausch durch kompatible, verfügbare Komponenten oder durch Eigenentwicklungen auf Basis der definierten Interfaces vereinfacht.

4.1.4.1 Sub- und Superrollen

Einer der größten Vorteile des TRS ist der Umstand, dass das aus der Objektorientierung stammende Konzept der Vererbung indirekt auch für Rollen verfügbar ist, so dass es möglich ist, allgemein definierte Basisrollen zu verwenden, die von abgeleiteten Rollen weiter spezialisiert werden. Ein ähnlicher Mechanismus steht zwar auch in UIMA zur Verfügung (wie anhand des in Abschnitt 3.2.2 beschriebenen *UIMA Type System* gezeigt), dort wird

Vererbung jedoch auf Objekt-Attribute beschränkt. IM TRS stehen hingegen nicht nur die Möglichkeiten, die Java hinsichtlich der (Mehrfach-) Vererbung für Interfaces bietet, zur Verfügung, vielmehr kann Vererbung auch zur Spezialisierung der durch AccessAdapter definierten Zugriffsmethoden eingesetzt werden.

Jedes in Tesla umgesetzte Rollensystem ist hierarchisch organisiert, da die Interfaces `DataObject` und `IAccessAdapter` zwingend implementiert werden müssen, welche in der Rolle *Annotator* zusammengefasst sind – diese *Systemrolle* definiert die technischen Anforderungen, die für Identifikation und Weitergabe der von einer Komponente produzierten Daten benötigt werden.[96]

Der Aufbau einer Rollenhierarchie ergibt sich daher automatisch: Eine Rolle R_{sub} ist eine Subrolle (oder Spezialisierung) einer Rolle R_{super}, falls die für R_{sub} definierten Interfaces von den in R_{super} angegebenen Interfaces abgeleitet wurden. Da diese Informationen zur Laufzeit ermittelt werden können, ohne auf eine (vollständige oder partielle) Hierarchisierung der Rollen angewiesen zu sein, führt das TRS dazu, dass eine implizite Vererbungshierarchie aufgebaut wird, die sich durch Definition neuer Rollen modifizieren und erweitern lässt. Damit kann eine grundlegende Kompatibilität von Komponenten realisiert werden, die andernfalls nicht möglich wäre. So wurde bspw. bei der Integration von *ABL4J*, *Berkeley Parser* und der *Suffix Align*-Komponente (vgl. Kapitel 5.3.1 für eine ausführliche Beschreibung) die Beobachtung gemacht, dass jede der Komponenten Annotationen generiert, die zwar in ihrer Bedeutung voneinander abweichen (im Gegensatz zum *Berkeley Parser* sind die von *Suffix Align* und *ABL Align* oder *ABL Select* generierten Strukturen als *Hypothesen* zu interpretieren), die sich jedoch hinsichtlich der benötigten Datenstrukturen zunächst kaum unterscheiden: Alle Komponenten generieren sequentielle, u.U. verschachtelte oder überlappende[97], konstituenten-ähnliche Annotationen, die als zusätzliche Information eine Kategoriebezeichnung tragen. Letztere Eigenschaft teilen sie u.a. mit POS-Tags, welche jedoch grundsätzlich keine Überlappungen enthalten können, da sie an Wortgrenzen gebunden sind.

Die Gemeinsamkeiten der genannten Komponenten werden in der Basisrolle *Categorizer* zusammengefasst, von der u.a. die Rollen *POS Tagger* und *Constituent Detector* abge-

[96] Die Anforderungen sind dabei minimal: Für eine `DataObject`-Implementation müssen lediglich zwei Methoden für Zugriff und Modifikation der Id (sowie das Marker-Interface `Serializable`) implementiert werden; das Interface `IAccessAdapter` erfordert (neben einigen ähnlich einfach umzusetzenden Verwaltungsmethoden) zwei Methoden, durch die auf alle Annotationen bzw. eine einzelne Annotation zugegriffen werden kann.

[97] Wie bereits in 2.2 erwähnt, sind die von einem Alignment-Verfahren generierten Hypothesen zunächst nicht zwingend überlappungs- und damit widerspruchsfrei – dies muss in einem weiteren Verarbeitungsschritt umgesetzt werden, der in Abschnitt 5.4.2 beschrieben wird.

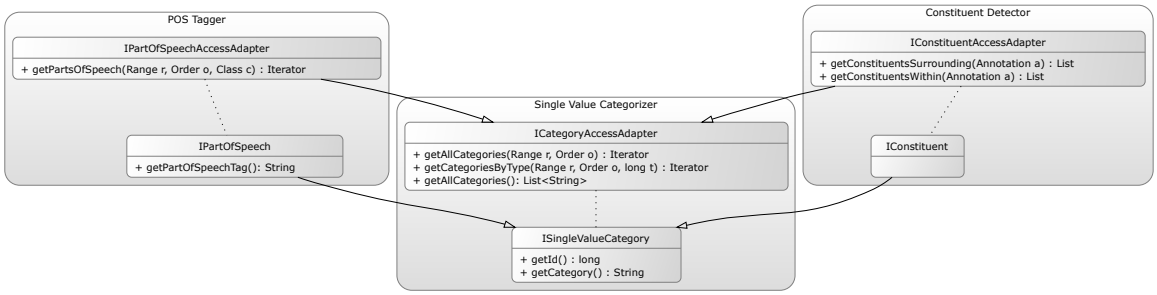

Abbildung 4.6: Ausschnitt der Rollenhierarchie von Tesla. Abgebildet ist die Assoziation zwischen *POS Tagger*, *Constituent Detector* und der Basisrolle *Categorizer*, die sich aus der Vererbungsrelation der AccessAdapter- und DataObject-Interfaces (durch Pfeile dargestellt) ergibt.

leitet sind – Abbildung 4.6 zeigt den entsprechenden Ausschnitt der Rollenhierarchie. Beide Subrollen werden weiter spezialisiert, um eine erweiterte Form von POS Tags umzusetzen (s.u.) bzw. um kontextbezogene Zusatzinformationen in Strukturhypothesen zu berücksichtigen (siehe Abschnitt 5.3.1). Das TRS ermöglicht es somit, sowohl die Wiederverwertbarkeit von Komponenten zu gewährleisten als auch die Kompatibliltät zwischen diesen zu erreichen, ohne dass Typsicherheit separat überprüft werden muss: Diese ergibt sich zwangsläufig bereits beim Kompilieren der verwendeten (Java-) Klassen.

Die Möglichkeit, (Mehrfach-) Vererbung bei der Definition von Datenstrukturen einzusetzen, ist nicht nur für die Spezialisierung von Rollen relevant, sondern kann auch innerhalb einer einzelnen Rolle sinnvoll sein. So verwendet beispielsweise die Rolle *POS Tagger* die Datenstruktur `IPartOfSpeech`, welche die Methode `String getPartOfSpeechTag()` anbietet, um – analog zu den in Listing 3.5 und 3.6 aufgeführten (J)CAS-Datenstrukturen in UIMA – den Zugriff auf eine (proprietäre) POS-Auszeichnung zu ermöglichen. In der von *POS Tagger* abgeleiteten Rolle *Tesla POS Tagger* wird diese Datenstruktur durch zahlreiche Subinterfaces spezialisiert: Vererbung wird hier dafür eingesetzt, unterschiedliche Kategorien von POS-Tags zu definieren und gleichzeitig unterschiedliche Eigenschaften dieser Tags anzugeben, sofern dies sprachunabhängig möglich ist[98]. Die für deutsche und englische POS-Tags entwickelte Rolle *Tesla POS Tagger* setzt dies dadurch um, dass bspw. mögliche Werte von Kasus, Numerus und Genus festgelegt werden (vgl. Abbildung 4.7). Im Vergleich zu datenorientierten Ansätzen erforderte dies zwar deutlich höheren Entwicklungsaufwand, bietet jedoch den Vorteil, dass die Eigenschaften eines POS-Tags für Entwickler unmittelbar zugänglich sind (indem die spezialisierten Interfaces entspre-

[98]Die Entwicklung der Tag-Hierarchie wurde im Rahmen einer Hausarbeit (online unter `http://www.spinfo.phil-fak.uni-koeln.de/mneumann.html`) durchgeführt, für die Implementierung dieser Hierarchie sei an dieser Stelle Sebastian Rose gedankt.

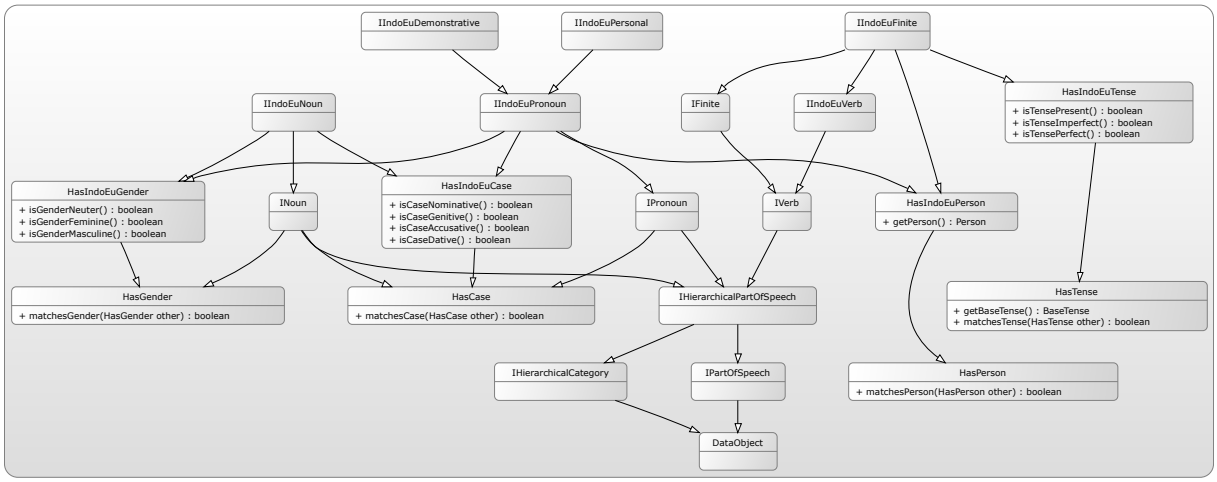

Abbildung 4.7: Ausschnitt der Interfacehierarchie der Part of Speech Tags in Tesla. Die objektorientierte Umsetzung bietet gegenüber String-basierten Tag-Sets die Vorteile fester Typisierung und vereinfachter Überprüfung verschiedener Merkmale (wie Kasus oder Genus). Zudem können verschiedene Tag-Sets in der Hierarchie abgebildet werden, die so als Meta-Format genutzt werden kann, um bspw. vom *Stuttgart-Tübinger Tagset* (STTS) oder dem *BNC Basic (C5) Tagset* zu abstrahieren.

chende Methoden anbieten, wie `boolean isCaseNominative()`, vgl. Abbildung 4.7), und dass sichergestellt wird, dass unterschiedliche POS-Tagger innerhalb der gleichen Sprache stets auch gleiche Kategorien zuweisen, so dass Komponenten, die derartige Tags konsumieren, nicht für spezifische Tagsets konfiguriert werden müssen. Statt die individuelle Terminologie eines Tagsets interpretieren zu müssen und ggfs. Komponenten zu entwickeln, die lediglich mit genau einem Tagset kompatibel sind, steht Entwicklern so eine objektorientierte, einfach zu verwendende Form des Zugriffs auf morphosyntaktische Informationen zur Verfügung. Die Konvertierung von POS-Tags in ein objektorientiertes Format muss zudem nur einmal implementiert werden, während eine Zeichenkettenbasierte Kodierung der morphosyntaktischen Informationen stets eine Einarbeitung in das jeweilige Tagset erfordert – der einmalige Mehraufwand bei Implementation einer Konvertierungskomponente kann somit zu einer Beschleunigung folgender Entwicklungsprozesse führen.

Während bei der Produktion von Annotationen die Möglichkeit der Spezialisierung durch Vererbung im Vordergrund steht, ist dies beim Konsumieren von Annotationen umgekehrt: Hier kann das Rollensystem verwendet werden, um die Anforderungen an zu verarbeitende Annotationen möglichst gering zu halten und die Einsatzmöglichkeiten einer Komponente somit zu erweitern. Die ABL4J-Komponenten benötigen bspw. die

Information über die Start- und Endposition von Sequenzen, die aligniert werden sollen, sowie die (geordnete und überlappungsfreie) Liste der Elemente, aus denen die Sequenz besteht – dabei ist es jedoch irrelevant, um welche Art von DataObject es sich bei einem Element handelt, denn für das verwendete Alignment-Verfahren muss ausschließlich die Gleichheit zweier Elemente überprüft werden können. Diese Anforderungen sind in Form von zwei konsumierten Rollen festgelegt, welche in der Rollenhierarchie knapp unter der Wurzel definiert sind (siehe Abschnitt 4.1.4.2) und somit von nahezu allen Komponenten, die Tokensequenzen annotieren, erfüllt werden. Unter technischen Gesichtspunkten ist ein Alignment von Wörtern oder Buchstaben dadurch ebenso möglich wie ein Alignment von POS-Tags, und auch eine Kombination aus diesen Annotationen[99] kann verarbeitet werden.

4.1.4.2 Diskussion und Beispiel

Wie in den vorherigen Abschnitten gezeigt, kann der durch das Tesla Role System entwickelte Ansatz einige Unzulänglichkeiten von datenorientierten Komponentensystemen beseitigen. Dies ist möglich, weil Daten als flexible, dynamische Objekte im Sinne der Objektorientierung betrachtet werden, die nicht aus (evtl. rekursiven Aggregationen von) Attributen bestehen, sondern durch ihre Funktionalität definiert werden, und weil Entwicklern die Möglichkeit gegeben wird, individuelle Zugriffsmethoden auf diese Objekte zu definieren. Die so erreichte Flexibilität kann von einem datenorientierten System nicht erreicht, jedoch gleichzeitig auch als Nachteil des Rollensystems betrachtet werden: Bei der Definition neuer Rollen ist es, wie die Erfahrung bei der Entwicklung von Tesla gezeigt hat (vgl. auch Hermes 2011, Kapitel 3.2.2), oftmals naheliegend, die bestehende Rollenhierarchie zu modifizieren – etwa, um die Gemeinsamkeiten mit existierenden Rollen zu modellieren und so die Wiederverwendbarkeit der Komponenten zu verbessern, oder um eine Verletzung des Substitutionsprinzips[100] zu vermeiden. Zudem ist es nahezu ausgeschlossen, dass zwei Entwickler, die unabhängig voneinander eine konzeptuell identische Rolle definieren, identische Bezeichnungen für die Interfaces und die dort aufgeführten Methoden verwenden, so dass Parallelentwicklung zu Inkompatibilität führt. Dies ist zwar auch bei datenorientierten Ansätzen sehr wahrscheinlich[101], jedoch ist eine Konvertierung zwischen unterschiedlichen Repräsentationen zweier (fast) identischer

[99] Ein Beispiel für einen derartigen Versuchsaufbau wird in Kapitel 5 gezeigt.

[100] Eigenschaften einer Superklasse müssen auch für sämtliche Subklassen gelten, um eine *ist-ein*-Relation zwischen Sub- und Superklasse aufrecht zu erhalten (vgl. Liskov & Wing 2001).

[101] vgl. die auf Seite 65 diskutierten Vor- und Nachteile einer domänenspezifischen Typisierung im Kontrast zu einer generischen Auszeichnung.

Datenstrukturen mit deutlich weniger Aufwand zu realisieren als eine Konvertierung inkompatibler Interface-Hierarchien und kann in einfachen Fällen sogar generisch, mit vom Komponentensystem bereitgestellten Hilfsmitteln, durchgeführt werden.

Um eine Fragmentierung des Rollensystems zu vermeiden, wurde daher eine Menge von Basisrollen definiert, die sich bezüglich der Art der Annotation unterscheiden und die bei der Entwicklung neuer Rollen berücksichtigt werden sollten. Diese Rollen dienen dazu, Spezialisierungen nach allgemeinen Eigenschaften bezüglich der Form der Annotation zu gruppieren, und wurden daher sehr abstrakt gehalten, wie folgende Auflistung zeigt:

- Die Basisrolle *Sequence Annotator* kann von Rollen, die eine Symbolsequenz in nicht überlappende Teilsequenzen aufteilt, spezifiziert werden. So sind bspw. Tokenizer und POS-Tagger von dieser Rolle abgeleitet, während die Umsetzung der in Kapitel 2 beschriebenen Alignment-Verfahren ebendiese Rolle konsumiert – entsprechend können sowohl Tokens als auch POS-Tags von dieser Komponente verarbeitet werden.

- Im Gegensatz dazu sind überlappende Annotationen bei einem *Categorizer* erlaubt: Die Auszeichnung von Konstituentenstrukturen durch einen Parser, oder die Auszeichnung von Wörtern und Wortketten auf Basis von Schlüsselwortlisten, wie sie von der *Gazetteer*-Komponente durchgeführt wird, sind als Beispiele derartiger Rollenimplementationen zu nennen. Categorizer müssen zudem eine Methode zur Definition der Type-Id (vgl. Abschnitt 4.2.2) einer Annotation implementieren, da der annotierte Signalausschnitt i.d.R. nicht für eine Berechnung dieser Id ausreicht.

- *Linker* zeichnen sich schließlich dadurch aus, dass derartige Rollen Annotationen zueinander in Relation setzen, wie etwa bei der Detektion von Dependenz-Relationen.

Die obige Auflistung ist sicherlich als unvollständig zu bezeichnen, da sie anhand der Rollen, die im Rahmen der Forschungsprojekte, in denen Tesla zur Zeit verwendet wird, erstellt wurde, sie kann jedoch beliebig erweitert oder modifiziert werden, so dass sie sich an individuelle Anforderungen anpassen lässt – die Flexibilität des TRS ist diesbezüglich vergleichbar mit der in UIMA genutzten *Common Analysis Structure*, durch die ebenfalls die Verwendung unterschiedlicher Typsysteme ermöglicht wird (vgl. Abschnitt 3.2.2).

Als zweiter Kritikpunkt kann der Implementationsaufwand, der für die Entwicklung von DataObject-Hierarchien benötigt wird, genannt werden – dies wird bspw. bei der im vorangegangenen Abschnitt bereits skizzierten Rolle *Tesla POS Tagger* deutlich, für die fast 100 Interfaces und ebensoviele Klassen implementiert wurden, während in einem

datenorientierten Framework im einfachsten Fall lediglich ein Feld für die Angabe der jeweiligen Kategorie benötigt wird. Allerdings handelt es sich hier zum einen um einen Extremfall, der bei der Konzeption anderer Rollen nicht auftrat, zudem reduzieren die in Abbildung 4.7 gezeigten Methoden den für die Verwendung eines POS-Taggers benötigten Einarbeitungsaufwand und erhöhen die Lesbarkeit des Programmcodes. In anderen Fällen, wie etwa bei der Integration des *Stanford Parsers* in Tesla bzw. GATE, kann der Implementationsaufwand hingegen durch das TRS reduziert werden, da eine vollständige Konvertierung der Datenstrukturen hier nicht notwendig ist.

Vorteile gegenüber datenorientierten Ansätzen bietet das TRS zudem dann, wenn durch die Anreicherung einer Datenstruktur mit Algorithmen eine feiner granulierte Spezifikation von Rollen (und damit auch von Komponenten) möglich ist – dies sei am Beispiel der Tesla-Komponenten *Word Vector Generator*, *Geo Location Extender* und *K-Means Clusterer*[102] gezeigt. Der K-Means-Algorithmus operiert auf vektoriellen Repräsentationen von Elementen, aus denen sowohl Cluster-Zentren als auch die Zugehörigkeit der Vektoren (und damit auch der Elemente) zu einem solchen Zentrum errechnet werden. Dafür muss lediglich die Distanz zwischen zwei Vektoren berechnet werden, wobei die Art der Berechnung von den Eigenschaften des Vektorraums abhängt. Sowohl *Word Vector Generator* als auch *Geo Location Extender* generieren vektorielle Elementrepräsentationen: Die letztgenannte Komponente erweitert mit Hilfe einer Datenbank die Vorkommen geographischer Bezeichnungen in Texten um eine Angabe zu Längen- und Breitengrad (die als zweidimensionaler Vektor interpretiert werden kann), während die erstgenannte Komponente zu jedem Wort eine vektorielle Repräsentation des Verwendungskontextes erstellt[103] – da die Dimension der generierten Vektoren der Anzahl unterschiedlicher Wortformen in den untersuchten Texten entspricht, liegt sie bspw. bei Verarbeitung des vergleichsweise kleinen *TüBa-D/S* bereits bei über 6.500 (vgl. auch Tabelle 5.1 auf Seite 163).

Bezüglich der Verarbeitung beider Arten von Vektoren sind zwei Beobachtungen festzuhalten: Zum einen müssen unterschiedliche Distanzmaße für die Berechnung der Entfernung zweier Vektoren verwendet werden, da es sich im Fall des Paares aus Längen- und Breitengrad um einen Vektor in einem sphärischen (bzw. elliptischen) Raum handelt[104], während die Wortvektoren in einem euklidischen Raum liegen. Zum anderen sind

[102] Eine ausführlichere Beschreibung der genannten Komponenten findet sich in Anhang B.

[103] Zur Generierung von Wortvektoren werden üblicherweise die Kontexte aller Vorkommen der jeweiligen Wortform untersucht und in einen Vektor überführt, der die Kookurrenzen mit anderen Wortformen abbildet. In Abschnitt 5.5 wird dieses Verfahren genauer erläutert.

[104] Die Berechnung der Distanz zwischen zwei Orten muss auf einer Kugeloberfläche erfolgen, wenn exakte Entfernungsangaben benötigt werden.

die meisten Positionen eines Wortvektors unbelegt, was bei der Berechnung der Distanz zugunsten der benötigten Laufzeit berücksichtigt werden kann: Wird bspw. die für zwei Vektoren a und b durch $\sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$ definierte euklidische Distanz verwendet (vgl. auch Witten *et al.* 2005, S. 128f), so genügt es, nur die Positionen der Vektoren zu betrachten, die in a oder b einen von *0* abweichenden Wert enthalten.

Da durch das TRS die Definition eines DataObjects um beliebige Methoden erweitert werden kann, ist es möglich, Vektoren so zu definieren, dass obige Beobachtungen berücksichtigt werden. So kann bspw. ein Interface `IVector` die Methode `double getDistance (IVector other)` deklarieren und die Berechnung der Distanz in einer Implementation des Interfaces entsprechend des zugrundeliegenden Vektorraumes umgesetzt werden, und eine Clustering-Komponente kann, da die notwendigen Algorithmen in den zu verarbeitenden Java-Klassen enthalten sind, in generischer, vom Vektorraum unabhängiger Form implementiert werden. Eine Optimierung von Laufzeit- und Speicherbedarf kann dadurch umgesetzt werden, dass die innerhalb einer `IVector`-Implementation genutzte Datenstruktur an die Eigenschaften des Vektors angepasst wird. So kann bspw. für dicht belegte Vektoren ein Array verwendet werden, dessen Länge der Dimension des Vektors entspricht, während dünn belegte Vektoren eine Map nutzen, in der in Form von Index-Wert-Paaren nur Elemente mit einem von *0* abweichenden Wert abgelegt werden (vgl. auch Witten *et al.* 2005, S. 55f).

Der hier beschriebene Anwendungsfall lässt sich nur bedingt in einem Framework wie UIMA umsetzen: Da sich in der CAS keine Algorithmen definieren lassen, müssten Methoden zur Distanzberechnung in der Clustering-Komponente definiert werden, was bei einer Erweiterung um neue Algorithmen die Modifikation der Komponente und damit die Verfügbarkeit des Quellcodes erfordert, oder aber voraussetzt, dass eine Erweiterungsschnittstelle in die Komponente integriert wurde. Zudem müsste die interne Struktur eines Vektors in der CAS festgelegt werden – eine flexible, für den jeweiligen Anwendungsfall optimierte Repräsentation der Datenstruktur kann nicht verwendet werden.

Schließlich kann die durch das TRS ermöglichte, objektorientierte Kapselung von Daten und Methoden u.U. auch zu einer verbesserten Fokussierung auf das Fachwissen der beteiligten Entwickler führen: So ist (insbesondere im Kontext eines Frameworks, dessen Schwerpunkt auf korpuslinguistischen Verfahren liegt) nicht unbedingt zu erwarten, dass der Entwickler einer Clustering-Komponente mit Berechnungen auf Basis geographischer Koordinaten vertraut ist – dieses Wissen ist eher beim Entwickler der Komponente, die diese Daten generiert, zu vermuten. Das TRS ermöglicht somit eine API-orientierte Form der Softwareentwicklung: Es ist nicht notwendig, sich mit dem internen Design ei-

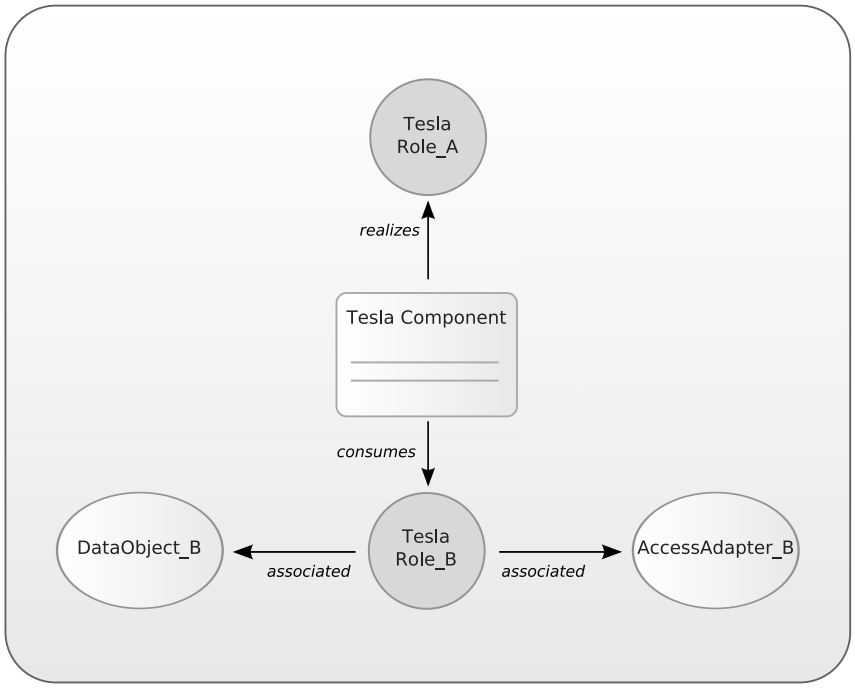

Abbildung 4.8: Assoziationen zwischen Komponenten und Rollen (Grafik aus Hermes & Schwiebert 2010). Eine Komponente kann beliebig viele Rollen konsumieren und produzieren bzw. realisieren. Da Rollen über Java-Interfaces umgesetzt sind (vgl. Abbildung 4.4), können auch komplexe Datenstrukturen zum Austausch verwendet werden, zudem lassen sich DataObject und AccessAdapter frei, d.h. ohne Einschränkungen durch das Framework, implementieren.

ner Komponente auseinanderzusetzen, sondern es genügt, die öffentlichen, idealiter gut dokumentierten und leicht zu nutzenden Schnittstellen zu verwenden.

4.1.5 Komponenten

In diesem Abschnitt werden Konzept und Aufbau einer Komponente in Tesla zusammengefasst. Insbesondere wird dabei erläutert, wie die Verbindung zwischen Rollen, Signalen und Komponenten definiert wird (Abschnitt 4.1.5.1), wie Komponenten konfiguriert werden können (Abschnitt 4.1.5.2) und wie Komponenten in Tesla ausgeführt werden (Abschnitt 4.1.5.4). Zunächst soll jedoch auf die unterschiedliche Verwendung des Begriff *Komponente* in Tesla im Vergleich zur Definition von Szyperski *et al.* (1998) – siehe Einleitung zu Kapitel 3 – eingegangen werden.

Abbildung 4.8 zeigt den schematischen Aufbau einer Tesla-Komponente, die eine Rolle konsumiert und eine weitere Rolle produziert. Die produzierte Rolle klassifiziert eine Komponente und legt fest, auf welche Art und Weise sie in einem Experiment verwendet

werden kann (bspw. als Tokenizer, POS-Tagger oder Clusterer). Eine Tesla-Rolle erfüllt so die Komponentendefinition von Szyperski, nach der eine Komponente als Schema bzw. *immutable plan* (vgl. Seite 45 in Abschnitt 3 dieser Arbeit) betrachtet werden kann und unabhängig von konkreten Implementationen existiert.

Im Kontext von Tesla wird der Komponentenbegriff hingegen auf die konsumierten Rollen ebenso wie auf die verwendeten Algorithmen ausgeweitet (siehe Abbildung 4.8).Tesla-Komponenten sind daher meist nicht *per se* austauschbar, auch wenn sie identische Rollen produzieren, da sie sich in ihren Anforderungen an konsumierte Rollen unterscheiden können und somit unterschiedlich aufgebaute Experimente erfordern. Dies ist jedoch kein Widerspruch, sondern lediglich ein terminologischer Unterschied: Eine Tesla-Komponente kann als Instanz eines Komponentenschemas nach Szyperski betrachtet werden, deren produzierte Daten kompatibel zu sämtlichen anderen Komponenten, die dieses Schema erfüllen, sind, während sie sich aufgrund unterschiedlicher Problemlösungsansätze und damit verbundenen unterschiedlichen Anforderungen (notwendigerweise) bezüglich der konsumierten Daten unterscheiden.

4.1.5.1 Metadaten

Technisch gesehen handelt es sich bei einer Tesla-Komponente um eine Javaklasse, die von der Basisklasse `de.uni_koeln.spinfo.tesla.runtime.TeslaComponent` abgeleitet wurde und zudem mit einigen Java-Annotationen[105] ausgezeichnet wurde. Letztere werden benötigt, um einer Komponente Metadaten hinzuzufügen, so dass eine natürlichsprachliche Beschreibung der Komponente angezeigt werden kann, gleichzeitig aber auch Felder, Methoden und Eigenschaften der Klasse zur Laufzeit der Komponente vom System interpretiert und modifiziert werden können. Bevor Annotationen mit Java 5 eingeführt wurden, mussten solche Metadaten in externen Dateien abgelegt werden (wie bei der Verwendung von *Enterprise Java Beans* bis Version 1.4 in sog. *Deployment Deskriptoren*, vgl. Shannon 2003, Abschnitt 8.1.2). Zwar ergeben sich durch dieses Vorgehen keine funktionalen Nachteile gegenüber dem Einsatz von Annotationen, letztere bieten jedoch u.a. den Vorteil, dass Metadaten im Quellcode und somit unmittelbar an den referenzierten Elementen vorhanden sein können, und dass Java-Entwicklungsumgebungen Annotationen

[105] Um terminologische Unklarheit zu vermeiden, soll an dieser Stelle darauf hingewiesen werden, dass der Begriff *Annotation* in diesem Abschnitt stets auf die Auszeichnung einer Java-Klasse durch Metadaten referiert und nicht mit der Auszeichung eines Textes zu verwechseln ist. Dies gilt auch für folgende Abschnitte, in denen der Begriff im Zusammenhang mit semantischen Auszeichungen einer Java-Klasse verwendet wird.

i.d.R. so unterstützen, dass syntaktische und (einige[106]) semantische Fehler verhindert und Refactoring-Operationen (wie das Umbenennen oder Verschieben von Klassen) unterstützt werden.

Im Wesentlichen handelt es sich bei den für Tesla entworfenen Annotationen um Metadaten über

- konsumierte Rollen und Signale
- produzierte Rollen
- Konfigurationsoptionen und
- die auszuführende Methode.

Diese Informationen werden mit Hilfe der *Java Reflection API* beim Start des Systems ausgelesen und (im Falle von Feldern) bei der Ausführung einer Komponente mit Hilfe von *Dependency Injection* (vgl. Abschnitt 4.3.1 für Definition und Erläuterung des Begriffs) zugewiesen.

Konsumiert eine Komponente ein Signal, so wird dies dadurch festgelegt, dass ein Feld vom Typ `SignalAccessor` definiert und mit der Annotation `@SignalAdapter` ausgezeichnet wird; zudem muss noch die Art des Signals (in korpuslinguistischem Kontext `java.lang.String`, vgl. Abschnitt 4.1.3.1) sowie eine innerhalb der Komponente eindeutige Bezeichnung, durch die mehrere konsumierte Signale unterschieden werden können, angegeben werden (vgl. Zeile 30 – 31 in Listing 4.2). Die Eingabe-Schnittstelle für konsumierte Rollen ist technisch analog realisiert: Hier muss das jeweilige Feld aus einem mit einem DataObject typisierten IAccessAdapter bestehen und mit der Annotation `@AccessAdapter` versehen werden. Zusätzlich zu einer eindeutigen lokalen Bezeichnung wird hier allerdings noch die eindeutige Id der konsumierten Rolle verlangt (vgl. Zeile 15 – 21 in Listing 4.2). Eine Komponente kann beliebig viele Rollen und Signale konsumieren, wobei mindestens eine Rolle oder ein Signal verarbeitet werden muss. Dabei ist es zulässig, eine flexible Anzahl identischer Rollen zu verarbeiten, wodurch es bspw. möglich ist, zu prozessierende Daten auf unterschiedliche Art zu filtern (vgl. Anhang B.8.3) oder die von verschiedenen Komponenten generierten Daten neu zu kombinieren (vgl. Anhang B.7.4.). In derartigen Fällen muss eine entsprechend typisierte Liste verwendet und die

[106] So kann bspw. gewährleistet werden, dass eine Klasse, die in einer Java Annotation referenziert wird, ein bestimmtes Interface implementiert. Die Validierung von primitiven Datentypen oder Strings, bspw. zur Überprüfung der Korrektheit einer Rollen-Id, ist hingegen nicht möglich (vgl. dazu auch Abschnitt 4.1.7.2).

`@AccessAdapter`-Annotation um die Angaben der minimalen und maximalen Anzahl von Rollen erweitert werden.

Wie Zeilen 23 – 28 in Listing 4.2 zeigen, erfordert die Output-Schnittstelle einer Komponente (die in Abschnitt 4.2 ausführlicher erklärt wird) einige zusätzliche Angaben, da neben der Rolle, die von der jeweiligen Schnittstelle erfüllt wird, auch die Implementationen der von der Rolle festgelegten Interfaces angegeben werden müssen. Dies ist notwendig, damit das System die entsprechenden Klassen bei Bedarf instantiieren und entsprechenden IAccessAdapter-Feldern anderer Komponenten zuweisen kann.

Um die Konfiguration einer Komponente zu ermöglichen, lassen sich primitive Datentypen, Strings und Listen dieser Typen mit der Annotation `@Configuration` auszeichnen (vgl. Zeile 33 – 41 in Listing 4.2), anhand derer Teslas graphische Benutzeroberfläche bei der Konfiguration eines Experiments entsprechende Editoren zur Modifikation der Parameter bereitstellt (siehe Abbildung 4.9). Da es sich hierbei um die komplexeste Annotation innerhalb einer Komponentendefinition handelt, wird sie im folgenden Abschnitt separat beschrieben.

Schließlich verwendet das Beispiel in Listing 4.2 noch drei weitere Annotationen: Mit `@Configure` kann ein zusätzlicher Konfigurationsschritt in Form einer Methode, die vor der Ausführung des Hauptalgorithmus (welcher durch die Annotation `@Run` markiert wird) aufgerufen wird, festgelegt werden. Die in der Annotation `@Component` angegebenen Metadaten werden in erster Linie für die Darstellung der Komponente in der graphischen Benutzeroberfläche benötigt – lediglich das Element `ThreadMode` ist für die Ausführung einer Komponente von Bedeutung, da sich hiermit u.U. eine Optimierung auf Mehrkern-Systemen erreichen lässt (vgl. dazu Abschnitt 4.1.5.4).

4.1.5.2 Konfiguration von Komponenten

Neben der Auswahl von Komponenten, die in einem Experiment verwendet werden sollen, und der Verknüpfung von produzierten und konsumierten Rollen handelt es sich bei der Konfiguration von Komponenten um eine der zentralen Aufgaben, die von Anwendern in Tesla durchzuführen sind. Gleichzeitig kann die Konfiguration im Vergleich zu den anderen genannten Aufgaben sehr komplex (und damit auch fehleranfällig) sein: Während sich die Auswahl von Komponenten aus der zu überprüfenden Hypothese ergibt (und sich dadurch die korrekte Verbindung von produzierten und konsumierten Rollen herleiten lässt), können sich die Konfigurationsoptionen unterschiedlicher Realisierungen einer identischen Rolle stark unterscheiden und Spezialwissen über die zugrundeliegenden Algorithmen erfordern. Zudem können Optionen auf einen bestimmten Wertebereich beschränkt sein;

```
@Component(threadMode=ThreadMode.CUSTOM,
   author=@Author(author="Stephan Schwiebert",
       email="sschwieb@spinfo.uni-koeln.de",
       web="http://www.spinfo.phil-fak.uni-koeln.de/sschwieb.html",
       organization="Sprachliche Informationsverarbeitung"),
   description=@Description(name="ABL Align",
       licence=Licence.LGPL_2,
       summary="Generates constituent hypotheses",
       bigO="depends on the chosen align method",
       version="1.0",
       web="http://developer.berlios.de/projects/abl4j/",
       reusableResults=true))
public class Abl4JAlignComponent extends AblTeslaComponent {

    @AccessAdapter(role="de.uni_koeln.spinfo.tesla.roles.core.AnchoredElementGenerator",
            name="Sequences", description="The sequences to align (for instance, sentences)")
    private IAnchoredElementAccessAdapter<IAnchoredElement> sequences;

    @AccessAdapter(role="de.uni_koeln.spinfo.tesla.roles.core.AnchoredElementGenerator",
            name="Sequence Items", description="The items to align (for instance, words)")
    private IAnchoredElementAccessAdapter<IAnchoredElement> words;

    @OutputAdapter(dataObject=MultiLabelConstituent.class,
            type=DefaultTunguskaOutputAdapter.class, name="Constituents",
            accessAdapterImpl=DefaultTunguskaConstituentAccessAdapter.class,
            description="All detected constituent hypotheses")
    @RoleDescription("de.uni_koeln.spinfo.tesla.roles.parser.MultiValueConstituentTagger")
    private IOutputAdapter<IMultiLabelConstituent> constituentOut;

    @SignalAdapter(signalType=String.class, name="Texts")
    private SignalAccessor<String> signalAdapter;

    @Configuration(defaultValue="true", name="Don't merge",
            description="Do not try to merge hypotheses. See ABL4J documentation for details.")
    private boolean noMerge = true;

    @Configuration(editor="...itemeditors.ChoiceEditor",
            defaultValue="...RightAlignmentMethod", name="Alignment Method",
            description="The alignment method that will be used.",
            min=1, max=1, delegate=AlignmentMethod.class)
    private String alignmentMethod = "...RightAlignmentMethod";

    @Configure
    public boolean configure() {
      ...
      return true;
    }

    @Run
    public Result run() throws Exception {
      ...
      return Result.OK;
    }
}
```

Listing 4.2: Java-Annotationen einer Tesla-Komponente. Dargestellt ist ein Ausschnitt der Klasse `Abl4JAlignComponent`, mit der die in Abschnitt 2.2 beschriebene *Align*-Phase von ABL in Tesla umgesetzt wird (vgl. auch Abschnitt 5.3).

Abbildung 4.9: Konfiguration von Komponenten eines Experiments am Beispiel des *Berkeley Parsers*. Auf der rechten Seite ist der Editor zur Konfiguration des Parameters *Grammar File* abgebildet, dessen Auswahlmöglichkeiten vom Tesla-Server zur Laufzeit (anhand der in Listing 4.3 gezeigten Annotation) generiert werden.

ebenso ist es möglich, dass einzelne Optionskombinationen inkompatibel zueinander sind – daher wurde bei der Konzeption von Tesla versucht, eine Möglichkeit zu finden, durch die Anwender bei der Konfiguration einer Komponente von deren Entwicklern unterstützt werden können.

Der gewählte Weg besteht darin, dass die im vorherigen Abschnitt bereits erwähnte `@Configuration`-Annotation um Elemente erweitert werden kann, die sowohl die Bedeutung des jeweiligen Parameters beschreiben als auch verschiedene Möglichkeiten bieten, eine dem Experiment und der Komponente angemessene Wahl von Einstellungen zu treffen. So können Parameter beispielsweise um eine Beschreibung ergänzt werden, die bei der Konfiguration angezeigt wird, und die zudem Links zu externen Quellen (wie der Beschreibung eines Algorithmus in der Wikipedia oder der Möglichkeit zum Download zusätzlicher Lexika, die von einer Komponente verarbeitet werden können) enthalten können. Da in der graphischen Benutzeroberfläche von Tesla u.a. ein Browser eingebettet ist[107], kann unmittelbar auf derartige Informationen zugegriffen werden.

Entwickler können zudem definieren, welcher Editor für einen Konfigurationsparameter verwendet werden soll. Standardmäßig nutzt Tesla für primitive Datentypen und String-Objekte Editoren, die der Art des jeweiligen Datentyps entsprechen (etwa eine Auswahlbox für boolesche Werte), bietet jedoch auch zusätzliche Editoren, die weitere Einschränkungen (wie etwa eine Beschränkung auf die Auswahl positiver Zahlen oder die Definition eines Wertebereichs zwischen 0 und 1) ermöglichen – eigene Editoren können dem System ebenfalls hinzugefügt werden.

Ferner ist es möglich, gewählte Parameter client- oder serverseitig validieren zu lassen, oder initiale Parameterbelegungen fest vorzugeben oder algorithmisch zu erzeugen. So werden bspw. serverseitig vorhandene Sprachkonfigurationsdateien, die vom *Berkeley Parser* verwendet werden, ermittelt und dem Anwender zur Auswahl bereitgestellt. Die ABL-Komponenten nutzen diese Schnittstelle, um eine Enumeration zulässiger Parameterbelegungen vorzugeben, oder um zur Laufzeit verfügbare Alignment-Methoden zu ermitteln und an die graphische Oberfläche weiterzuleiten. Die `@Configuration`-Annotation wird

[107] Die graphische Benutzeroberfläche von Tesla basiert auf dem Eclipse Framework (vgl. Abschnitt 4.1.7 ab Seite 127), welche dieses Feature zur Verfügung stellt.

dazu um die Angabe einer Restriktion, deren Konfiguration und die Information darüber, ob die Auswertung client- oder serverseitig durchgeführt werden soll, erweitert (vgl. Listing 4.3) und von der graphischen Oberfläche während der Konfiguration einer Komponente ausgewertet (vgl. die mit Listing 4.3 korrespondierende Abbildung 4.9).

```
1  @Configuration(name="Grammar File",
2      defaultValue="ger_sm5.gr",
3      editor="...ServerChoiceEditor",
4      restriction=ServerResourcesChoice.class,
5      editorConfiguration={"single", ".gr"},
6      evaluateOnServer=true,
7      description="The language file...")
8  private String inFileName = "ger_sm5.gr";
```

Listing 4.3: Verwendung von Java-Annotationen in Tesla am Beispiel `@Configure`. Das aus der Klasse `BerkeleyParserWrapper` entnommene Fragment ermöglicht die dynamische Restriktion von Parameterbelegungen bei der Auswahl von Sprachmodellen.

Neue Restriktionen können durch Implementation des Interfaces `de.uni_koeln.spinfo.tesla.runtime.component.annotations.Restriction` realisiert werden – die hier beschriebenen Restriktionen lassen sich jedoch für vergleichbare Anwendungsfälle (wie die Auflistung von Dateien in einem serverseitig vorhandenen Verzeichnis) wiederverwerten.

In einigen Fällen ist es sinnvoll, mehrere Parameterbelegungen zu bündeln und Anwendern zur Auswahl zu stellen – etwa dann, wenn Komponenten in verschiedenen Modi verwendet werden können, oder wenn die manuelle Konfiguration eines einzelnen Parameters hohen Aufwand erfordert, weil z.B. große Listen oder komplexe XML-Strukturen definiert werden müssen, die gleichzeitig jedoch wiederverwertbar sind (wie etwa Listen von Personen- oder Ortsnamen in unterschiedlichen Sprachen). Hierfür bietet Tesla *Templates* und *Template Sets*, mit denen Parameterbelegungen in separaten Dateien definiert und mit den betroffenen Parametern verknüpft werden können. Listing 4.4 zeigt einen Ausschnitt aus einer solchen Datei.

Da Templates extern, d.h. außerhalb des Quellcodes einer Komponente, definiert werden, können auch eigene Konfigurationen einer Komponente als Template bzw. Template Set gespeichert werden – diese werden auf dem Tesla Server abgelegt und stehen anschließend sämtlichen Anwendern zur Verfügung.

Durch die in diesem Abschnitt erläuterten Möglichkeiten wird der Aufwand, der für die Konfiguration einer Komponente notwendig ist, reduziert, und Fehlkonfigurationen können verhindert werden. Entwickler können sowohl eigene Editoren und Restriktionen

```
<templates>
  <component id="de.uni_koeln.spinfo.tesla.component.gazetteer.GazetteerComponent"/>
  <set id="1" default="true">
    <name>German Categories</name>
    <description>...</description>
    <template id="names_de"/>
    <template id="colors_de"/>
  </set>
  <template id="names_de" name="German given names" category="List">
    <description>Given names which are popular in Germany</description>
    <reference file="names_de.lst"/>
  </template>
  <template id="ingore_case" name="Ignore Case" category="Ignore Case">
    <description>...</description>
    <value>true</value>
  </template>
</templates>
```

Listing 4.4: Ausschnitt einer vom *Gazetteer* (siehe Anhang B.7.1) verwendeten *Template*-Datei. Das Beispiel zeigt die Definition von Templates für zwei Konfigurationsparameter, die zusätzlich in einem Template Set zusammengefasst werden. Während das erste Template mit Hilfe des `reference`-Elements auf eine externe Datei verweist, wird der Inhalt der Vorlage im zweiten Template (durch das Element `value`) inkludiert.

entwerfen als auch auf bestehende Klassen zurückgreifen, so dass das Konfigurationskonzept nur dann zusätzlichen Implementationsaufwand verursacht, wenn die Konfiguration eines Parameters derart komplex ist, dass die bereits existierenden Lösungen sie nicht hinreichend vereinfachen. In diesem Fall ist es Entwicklern freigestellt, auf eine Validierung des Parameters zu verzichten, die Dokumentation des Parameters zu erweitern oder aber eine entsprechende Restriktion zu implementieren.

Die Verwendung von Java Annotationen zur Konfiguration einer Komponente, aber auch zur Auszeichnung von konsumierten und produzierten Rollen und Signalen, wie in Listing 4.2 dargestellt, bietet gegenüber einer externen Auszeichnung zudem den Vorteil, dass der Quellcode der Komponente gleichzeitig dokumentiert wird, was als eine Form des *Literate Programming* nach Knuth (1984) betrachtet werden kann. Alle Annotationen, die für die Interaktion mit Anwendern relevant sind, verfügen über ein *description*-Element, dessen Inhalt ausgelesen und in Teslas graphischer Oberfläche dargestellt wird, das gleichzeitig jedoch auch den Quellcode der Komponente kommentiert. In Kombination mit den von AccessAdaptern und DataObjects definierten Methoden, welche an die Semantik der jeweiligen Rolle angepasst sind, wird die Lesbarkeit des Quellcodes und damit auch dessen Wartbarkeit gegenüber generischen Ansätzen (wie im Fall von GATE, vgl. Listing 3.3 auf Seite 59) deutlich verbessert.

Zwar kann es als Nachteil von Annotationen betrachtet werden, dass Änderungen der

Metadaten ein erneutes Kompilieren des Quellcodes erforderlich ist, doch trifft dieser Kritikpunkt in erster Linie für Ansätze zu, in denen Abhängigkeiten zwischen verschiedenen Systembestandteilen durch Metadaten modelliert werden, wie im Fall von *Dependency Injection* im *Spring Framework* (vgl. Abschnitt 4.3.1). Derartige Abhängigkeiten werden in Tesla jedoch nicht durch Metadaten, sondern in Form von Experimenten umgesetzt (vgl. Abschnitt 4.1.2), weshalb dieser Kritikpunkt hier kaum von Bedeutung ist.

4.1.5.3 Reader Komponenten

Das Konzept von Readern in Zusammenhang mit Korpora wurde bereits in Abschnitt 4.1.3.1 vorgestellt; hier werden daher nur die Gemeinsamkeiten und Unterschiede von Readern und Komponenten erläutert.

Reader werden von der Basisklasse `TeslaReaderComponent` abgeleitet, bei der es sich wiederum um eine Subklasse von `TeslaComponent` handelt, die um zwei weitere Methoden ergänzt wurde, welche für die manuelle Zuordnung von Readern und Signalen benötigt werden. Da, wie in Abschnitt 4.1.3 gezeigt, in Tesla nicht nur statisch definierte Dokumentmengen als Korpora verwendet werden, sondern Korpora auch dynamisch um eigene Dateien erweitert werden können, ist es notwendig, Anwender bei der Auswahl eines geeigneten Readers für solche Dateien zu unterstützen. Dies führt dazu, dass Reader im Gegensatz zu Komponenten nicht nur innerhalb eines Experiments auf dem Server verwendet werden, sondern auch (ohne vorherige Konfiguration) eine clientseitige Voransicht unterstützter Dokumente generieren müssen, wie in Abbildung 4.10 dargestellt, weshalb eine Erweiterung der Klassenhierarchie erforderlich ist.

Reader müssen zusätzlich ein mit `@SignalOutputAdapter` annotiertes Feld vom Typ `ISignalOutputAdapter` sowie ein mit `@SignalInputAdapter` annotiertes Feld des Typs `ISignalInputAdapter` definieren. Diese werden analog zu den in Abschnitt 4.1.5.1 beschriebenen Ein- und Ausgabeschnittstellen für DataObjects verwendet, dienen jedoch dazu, das in einem Dokument gespeicherte Signal von evtl. ebenfalls vorhandenen Annotationen zu separieren, in UTF-8-Kodierung zu überführen und zu speichern, damit es von weiteren Komponenten verarbeitet werden kann.

Enthält ein Dokument bereits Annotationen zu einem Signal, so können diese mit Hilfe der auch von Komponenten verwendeten Ausgabeschnittstelle gespeichert werden – im Gegensatz zu Komponenten ist es Readern jedoch nicht möglich, die Annotationen anderer Komponenten zu konsumieren, da sie *per definitionem* den Beginn eines Versuchsaufbaus darstellen und zu diesem Zeitpunkt noch keine Ergebnisse anderer Komponenten zur Verfügung stehen können.

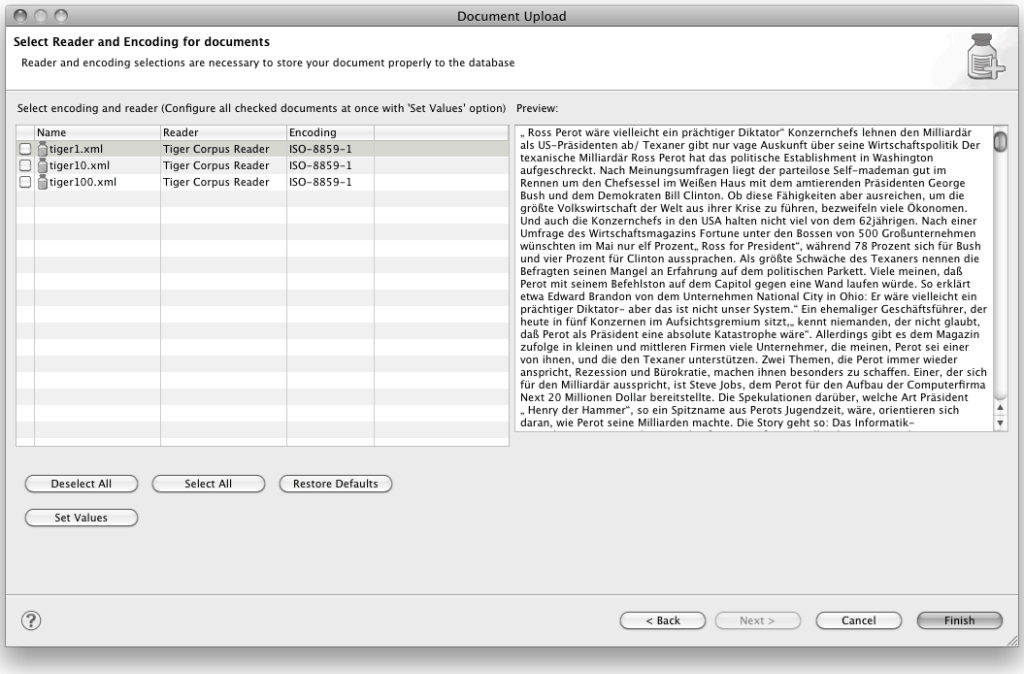

Abbildung 4.10: Upload-Dialog des Tesla Clients. Im linken Bereich sind zuvor ausgewählte Dateien dargestellt, deren Reader und Encoding modifiziert werden können, auf der rechten Seite ist die Vorschau des aktiven Dokuments zu sehen.

Um Reader von Komponenten zu unterscheiden, werden diese nicht mit der Annotation `@Component`, sondern mit `@Reader` markiert. Die Elemente der Annotationen sind im Wesentlichen identisch; Reader verfügen jedoch über eine zusätzliche *Level*-Angabe, mit der die Eignung eines Readers bei der manuellen Zuweisung zu einem Dokument bewertet werden kann. So können bspw. Dokumente des British National Corpus nicht nur vom *BNCReader*, sondern auch von *XMLReader* und *TextReader* verarbeitet werden – da letztere jedoch ein niedrigeres *Level* definieren, wird der *BNCReader* als Standard-Reader für derartige Dokumente ausgewählt.

4.1.5.4 Laufzeitmodell

Wie bereits in Abschnitt 4.1.2 erwähnt, ergibt sich die Reihenfolge der Ausführung von Komponenten in einem Experiment aus den dort modellierten Abhängigkeiten. Wird ein Experiment prozessiert, so werden zunächst alle dort genutzten Reader konfiguriert und ausgeführt. Nach erfolgreicher Ausführung eines Readers versendet das Framework asynchron eine Nachricht, woraufhin die Komponenten, deren Abhängigkeiten nun erfüllt sind, analog zu den Readern ausgeführt werden. Das Sequenzdiagramm in Abbildung 4.11 veranschaulicht den Lebenszyklus einer Komponente in Tesla.

Dieses asynchrone Vorgehen hat zur Folge, dass insbesondere in komplexen Experimenten mehrere Komponenten parallel ausgeführt und die Ressourcen des Servers besser ausgenutzt werden. Da dies jedoch nicht zwingend der Fall sein muss, wurde das Framework um eine einfache Form des *deklarativen Multithreading* ergänzt: Durch eine entsprechende Modifikation des Parameters `threadMode` der `@Component`-Annotation einer Komponente (vgl. Listing 4.2) kann eine automatisierte Parallelisierung auf Signalebene ermöglicht werden.

Wird bspw. die Einstellung `ThreadMode.INSTANCE_PER_SIGNAL` gewählt, so werden bei der Ausführung einer Komponente mehrere Instanzen der betroffenen Klasse erzeugt, konfiguriert und ausgeführt. Die von den Instanzen verwendeten Input-Schnittstellen werden in diesem Fall dahingehend modifiziert, dass die Methoden `IAccessAdapter.getAllSignalIds()` und `SignalAccessor.getSignalIterator()` nur den Teil der Signale zurückgeben, der von der jeweiligen Instanz der Komponente prozessiert werden soll. Damit eignet sich dieser Modus grundsätzlich für Komponenten, die signalbasiert arbeiten, wie etwa Tokenizer oder Parser, und entspricht der konfigurierbaren Parallelisierung in UIMA (vgl. S. 60).

Die Einstellung `ThreadMode.THREAD_PER_SIGNAL` unterscheidet sich dahingehend, dass hier nicht mehrere Instanzen einer Klasse erzeugt werden, sondern mehrere

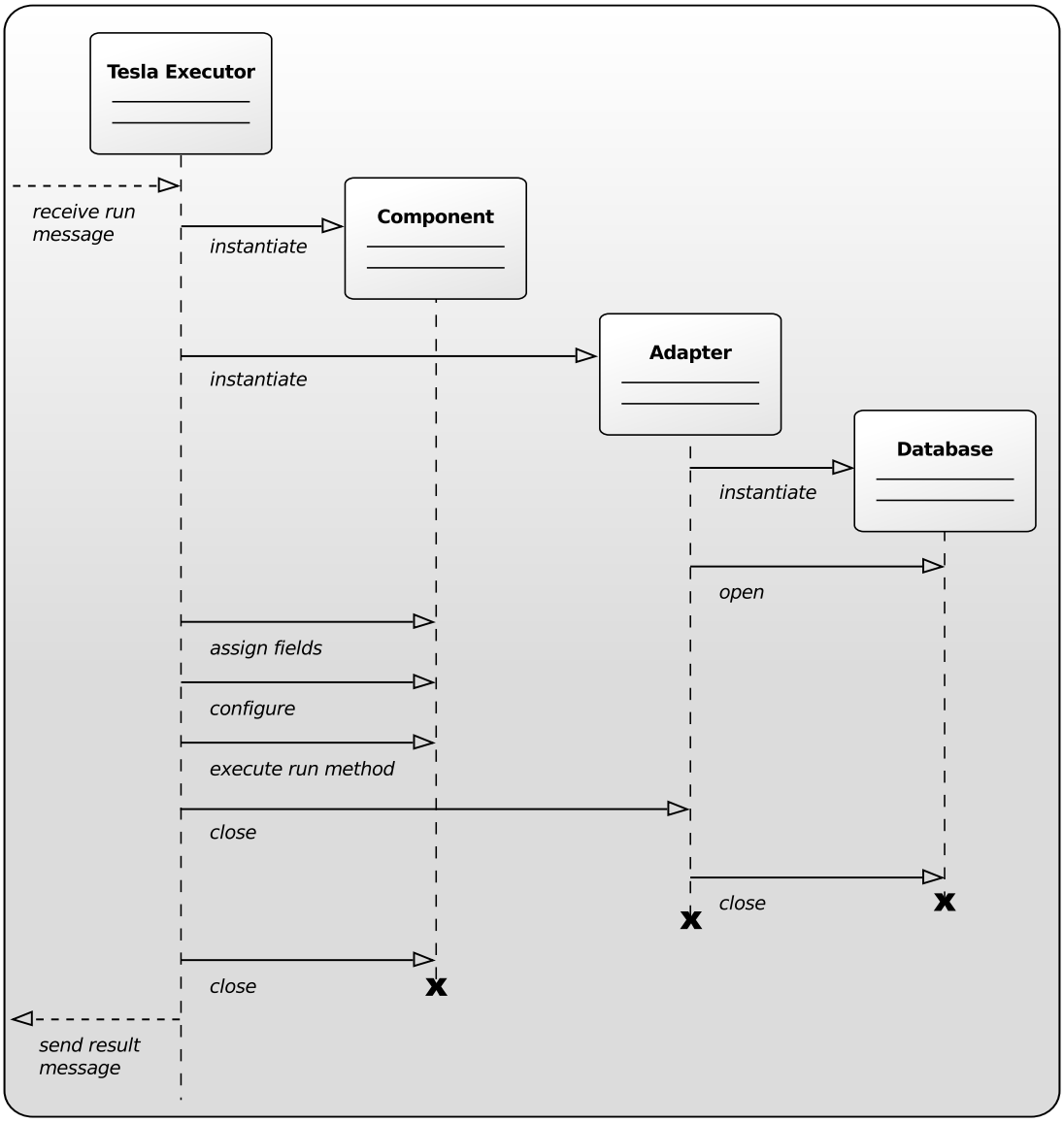

Abbildung 4.11: Lebenszyklus einer Tesla-Komponente. Die Ausführung einer Komponente wird durch eine asynchron versendete Nachricht initiiert, woraufhin zunächst die benötigten Klassen (Komponente und konsumierte bzw. produzierte Adapter) instantiiert werden. Anschließend wird den mit Metadaten versehenen Feldern der Klasse (wie bspw. Konfigurations-Optionen) der im Experiment definierte Wert zugewiesen. Nach optionalem Aufruf einer Konfigurationsmethode werden schließlich die mit `@Run` annotierten Methoden ausgeführt. Darauf folgend werden die verwendeten Adapter (und die von ihnen genutzten Ressourcen) freigegeben und das Ergebnis der Ausführung in Form einer weiteren Nachricht an den Server verschickt.

parallel laufende Threads, die auf das gleiche Objekt zugreifen. Im Unterschied zu `INSTANCE_PER_SIGNAL` muss somit lediglich eine Instanz der Komponente konfiguriert werden[108], und evtl. benötigte Daten müssen nur einmal geladen und im Arbeitsspeicher gehalten werden. Auch dieser Modus eignet sich für signalbasierte Komponenten, bringt jedoch die Einschränkung mit, dass Entwickler bei der Modifikation von Objektvariablen die Asynchronität der Verarbeitung berücksichtigen und ggfs. Vorkehrungen zur Synchronisation treffen müssen. Wird hingegen nur lesend auf Objektvariablen zugegriffen (wie etwa bei einer lexikonbasierten Komponente), so kann dieser Modus ohne weitere Änderungen am Quellcode der Komponente aktiviert werden.

Theoretisch lassen sich zwar zahlreiche signalbasierte Komponenten durch `ThreadMode.INSTANCE_PER_SIGNAL` oder `ThreadMode.THREAD_PER_SIGNAL` parallelisieren, dies ist jedoch in der Praxis nicht immer sinnvoll und nicht immer möglich. Im Falle eines Tokenizers ist bspw. keine Verbesserung des Laufzeitverhaltens zu erwarten, da die verwendete Algorithmik vergleichweise trivial ist und die Performanz in erster Linie von Ein- und Ausgabeoperationen abhängig ist, welche wiederum durch die Hardware bestimmt werden. In anderen Fällen, wie beim *Stanford Parser*, stellte sich heraus, dass die verwendete API nicht multithreading-fähig ist und die Ausführung mehrerer Instanzen des Parsers in der gleichen virtuellen Maschine zu Fehlern führt. Dadurch, dass für die Aktivierung eines dieser Modi jedoch ausschließlich ein Element der `@Component`-Annotation modifiziert werden muss, kann die Eignung für Parallelverarbeitung im Fall von signalbasierten Komponenten auch nach der eigentlichen Entwicklung einer solchen Komponente überprüft werden.

Tesla unterstützt bisher kein Rechner-Clustering, d.h. es ist bisher nicht möglich, die Parallelisierung auf mehrere Server oder Workstations auszuweiten. Die nachrichtengestützte Form der Ausführung von Experimenten und die Kommunikation zwischen Komponenten durch AccessAdapter- und DataObject-Interfaces würden es jedoch ermöglichen, das Framework um diese Funktionalität zu erweitern, ohne Änderungen an der Architektur durchführen zu müssen: So könnten bspw. bei der Erzeugung von AccessAdaptern Proxy-Klassen erzeugt werden, so dass sich Methodenaufrufe über *Remote Method Invocation* an weitere Server delegieren ließen. Durch Analyse der Abhängigkeiten zwischen Komponenten wäre es zudem möglich, unabhängige Teile eines Experiments auf getrennten Einheiten eines Clusters ausführen zu lassen.

[108] Aus diesem Grund wurde, wie bereits auf Seite 110 angedeutet, die optionale `@Configure`-Annotation eingeführt: Im Gegensatz zur Run-Methode wird die Konfigurationsmethode in diesem Modus nur einmal aufgerufen.

4.1.6 Evaluation

Datenanalyse ist ein wesentlicher Bestandteil experimenteller Forschung, unabhängig davon, ob diese manuell oder (teil-) automatisch durchgeführt wird. Für diesen Zweck bietet Tesla verschiedene Werkzeuge und Schnittstellen, mit denen die Analyse von Ergebnissen unterstützt wird. So lassen sich für ein ausgeführtes Experiment allgemeine Informationen bezüglich der Komponenten darstellen, wie in Abbildung 4.12 gezeigt – dabei handelt es sich sowohl um generische Informationen zu benötigter Laufzeit und zum Zustand der Komponente, als auch um individuelle Daten, die von den verwendeten Komponenten während ihrer Ausführung erzeugt wurden. Neben dieser Schnittstelle können Entwickler zudem tabellarische Zusammenfassungen der Ergebnisse einer Komponente generieren, die als CSV- und LaTeX-Dateien exportiert werden können. Dies ist u.a. dann sinnvoll, wenn eine Komponente Daten generiert, die unmittelbar in eine wissenschaftliche Arbeit übernommen oder mit Hilfe externer Werkzeuge in ein Diagramm konvertiert werden können – viele der in Kapitel 5 abgebildeten Tabellen und Diagramme wurden auf diese Weise erzeugt.

Für eine detaillierte Analyse können bspw. Evaluationskomponenten verwendet werden, um die Qualität der von anderen Komponenten produzierten Ergebnisse zu überprüfen – dieses Vorgehen eignet sich besonders dann, wenn ein Gold-Standard vorhanden ist, mit dem Daten abgeglichen werden können, so dass bspw. Precision und Recall (wie auf Seite 15 definiert) extrahiert werden können. Allerdings ist ein solcher Ansatz nicht für sämtliche Komponenten einsetzbar: Können die Ergebnisse einer Komponente beispielsweise nicht eindeutig als korrekt oder falsch klassifiziert werden, so ist eine automatisierte Evaluierung u.U. nicht sinnvoll, nicht hinreichend oder nicht möglich.

Eine manuelle Datenanalyse steht jedoch im Widerspruch zum Tesla Role System und dem dort definierten interfacebasierten Konzept von Datenstrukturen und Zugriffsmethoden, welches in eine statische Repräsentation überführt werden müsste, um von Anwendern evaluiert zu werden. Im Rahmen der Entwicklung von Tesla wurde zunächst eine erweiterbare Visualisierungsschnittstelle entworfen, dank derer unterschiedliche Formen der Visualisierung (bspw. in Form farblich hervorgehobener Auszeichnungen, vergleichbar mit der in Abbildung 3.2 dargestellten Visualisierung in GATE auf Seite 56, aber auch in Form von Baum- und Tabellenstrukturen) implementiert werden konnten. Dies führte jedoch zu einer unerwünschten Abhängigkeit zwischen Visualisierungen und Rollendefinitionen, da Visualisierungs-Plugins an die in DataObject- und AccessAdapter-Interfaces definierten Methoden angepasst werden mussten, und da umgekehrt die Konzeption neuer Rollen dadurch beeinflusst wurde, dass vorhandene Visualisierungs-Plugins berücksichtigt wurden,

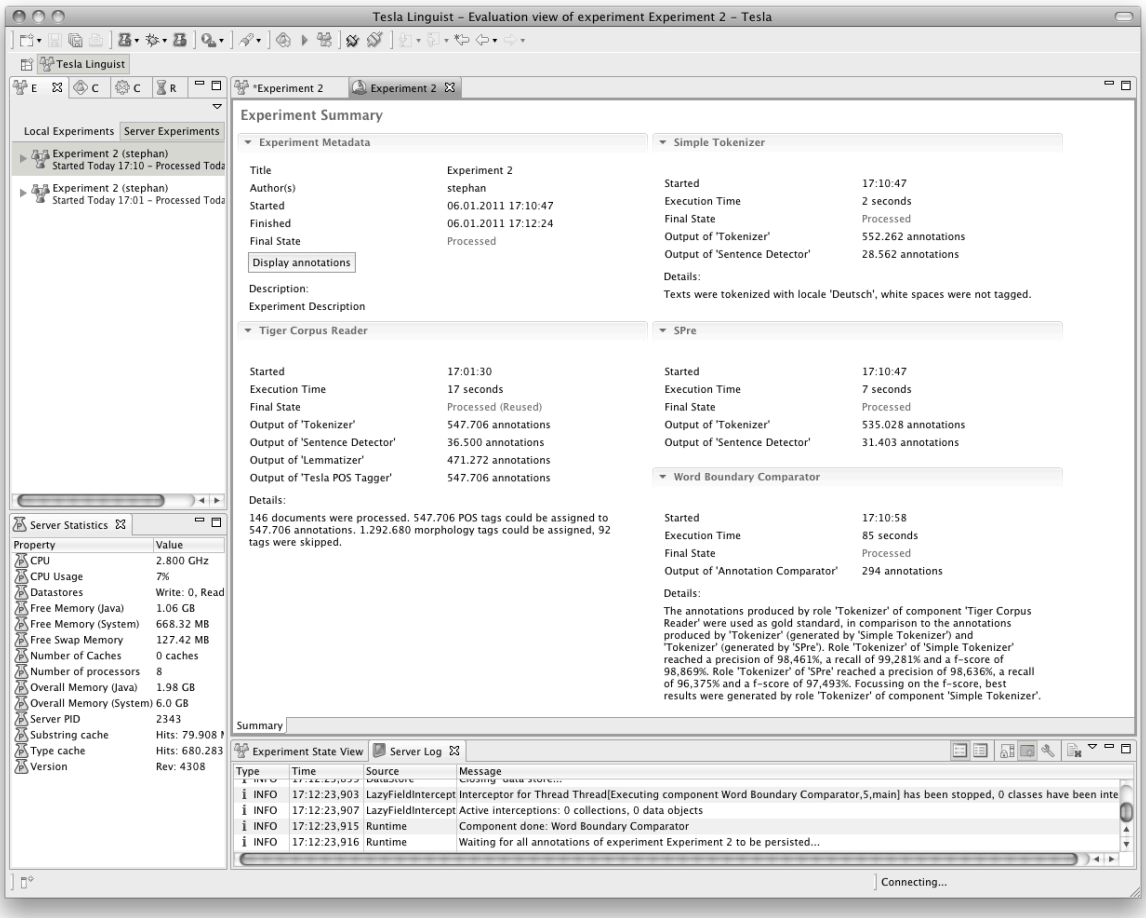

Abbildung 4.12: Zusammenfassung der Ergebnisse eines ausgeführten Experiments. Allgemeine grundlegende Informationen zu einer Komponente (wie etwa der benötigten Laufzeit, der Anzahl produzierter Annotationen oder dem aktuellen Status) und Konfigurationsparametern werden generisch erzeugt und können von Komponenten um zusätzliche individuelle Informationen erweitert werden – so gibt bspw. die *Sentence Length Filter* -Komponente Auskunft über die Anzahl akzeptierter Sequenzen. Im Experiment-Bereich der Übersicht (links oben) sind die Schaltflächen zum Export von Annotationen und komponentenspezifischer tabellarischer Zusammenfassungen dargestellt (vgl. auch Abbildung 4.13).

Abbildung 4.13: Auswahl der zu exportierenden Rollen und DataObject-Methoden in Teslas Export-Wizard. Links ist die Auswahl von Rollen und Dokumenten dargestellt, auf der rechten Seite sind die exportierbaren Methoden eines DataObjects aufgeführt. Nicht dargestellt ist die Auswahl der AccessAdapter-Methode, die für den Zugriff auf die DataObject-Instanzen verwendet werden soll.

um die Implementation neuer Plugins zu vermeiden.

Weiterhin sollte Tesla nicht als geschlossenes System implementiert werden, sondern eine Wiederverwendung und Weiterverarbeitung der generierten Daten ermöglichen, ohne programmatisch auf Elemente des Rollensystems zugreifen zu müssen – daher wurde dieser Ansatz zugunsten einer generischen Möglichkeit des Exports von Daten, die im Folgenden beschrieben wird, verworfen.

Einer der wesentlichen Kritikpunkte an den in Kapitel 3 beschriebenen Komponentensystemen besteht darin, dass die Fokussierung auf ein datenorientiertes Format nicht hinreichend ist, um eine flexible Schnittstelle zum Datenaustausch zwischen Komponenten zu definieren. Dies betrifft jedoch nicht den Export dieser Daten zur Verarbeitung durch externe Programme: So führt der Export zwangsläufig dazu, dass die durch das TRS definierte dynamische Struktur von Annotationen und Zugriffsmethoden zugunsten einer statischen Repräsentation aufgegeben werden muss. Ferner impliziert der Export der Daten einen anschließenden Import durch ein weiteres Programm oder eine Aufbereitung der Daten zur unmittelbaren Nutzung – da XML diese Anforderungen erfüllt und es zudem erlaubt, strukturelle Modifikationen mit etablierten Technologien umzusetzen (s.u.), wurde der Datenexport in Tesla entsprechend realisiert.

Dabei wird jedoch ein Ansatz verfolgt, mit dem die dynamische Struktur von Annotationen in Tesla berücksichtigt werden kann und durch den Entwickler beim Design neuer Rollen nicht einschränkt werden, da die Konvertierung generisch durchgeführt wird: Anwender legen über den in Abbildung 4.13 dargestellten Export-Dialog fest, welche Rollen und Signale eines Experiments konvertiert werden sollen. Anschließend kann ausgewählt werden, welche `IAccessAdapter`-Methode die relevanten Datenstrukturen liefern soll, und welche Methoden der `DataObject`-Implementationen in XML-Elemente transformiert werden sollen. Durch dieses Vorgehen können verschiedene Ansichten der generierten Daten erzeugt werden, die unterschiedlichen Verwendungsarten der jeweiligen Rolle entsprechen.

Auf Basis der vorgenommenen Einstellungen konvertiert ein generischer Export-Mechanismus die Rückgabewerte der ausgewählten Methoden in eine XML-Darstellung.

Komplexe Datenstrukturen werden dabei automatisch in ihre einzelnen Bestandteile zerlegt, wobei die Struktur der Daten ebenso wie referentielle Identität innerhalb des Objektgraphen erhalten bleibt: Enthält der Graph mehrere Referenzen auf ein Objekt, so wird dieses dennoch nur einmal konvertiert; weitere Vorkommen werden über einen Verweis auf die Objekt-Id referenziert. Die Art der Referenzierung ist konfigurierbar – so können bspw. auch *XPath*-Ausdrücke erzeugt oder Kopien der referenzierten Objekte in die generierte Baumstruktur eingefügt werden.

Hierbei ist zu beachten, dass die Konvertierung von Daten nicht auf Basis der Felder von DataObject-Implementationen durchgeführt wird, sondern ausschließlich anhand der Methoden, die in den implementierten Interfaces definiert wurden, so dass die Konzepte des TRS (wie bspw. die Austauschbarkeit und Vergleichbarkeit unterschiedlicher Implementationen der gleichen Rolle) weiterhin berücksichtigt werden. Diese Form der Konvertierung auf Basis von Methodenaufrufen bringt allerdings technische Einschränkungen mit sich, denn zum einen können nicht sämtliche Methoden auf eine generische Art aufgerufen werden (etwa dann, wenn sie Parameter benötigen), zum anderen ist nicht jede in einem DataObject definierte Methode für den Export geeignet (wie der Aufruf von Methoden wie `getChildren()` und `getParent()` auf einem Knoten einer Baumstruktur, da dies zu einer Endlosschleife führen würde). Zudem erfordert die Auswahl der Methoden, die exportiert werden sollen, Kenntnisse über Struktur und Konzept der Interfaces einer Rolle.

Um diese Nachteile weitestgehend zu umgehen, wurde erneut auf die Möglichkeiten von Java Annotationen zurückgegriffen: `DataObject`- und `IAccessAdapter`-Methoden können nicht nur mit einer natürlichsprachlichen Bezeichnung und einer Beschreibung annotiert werden, sondern zusätzlich auch für die Konvertierung empfohlen oder von dieser ausgeschlossen werden. Zudem können default-Werte für Methoden, die primitive Datentypen oder Strings als Parameter erwarten, definiert werden, so dass diese Methoden beim Datenexport berücksichtigt werden können (vgl. Listing 4.5 und 4.6).

```
1 public interface IBigram<T extends DataObject> extends INgram<T> {
2
3    @Hint(name="Bigram Occurrences", display=Display.RECOMMENDED)
4    public Map<Annotation<T>, Annotation<T>> getOccurrences();
5
6 }
```

Listing 4.5: Anreicherung von `DataObject`-Methoden mit Metadaten. Neben einer Bezeichnung der Methode, die in der graphischen Benutzeroberfläche von Tesla verwendet wird, ist hier definiert, dass der Export der zurückgelieferten Daten empfohlen wird.

Abbildung 4.14: Der in Tesla integrierte XSL-Prozessor zur Konvertierung exportierter Daten. Im Editor-Bereich (links) ist die dynamisch erzeugte XML-Repräsentation der von einer Komponente generierten Annotationen zu sehen, rechts daneben der XSL-Prozessor. Verschiedene Konvertierungsschritte können ausgewählt (rechts oben) und einer Prozessierungs-Pipeline hinzugefügt werden (rechts unten).

```
1 public interface IBigramAccessAdapter<T extends IBigram> extends ICategoryAccessAdapter<T> {
2
3   @Exportable(displayName="All Bigrams", description="All detected Bigrams", recommended=false)
4   public InputIterator<Annotation<T>> getAllBigrams();
5
6   @Exportable(displayName="Most interesting Bigrams",
7            description="The most interesting bigrams (default value is 50)",
8            recommended=true, defaults={"50"})
9   public List<Annotation<IBigram>> getMostInterestingBigrams(int limit);
10  ...
11 }
```

Listing 4.6: Anreicherung von `IAccessAdapter`-Methoden mit Metadaten. Analog zu DataObjects werden Methoden ebenfalls um natürlichsprachliche Beschreibungen und Export-Empfehlungen ergänzt; zusätzlich zeigt das Beispiel die Definition von Standard-Werten für primitive Parameter.

Im Unterschied zu den zu Beginn dieses Abschnittes beschriebenen Überlegungen, Visualisierungen anhand von Rollendefinitionen zu implementieren, ist die hier realisierte Lösung flexibler, während sich der Mehraufwand für Entwickler auf die Annotation von Methoden beschränkt (die zudem, ebenso wie die in Abschnitt 4.1.5.2 beschriebene Annotation von Komponenten, als weitere Form des *Literate Programming* betrachtet werden kann). Des Weiteren ist eine Visualisierung der Daten nicht ausgeschlossen – im Gegensatz zum ersten Ansatz muss diese jedoch auf Basis der XML-Repräsentation der Daten implementiert werden, was sowohl Vor- als auch Nachteile mit sich bringt: So können Daten vor der Visualisierung mit Werkzeugen zur XML-Verarbeitung wie XSLT aufbereitet werden, um unerwünschte Elemente zu entfernen oder die Struktur der Daten an die Anforderungen eines Visualisierungs-Plugins anzupassen. Der in Teslas graphische Oberfläche integrierte XSLT-Prozessor (vgl. Abbildung 4.14) ermöglicht es, grundlegende XSLT-Operationen[109] vorzunehmen und kann um zusätzliche Skripte erweitert werden.

Die so aufbereiteten Daten können anschließend entweder mit den integrierten

[109] Die in Java integrierte XSLT-Unterstützung ist auf den Funktionsumfang von XSLT 1.0 beschränkt, wodurch u.a. deutlich weniger String-Funktionen zur Verfügung stehen als in XSLT 2.0. Es standen jedoch zum Zeitpunkt der Implementation dieser Funktion keine leistungsfähigeren XSLT-Prozessoren zur freien Nutzung und Verbreitung zu Verfügung, so dass diese Einschränkung in Kauf genommen werden musste.

Visualisierungs-Plugins dargestellt oder als XML-Datei exportiert werden. Allerdings zeigt sich hier auch, dass XML und XSLT ungeeignet für Speicherung und Verarbeitung großer Datenmengen sind: So können die von Tesla exportierten Daten in XML-Repräsentation mehrere Gigabyte groß werden, was, wie Listing 4.7 zeigt, nicht nur durch die wenig platzsparende Syntax von XML, sondern auch durch die methoden- und interfacebasierte Datenrepräsentation in Teslas XML-Format bedingt ist.

Um die objektorientierte Struktur der exportierten Daten zu übernehmen und einen Informationsverlust weitestgehend zu vermeiden, wird jedes Interface, das von einem DataObject erweitert wird, als einzelnes Element im XML-Baum repräsentiert – dies schließt auch Marker-Interfaces (wie `IPunctuation` in Zeile 24 in Listing 4.7), die keine Methoden definieren, sondern lediglich zur Markierung von Klassen oder zur Modellierung von Hierarchien verwendet werden, mit ein.

Teslas Konvertierungsmechanismus unterstützt zudem neben primitiven Datentypen, Strings und DataObject-Subinterfaces auch die Abbildung von Collections, Maps, (eindimensionalen[110]) Arrays und Iteratoren, wobei in der derzeitigen Implementation nicht zwischen primitiven und komplexen Datenstrukturen unterschieden wird: Ein einzelnes Element einer Collection oder eines Arrays wird, wie in den Zeilen 38 bis 40 in Listing 4.7 gezeigt, als `item` in einem `collection`-Element abgebildet. Diese Art der Repräsentation ist jedoch bspw. für hochdimensionale Vektoren nicht geeignet, da die Größe der generierten Dateien eine effiziente Prozessierung verhindert.[111]

Trotz der genannten Einschränkungen bietet Tesla durch den XML-Export die Möglichkeit, die Ergebnisse eines Experiments in anderen Anwendungen weiterzuverarbeiten und dort bspw. zur Visualisierung aufzubereiten oder einer Analyse zu unterziehen – grundsätzlich sollte es über die beschriebene Schnittstelle zudem möglich sein, das von Tesla generierte Format in die von UIMA verwendete *Common Analysis Structure* zu überführen.

Mit dem Export von Daten ergibt sich zwar zwangsläufig der Verlust erweiterter Funktionalität, die das Tesla Role System gegenüber datenorientierten Modellen bietet, dies kann jedoch nicht vermieden werden. Durch die Möglichkeit, die Export-Einstellungen zu modifizieren und gemäß den Anforderungen der folgenden Verarbeitung in einem externen Programm anzupassen, kann die Dynamik des TRS jedoch zumindest teilweise in diesen Schritt mit einbezogen werden.

[110] Mehrdimensionale Arrays können, falls gewünscht, in binärer Form gespeichert werden.

[111] Dies gilt zumindest dann, wenn die Verarbeitung der Daten voraussetzt, dass die XML-Struktur vollständig im Speicher gehalten wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<tesla>
  <signals>
    <content id="1023529497_6028300e3543e6bdc0d8c343881cded8">&#8222; Ross Perot ...
        zuvor.</content>
  </signals>
  <annotations>
    <signal id="1023529497_6028300e3543e6bdc0d8c343881cded8">
      <role id="1634610">
        <method name="All Annotations in Range" role="Tokenizer" id ="getAnnotations">
          <collection id="1">

            <annotation id="an_862017321697281" typeid="-2087091504">
              <range signalid="1023529497_6028300e3543e6bdc0d8c343881cded8" left="0"
                  right="1"/>
              <dataobject id="do_862017321697281">
                <interface class="IPartOfSpeech"/>
                <interface class="ISubSequence"/>
                <interface class="ISingleValueCategory">
                  <object dt="recommended" rt="String" method="getCategory"
                      display="Category">Punctuation</object>
                </interface>
                <interface class="DataObject">
                  <object dt="supported" rt="long" method="getId" display="Unique
                      Identifier">862017321697281</object>
                </interface>
                <interface class="IPunctuation"/>
                <interface class="ICategory"/>
                <interface class="IAnchoredElement"/>
                <interface class="IHierarchicalPartOfSpeech"/>
                <interface class="IWord"/>
                <interface class="ILabeledElement">
                  <object dt="recommended" rt="String" method="getLabel"
                      display="Label">Punctuation</object>
                </interface>
                <interface class="IToken"/>
                <interface class="IPunctuationToken">
                  <object dt="recommended" rt="String" method="getPunctuationType"
                      display="Punctuation Type">Quotation mark (&quot;)</object>
                </interface>
                <interface class="IHierarchicalCategory">
                  <object dt="recommended" rt="Set" method="getSubCategories" display="Sub
                      Categories" id="2">
                    <collection type="set" vcls="String">
                      <item>Quotation mark (&quot;)</item>
                    </collection>
                  </object>
                </interface>
              </dataobject>
            </annotation>

          </collection>
        </method>
      </role>
    </signal>
  </annotations>
</tesla>
```

Listing 4.7: Ausschnitt eines nach XML exportierten Experiments.

4.1.7 Labor

Dieser Abschnitt behandelt die graphische Benutzeroberfläche von Tesla (sofern noch nicht in Zusammenhang mit den bisher vorgestellten Konzepten geschehen), welche als laborative Arbeitsumgebung für die Entwicklung neuer Komponenten ebenso wie für den Aufbau und die Ausführung von Experimenten dient. Diese Arbeitsumgebung wurde auf Basis des Eclipse Frameworks realisiert, dessen Konzepte in Abschnitt 4.1.7.1 kurz vorgestellt werden, um anschließend in Abschnitt 4.1.7.2 auf die Tesla-spezifischen Erweiterungen der Java-Entwicklungsumgebung einzugehen und in Abschnitt 4.1.7.3 die Umsetzung des virtuellen Labors vorzustellen.

4.1.7.1 Eclipse als Framework für Rich Client Applications

Eclipse wurde ursprünglich von IBM konzipiert und implementiert, um eine gemeinsame Basis der *VisualAge*-Entwicklungsumgebungen bereitzustellen[112], wobei wiederverwertbare Programmteile in Form von Plugins definiert wurden, um die Entwicklung unterschiedlicher IDEs zu vereinfachen. 2001 wurde eine Veröffentlichung unter einer Open Source Lizenz angekündigt, und in der 2004 veröffentlichten Version 3 wurden die Plugin-Schnittstellen derart optimiert, dass auch ein Einsatz außerhalb des Kontextes von Softwareentwicklung, als Basis von *Rich Client*-Anwendungen, möglich wurde, wie McAffer & Lemieux (2005, S. 25) festhalten:

> In Eclipse, everything is a plug-in [... and all] plug-ins interact via the extension registry and public API classes. These facilities are available to all plug-ins. There are no secret back doors or exclusive interfaces [..., so that one can use it] to build portable, highly scalable, and customizable UIs that have the look and feel of the platform on which you are running.

Ein javabasiertes Plugin-Framework muss bezüglich traditioneller Konzepte der Objektorientierung, wie Vererbung oder Kompositionalität, einerseits zusätzliche Flexibilität bieten, andererseits aber auch die Möglichkeiten der Erweiterung einschränken. Wie Bloch (2008, S. 81) argumentiert, ist es „safe to use inheritance within a package, where the subclass and the super-class implementations are under the control of the same programmers [... or] when extending classes specifically designed and documented for extension“ – dies ist jedoch im Kontext eines Plugin-Frameworks nicht der Fall. Bloch empfiehlt daher, dass Komposition grundsätzlich gegenüber Vererbung (bei Klassen) bevorzugt verwendet

[112] vgl. http://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from%3F.

werden sollte, um andernfalls u.U. entstehende Seiteneffekte zu vermeiden, und schlägt bzgl. der Architektur von Komponentenframeworks die Kombination von Komposition und Interface-Vererbung als beste Strategie vor (vgl. Bloch 2008, Seite 93ff).[113]

Eine solche Strategie erfordert allerdings einen Mechanismus, der Interfaces (und abstrakte Klassen) mit deren konkreten Implementationen verbindet. Soll bspw. die Funktionalität eines Plugins A erweitert werden, so könnte ein zweites Plugin B entwickelt werden, welches die relevanten Interfaces oder Klassen aus A extendiert. Dies führt jedoch nicht nur zu einer (unvermeidlichen) Abhängigkeit von B zu A, sondern verhindert, dass A auf die neu implementierten Klassen zugreifen kann, da diese zum Zeitpunkt der Kompilierung von A nicht existierten und somit auch nicht referenziert werden konnten.

Der *Extension Point*-Mechanismus von Eclipse löst dieses Problem, indem eine Registrierungsschnittstelle eingeführt wird, die den Klassen aus A eine Verwendung der Erweiterungen aus B erlaubt, während gleichzeitig sichergestellt wird, dass nur die Schnittstellen, die vom Entwickler von A dafür ausgelegt wurden, extendiert werden können: Soll ein Plugin erweiterbar sein, so kann ein Extension Point erstellt werden, in dem u.a. definiert wird, welche Interfaces oder Klassen erweiterbar sind. Zur Laufzeit der Plugins können Anfragen über vorhandene Implementationen des Extension Points an die Registrierungsschnittstelle gestellt und die zurückgegebenen Erweiterungen anschließend instantiiert und verwendet werden (vgl. Clayberg & Rubel 2006, S. 595ff).

Dieser Mechanismus steht nicht nur Drittentwicklern zur Verfügung, sondern wurde auch für die Entwicklung von Eclipse verwendet, so dass zum einen zahlreiche Extension-Points zur Verfügung stehen, mit denen die IDE bspw. um neue Menüpunkte und Editoren (wie bei den in Abschnitt 3.2.4 beschriebenen UIMA-Plugins der Fall) erweitert werden kann[114], zum anderen aber auch Funktionen entfernt werden können, um das Framework als Basis einer Anwendung außerhalb der Softwareentwicklung zu verwenden.

Im hier vorliegenden Anwendungsfall wurde von beiden Möglichkeiten Gebrauch gemacht: Zum einen wurde, wie im folgenden Abschnitt beschrieben, die IDE von Eclipse um Tesla-spezifische Funktionen erweitert, zum anderen wurde aber auch eine RCP-Anwendung erzeugt, die sämtliche benötigten Plugins (einschließlich eines Tesla-Servers) bündelt. Eine weitere, leichtgewichtigere Anwendung für Nutzer, die lediglich an der Ausführung von Experimenten interessiert sind, aber keine IDE-Funktionalität benötigen, wäre bei Bedarf ebenfalls schnell realisierbar.

[113] Diese Überlegung deckt sich mit den in Abschnitt 2.3 und 4.1.4 diskutierten Anforderungen und Argumenten, die zur Entwicklung des TRS führten.

[114] Die vorliegende Arbeit wurde bspw. mit Hilfe des LaTeX-Plugins *TeXlipse* (http://texlipse.sourceforge.net/) in Eclipse verfasst.

4.1.7.2 Tesla IDE

Um die Entwicklung eigener Komponenten zu erleichtern, wurde u.a. ein vollständiger Tesla Server im Client integriert, der über die graphische Benutzeroberfläche konfiguriert und in einer separaten virtuellen Maschine gestartet werden kann. Hierfür wurde der standardmäßig für die Ausführung von Programmen verwendete Extension Point von Eclipse genutzt, wodurch nicht nur diverse Einstellungsmöglichkeiten (wie etwa die Zuweisung von Arbeitsspeicher oder die Konfiguration von Umgebungsvariablen) zur Verfügung stehen, sondern wodurch der Tesla Server auch in einem *Debug*-Modus gestartet werden kann. Ist dies der Fall, so können Entwickler sämtliche von Eclipse zur Fehlersuche angebotenen Optionen verwenden und bspw. die Ausführung eines Experiments durch Setzen eines *Break Points* im Quellcode einer Komponente unterbrechen, um anschließend den Programmablauf ebenso wie die Belegung der verwendeten Variablen schrittweise zu verfolgen.

Weder die Ausführung des lokalen Servers noch die Art der Kommunikation zwischen Client und Server unterscheidet sich dabei von der Verwendung eines dedizierten Servers. Verbindet sich der Client mit einem Server, so werden zunächst alle für die Entwicklung neuer Komponenten benötigten Bibliotheken auf den Client übertragen, so dass beim Kompilieren eigener Komponenten stets die aktuellen Bibliotheken verwendet werden.

Um eine neue Komponente zu entwickeln, kann ein dafür implementierter *Wizard* verwendet werden, bei dem es sich um eine modifizierte Version der von Eclipse zur Verfügung gestellten Wizards für neue Java-Projekte und -Klassen handelt. Dies reduziert insbesondere bei Entwicklern, die bereits mit Eclipse vertraut sind, die benötigte Einarbeitungszeit[115], zumal der Wizard anhand der Anwendervorgaben eine Beispielkomponente generiert, die unmittelbar eingesetzt werden kann: Sämtliche im Arbeitsverzeichnis von Eclipse enthaltenen Komponenten werden von einem lokal gestarteten Server automatisch geladen und können somit auch in Experimenten verwendet werden.

Die Tesla-IDE vereinfacht zudem die Verwendung des Tesla Role System bei der Implementation neuer Komponenten: Da eine konsumierte Rolle nicht nur durch eine eindeutige ID (in Form eines Strings) referenziert wird, sondern zusätzlich auch der rollen-spezifische IAccessAdapter entsprechend der Rollendefinition typisiert werden muss, können bei der

[115] Soweit möglich orientieren sich alle für Tesla implementierten Erweiterungen der graphischen Oberfläche an den Designvorgaben und der Benutzerführung von Eclipse: So ist bspw. auch das Fenster zur Konfiguration und Ausführung eines lokalen Servers eine Erweiterung des Dialogs, der von Eclipse zur Konfiguration von Java-Anwendungen verwendet wird, wodurch etwa die Modifikation von Systemvariablen vereinfacht wird. Gleiches gilt für den in Abbildung 4.16 dargestellten Rollen-Editor, der an den von Eclipse zur Bearbeitung von Plugins verwendeten Editor angelehnt ist.

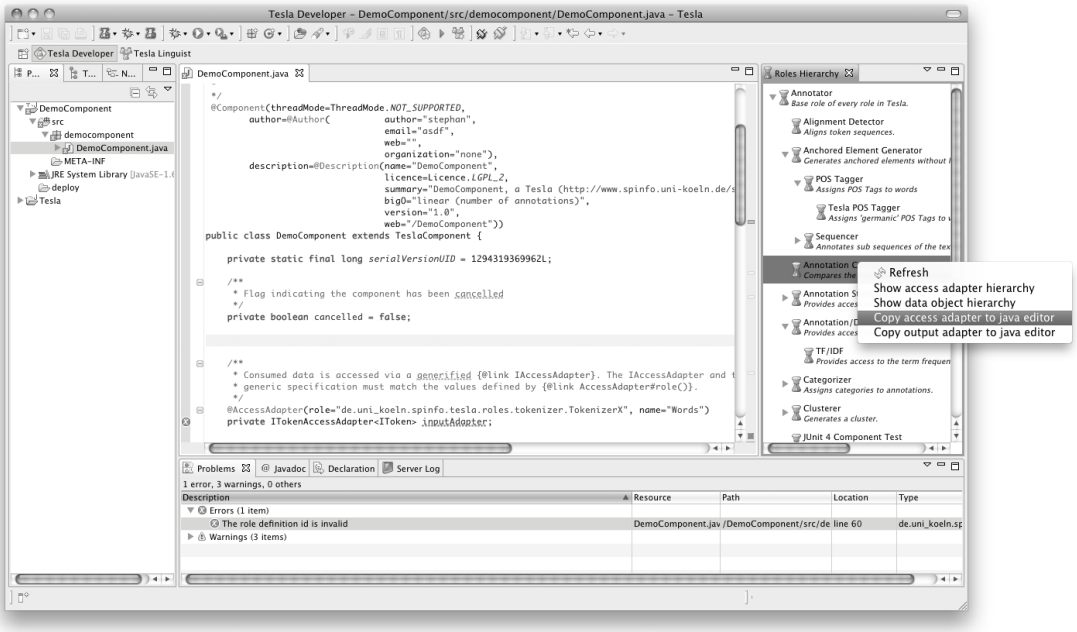

Abbildung 4.15: Erweiterung der Eclipse-IDE um Tesla-spezifische Funktionen. Das Kontextmenü des *Role Hierarchy*-View ermöglicht es, den für die Deklaration eines Access- oder Output-Adapter-Feldes benötigten Code in den Quellcode einer Tesla-Komponente zu übertragen. Bei der Kompilierung einer Komponente werden zudem konsumierte und produzierte Rollen auf semantische Korrektheit überprüft; ggfs. vorhandene Fehler werden im *Problems*-View dargestellt.

Einbindung einer neuen Rolle verschiedene Fehler auftreten. Gleiches gilt für die Definition produzierter Rollen, wobei hier weitere Angaben erforderlich sind, was die Wahrscheinlichkeit von logischen Fehlern (die im Gegensatz zu syntaktischen Fehlern erst zur Laufzeit einer Komponente erkannt werden könnten) erhöht. Zur Lösung dieses Problems wurde der Plug-In-Mechanismus von Eclipse genutzt, um einen Tesla-spezifischen *Builder*[116] in den Kompilierungsprozess zu integrieren, der automatisch ausgeführt wird, wenn eine Java-Klasse eines Komponentenprojekts modifiziert wird. Der Builder greift dabei nicht in den Kompilierungsprozess ein (so dass andere Build-Tools wie *Ant* oder *Maven* weiterhin verwendet werden können), analysiert aber die in einer Tesla-Komponente enthaltenen Metadaten. Wird dabei ein logischer Fehler erkannt (beispielsweise weil eine `@AccessAdapter`-Annotation ungültig ist, da die referenzierte Rolle nicht mit dem referenzierten DataObject kompatibel ist, wie in Abbildung 4.15), so wird die Deklaration des Feldes im Editor markiert, und es wird eine entsprechende Fehlermeldung dargestellt.

[116] Mit Hilfe dieses *Extension Points* kann die IDE von Eclipse u.a. um Unterstützung für zusätzliche Programmiersprachen erweitert werden.

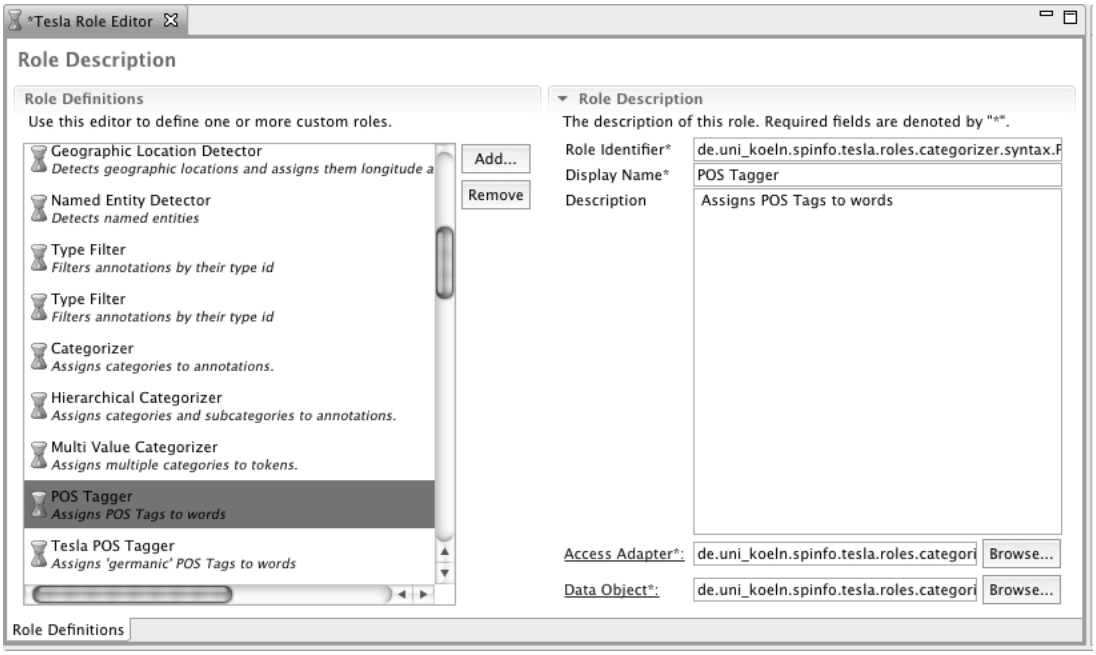

Abbildung 4.16: Screenshot von Teslas Rollen-Editor. Auf der linken Seite sind alle in einer `roles.xml`-Datei definierten Rollen aufgeführt, rechts sind die Details zur ausgewählten Rolle abgebildet.

Zusätzlich zu dieser Unterstützung bietet die Tesla-IDE jedoch auch die Möglichkeit, die für Ein- und Ausgabe benötigten Felder automatisch generieren zu lassen: So wurde ein *View*[117] implementiert, der eine Repräsentation der Rollenhierarchie des TRS dargestellt (wie am rechten Rand von Abbildung 4.15 zu sehen) und der zur Codegenerierung verwendet werden kann. Nach Auswahl der gewünschten Rolle und anschließendem Aufruf des entsprechenden Elements im Kontextmenü generiert Tesla den benötigten Java-Quellcode[118], der in die geöffnete Javaklasse eingesetzt wird.

Für die Definition neuer Rollen wurde Eclipse um einen Wizard sowie den in Abbildung 4.16 dargestellten Editor erweitert, der nicht nur von der XML-Kodierung der Rollendefinitionen (vgl. Listing 4.1 auf Seite 97) abstrahiert, sondern (durch Verwendung von in Eclipse integrierten Werkzeugen) auch die Referenzierung der benötigten Interfaces vereinfacht.

Trotz der hier beschriebenen Unterstützungen bei der Entwicklung von Komponenten ist Teslas IDE in gewissem Sinne unvollständig – so wird bspw. noch keine Hilfe oder

[117] Im Kontext von Eclipse bezeichnet dieser Begriff einen getrennten Bereich im Hauptfenster, der zur Darstellung und Modifikation von Daten genutzt werden kann (siehe auch Abbildung 4.17).

[118] Der für `IOutputAdapter` generierte Quellcode muss anschließend manuell ergänzt werden, da u.a. die Bezeichnungen der Klassen, die eine Rolle implementieren, nicht automatisch bestimmt werden können.

Abbildung 4.17: Die *Linguist*-Perspektive der graphischen Oberfläche von Tesla. Auf der linken Seite sind *Experiments* View (1), *Components* View (2), *Corpus Manager* View (3) und *Roles Hierarchy* View (4) abgebildet. Der Editor-Bereich zeigt die Visualisierung eines Textes des TüBa-D/S mit den vom Reader hinzugefügten Annotationen zur syntaktischen Struktur. Darunter sind *Experiment State* View (5) und *Server Log* View dargestellt (6).

Validierung bei der Erstellung neuer DataObject-Subinterfaces geboten, auch sind keine Tesla-spezifischen Werkzeuge zum Refactoring integriert, so dass bspw. Änderungen an einer Rollendefinition manuelle Anpassungen erfordern. Dank der Verwendung von Java-Annotationen können jedoch die von Eclipse bereitgestellten Mechanismen gegenüber den in Abschnitt 3.2.4 vorgestellten UIMA-Werkzeugen verstärkt genutzt werden.

4.1.7.3 Virtueller Arbeitsplatz

Um innerhalb von Eclipse zwischen verschiedenen Anwendungsfällen zu unterscheiden (bspw. zwischen Entwicklung und Fehlersuche), verwendet das Framework *Perspektiven*, durch die u.a. Views und Menüs gruppiert werden, so dass die graphische Oberfläche schnell an die jeweilige Verwendung angepasst werden kann.[119] Für Tesla wurden zwei Perspektiven entworfen, die auf die Entwicklung von Komponenten bzw. die Ausführung von Experimenten ausgelegt sind. Während es sich bei der *Developer*-Perspektive lediglich um eine Erweiterung der bereits integrierten Perspektive zur Entwicklung von Java-Programmen handelt (so wurde bspw. der im vorherigen Abschnitt vorgestellte View zur Darstellung der Rollenhierarchie hinzugefügt), vereint die in Abbildung 4.17 gezeigte *Linguist*-Perspektive sämtliche Elemente der graphischen Benutzeroberfläche, die für die Verwaltung und Konfiguration von Experimenten benötigt werden.

Am linken Rand sind vier Views zu sehen, die jeweils eines der in diesem Kapitel beschriebenen Konzepte repräsentieren: An oberster Stelle der *Experiments* View (1), in welchem lokal und serverseitig vorhandene Experimente dargestellt sind, die über das Kontextmenu des Views verwaltet (also bspw. gelöscht oder evaluiert) werden können. Im darunter liegenden *Components* View (2) werden die serverseitig verfügbaren Komponenten angezeigt, welche durch Doppelklick oder Drag-And-Drop zu einem Experiment hinzugefügt werden können. Dies ist ebenfalls mit den im dritten View dargestellten Korpora und Document Selections möglich, wobei dieser als *Corpus Manager* View (3) zusätzliche

[119] Das TextGrid-Framework nutzt bspw. Perspektiven, um zwischen verschiedenen Anwendungsfällen (wie XML-Editor oder Suchmaske) umzuschalten; Eclipse selbst bietet standardmäßig bspw. Entwicklungs- und Debug-Perspektiven für die Java-Programmierung an.

Verwaltungsfunktionen (wie Hinzufügen/Entfernen von Document Selections und Dokumenten oder die Suche nach Dokumenten anhand von Metadaten und/oder Volltext) bietet. Der an unterster Stelle stehende *Roles Hierarchy* View (4) zeigt die Hierarchie des Tesla Role System an, die beim Start des Servers anhand der definierten Rollen erzeugt wird. Rechts von diesem werden im *Experiment State* View (5) die Komponenten, die zum aktuellen Zeitpunkt auf dem Server ausgeführt werden, einschließlich ihres aktuellen Status dargestellt. Im *Server Log* View (6) werden schließlich die Log-Meldungen des Servers aufgeführt, während der Editor-Bereich eines der integrierten Visualisierungsmodule zur Ergebnisevaluation zeigt.

4.1.8 Zusammenfassung und Diskussion

Bevor in Abschnitt 4.2 einige technische Details von Tesla vorgestellt und diskutiert werden (etwa die Implementation von Rollen, insbesondere die dafür notwendige Anbindung der Persistierung der Daten, auf die bisher noch nicht eingegangen wurde), wird im Folgenden die Eignung von Tesla für experimentelle Anwendungsfälle, wie den in Abschnitt 2.1 beschrieben Alignment-Verfahren, diskutiert.

Die auf Seite 42 erwähnten Anforderungen (Nachvollziehbarkeit von Experimenten, Austauschbarkeit von Komponenten, Evaluation von Ergebnissen und Design komplexer Datenstrukturen einschließlich Zugriffsmethoden) lassen sich mit den hier vorgestellten Konzepten größtenteils erfüllen: Das TRS bietet sowohl die Möglichkeit der Abstraktion von proprietären Datenstrukturen als auch die Möglichkeit zur Speicherung beliebig komplexer Objekte, wodurch es dem Anspruch an Austauschbarkeit von Komponenten gerecht wird. Im Gegensatz zu anderen Typsystemen, wie der in UIMA verwendeten CAS, ist es nicht notwendig, systemseitig vorgegebene Grunddatentypen zu verwenden, zudem kann die Art des Zugriffs auf Daten durch Erweiterung des `IAccessAdapter`-Interfaces an domänen- und rollenspezifische Anforderungen angepasst werden, wie das in Abschnitt 4.1.4.2 skizzierte Beispiel einer generischen Clustering-Komponente veranschaulicht. Die für eine (in Abschnitt 2.3 am Beispiel von Alignment-Verfahren beschriebene) API-ähnliche Komponentenmodellierung notwendige Voraussetzung wird somit erfüllt.

Dem Anspruch der Nachvollziehbarkeit von Experimenten kann Tesla dadurch gerecht werden, dass es sich bei Experimenten um im- und exportierbare XML-Dokumente handelt, die redistribuiert werden können und den Versuchsaufbau einschließlich sämtlicher verwendeten Parameter beschreiben. Eine erneute Ausführung eines Experiments ist somit grundsätzlich möglich, setzt aber (wie in jedem Framework) voraus, dass die verwendeten Komponenten und Korpora verfügbar sind. Ist dies der Fall, so stellt u.a. das in Abschnitt

4.1.3 beschriebene Verfahren zur Referenzierung von Dokumenten über quasi-eindeutige, inhaltsbezogene Schlüssel sicher, dass die Umgebung, in der das Experiment erneut ausgeführt wird, identisch zur Ursprungsumgebung ist.[120]

Eine Evaluation von Ergebnissen ist in Tesla ebenfalls möglich, allerdings mit den in Abschnitt 4.1.6 diskutierten Einschränkungen, die u.a. auch die Evaluation von Alignment-Verfahren betreffen, da die Berechnung von Precision und Recall durch eine generische Komponente in diesem Fall nicht möglich (bzw. nicht hinreichend) ist: Eine semantische Evaluation erfordert hier die Berücksichtigung der internen Struktur der alignierten Elemente ebenso wie deren kategorielle Auszeichnung – letzteres geht jedoch über eine einfache Pattern-Matching-Strategie hinaus und erfordert, wie bereits in Kapitel 1 erwähnt, einen umfassend annotierten Gold-Standard. In Kapitel 5 wird daher eine einfachere Evaluationskomponente genutzt, die das TRS dafür verwendet, eine komponenten- und implementationsübergreifende Vergleichbarkeit von Strukturdetektionsansätzen zu erreichen, indem lediglich die Bereiche der ausgezeichneten Strukturen, nicht jedoch die zugewiesenen Kategorien, berücksichtigt werden (siehe Kapitel 5 und Anhang B.6.2).

Die in Abschnitt 4.1.7 beschriebene Laborumgebung und die in 4.1.5.1 vorgestellte Verwendung von Java-Annotationen zur Auszeichnung von Metadaten vereinfachen den Umgang mit den in diesem Kapitel vorgestellten Konzepten und ermöglichen es, Anwender sowohl bei der Definition von Experimenten als auch bei der Entwicklung neuer Komponenten zu unterstützen, wodurch der Einsatz in experimentellen, häufiges Refactoring erfordernden Forschungsbereichen möglich wird.

4.2 Persistenz

In 4.1 wurde u.a. das Konzept eines Laborbuchs, in dem sämtliche Schritte eines Experiments dokumentiert werden, und das zur Reproduktion der Ergebnisse verwendet werden kann, vorgestellt. Ebenso wurde dort das *Tesla Role System* erläutert, welches eine Abstraktion von konkreten Datenstrukturen und Zugriffsmethoden repräsentiert. Beiden Konzepten ist gemein, dass sie die Verwendung eines Persistenz-Frameworks erfordern, durch das eine robuste Verwaltung der anfallenden Daten umgesetzt werden kann.

[120] In zukünftigen Versionen des Systems könnte zudem ein Mechanismus integriert werden, der die Definition der im Experiment verwendeten Rollen (einschließlich referenzierter Interfaces) in der Experimentbeschreibung speichert, so dass, falls eine Komponente nicht verfügbar ist, die entsprechende Schnittstelle wohldefiniert ist und reimplementiert werden könnte. Bei Bedarf wäre es ebenfalls möglich, ein alternatives Verfahren zur Berechnung von Schlüsseln zu nutzen, das einen größeren Wertebereich umfasst und eine entsprechend geringere Wahrscheinlichkeit einer Schlüsselkollision bietet.

Verschiedene Java-Bibliotheken stellen einen Zugang zu unterschiedlichen Persistenz-Mechanismen bereit, wobei Bibliotheken wie Frameworks mit individuellen Vor- und Nachteilen verbunden sind. Mit Veröffentlichung des JDK 1.1 wurde das `Serializable` -Interface als ein einfach zu verwendender Mechanismus zur Persistierung binärer Daten eingeführt: Instanzen einer Klasse, die dieses Interface implementiert, können unmittelbar in eine binäre Repräsentation überführt und bspw. in Dateien gespeichert werden, um zu einem späteren Zeitpunkt wieder ausgelesen und rekonstruiert zu werden. Der zugrundeliegende Mechanismus übernimmt dabei die Verwaltung des Objektgraphen, wie etwa das Auflösen zirkulärer Referenzen, wodurch diese Form der Serialisierung sehr einfach zu verwenden ist. Ein Nachteil liegt jedoch darin, dass bspw. die Modifikation einer Klasse zur Inkompatibilität mit bereits gespeicherten Daten, in denen eine ältere Version der Klasse verwendet wurde, führen kann; zudem können gespeicherte Daten weder durchsucht noch modifiziert werden, ohne sie zuvor vollständig zu laden, wodurch der Ansatz für die Verwaltung großer Datenmengen ungeeignet ist. Unter anderem aus diesem Grund (aber auch, um den Einsatz von Java in Business-Anwendungen zu erleichtern und eine standardisierte Schnittstelle zu schaffen), wurde mit dem JDK 1.1 zudem die *Java Database Connectivity* (JDBC) API eingeführt, durch die es möglich ist, aus Java-Programmen auf relationale Datenbanken zuzugreifen.

Das von Codd (1970) entworfene Konzept relationaler Datenbanken wurde entwickelt, um den Anforderungen von Business-Anwendungen gerecht werden zu können: Datensätze (wie Informationen über Kunden, Rechnungen oder Produkte) werden in Form von Tabellen gespeichert, wobei jede Zeile einen Eintrag repräsentiert, und jede Spalte eine Eigenschaft des jeweiligen Datentyps (wie etwa Vor- und Nachname eines Kunden) repräsentiert. Jedem Eintrag (bzw. jeder Entität) wird i.d.R. ein eindeutiger Primärschlüssel zugewiesen, durch den Querverweise zwischen Entitäten (bspw. zwischen Kunden und Rechnungen) umgesetzt redundanzfrei werden können. Einer der größten Vorteile dieses Entwurfs liegt darin, dass er konzeptuell eng mit der Mengentheorie verbunden ist, so dass mengentheoretische Operationen wie das Erzeugen einer Vereinigung oder Schnittmenge effizient umgesetzt werden können – zu diesem Zweck wurde die *Structured Query Language* (SQL) entwickelt, die nicht nur die Möglichkeit bietet, Einträge hinzuzufügen, zu modifizieren oder zu löschen, sondern auch mengentheoretisch motivierte Query-Operationen auf den gespeicherten Daten auszuführen (vgl. Özsu & Liu 2009, S. 1929).

Innerhalb einer objektorientierten Programmiersprache ist die Verwendung relationaler Datenbanken jedoch nicht unproblematisch, da sich die Konzepte teilweise widersprechen: So muss u.a. ein objekt-relationales Mapping – d.h. eine Abbildung von Objekten auf Ta-

bellen und Relationen und umgekehrt – umgesetzt werden, falls die Funktionalität der Programmiersprache voll ausgenutzt werden soll. Zu diesem Zweck wurden verschiedene *Objekt-Relationale Datenbank Management Systeme* (ORDBMS) entwickelt, die in Form einer Abstraktionsschicht zwischen Objekt-Serialisierung und JDBC die notwendige Konvertierung vereinfachen. Allerdings können nicht sämtliche dabei auftretenden Probleme generisch gelöst werden: Erstens kann nicht jedes objektorientierte Konzept (wie etwa Vererbung) unmittelbar auf ein relationales Modell abgebildet werden, was auch als *Impedance Mismatch* (vgl. Özsu & Liu 2009, S. 1929) bezeichnet wird. Zweitens werden zusätzlich zu den in einer Java-Klasse enthaltenen Informationen weitere Metadaten benötigt, um ein objekt-relationales Mapping umzusetzen: Beispielsweise muss (bzw. sollte) ein Primärschlüssel definiert werden, auch muss zwischen verschiedenen Formen der Referenz zwischen Objekten (wie etwa 1:n oder n:n) unterschieden werden. Als Alternative zu ORDBMS wurden daher *Objekt-Orientierte Datenbank Management Systeme* (OODBMS) entwickelt, welche eine verbesserte Integration des Persistenzframeworks in eine Programmiersprache bieten und den *Impedance Mismatch* zu vermeiden versuchen – erste kommerzielle Produkte stehen seit den 1990er Jahren zur Verfügung (vgl. Edlich *et al.* 2006, S. 5). Ramakrishnan & Gehrke (2000, S. 770) stellen dazu fest:

> The fundamental difference is really a philosophy that is carried all the way through: OODBMSs try to add DBMS functionality to a programming language, whereas ORDBMSs try to add richer data types to a relational DBMS. Although the two kinds of object-databases are converging in terms of functionality, this difference in their underlying philosophies (and for most systems, their implementation approach) has important consequences in terms of the issues emphasized in the design of these DBMSs, and the efficiency with which various features are supported.

Da OODBMS keine Tabellen zur Speicherung der Daten verwenden, müssen Primärschlüssel, Assoziationstypen und andere ORM-bezogene Konzepte nicht berücksichtigt werden, wodurch die Entwicklung vereinfacht wird. Allerdings ergeben sich andere Nachteile: So können Queries bspw. nicht mehr als mengentheoretische Funktionen interpretiert werden und benötigen u.U. mehr Zeit als vergleichbare Abfragen in einer relationalen Datenbank. Abhängig von der Komplexität der gespeicherten Datenstrukturen muss zudem berücksichtigt werden, wie die jeweilige Datenbank implementiert ist. So nutzt bspw. die Objekt-Datenbank *db4o* einen als *Activation Depth* bezeichneten Parameter, durch den festgelegt wird, bis zu welcher Tiefe der von einer Query zurückgelieferte Objekt-Teilgraph

geladen wird, um den benötigten Arbeitsspeicher gering zu halten (vgl. Edlich *et al.* 2006, S. 253). In Bezug auf Anwendungsfälle generalisieren Ramakrishnan & Gehrke (2000, S. 769) dahingehend, dass sich OODBMS dann eignen, wenn nur wenige Objekte mit komplexer interner Struktur gespeichert werden sollen, während ORDBMS dann vorzuziehen sind, wenn große Mengen vergleichweise einfach strukturierter Objekte verwaltet werden müssen.

Die hier diskutierten Vor- und Nachteile unterschiedlicher Persistenz-Frameworks[121] führen zu dem Schluss, dass im Kontext eines korpuslinguistischen Komponentenframeworks, welches den in Abschnitt 2.3 aufgeführten Ansprüchen genügen soll, kein Persistenz-Mechanismus universell geeignet ist. Vielmehr muss Entwicklern die Möglichkeit gegeben werden, abhängig von den Daten, die eine Komponente generiert, und den Query-Methoden, die durch einen AccessAdapter zur Verfügung gestellt werden sollen, zu entscheiden, welcher Ansatz am besten geeignet ist, und diesen entsprechend zu verwenden. Der folgende Abschnitt wird dies in Zusammenhang mit der Implementation von Rollen veranschaulichen.

4.2.1 Implementation von Rollen

Die mit dem *Tesla Role System* verbundenen Konzepte wurden zwar bereits in Abschnitt 4.1.4 vorgestellt, dort wurde jedoch nicht weiter auf die Implementation von Rollen und die in Tesla verwendete Umsetzung des Annotationsgraph-Konzepts eingegangen. Dies soll, da es sich um einen für die Entwicklung neuer Komponenten zentralen Aspekt handelt, in diesem Abschnitt nachgeholt werden.

Eine Konsequenz aus dem TRS und den zu Beginn von Abschnitt 4.2 zusammengefassten Vor- und Nachteilen unterschiedlicher Persistenz-Frameworks ist, dass die Verantwortung für die fehlerfreie und vollständige Implementation der von einer Rolle geforderten Speicher- und Querymechanismen (im Gegensatz zu GATE oder UIMA) beim Entwickler einer neuen Komponente liegt. Dieser muss zudem entscheiden, welcher Datenbanktyp zur Speicherung der generierten Daten geeignet ist – je nach Art der Datenstrukturen und des Zugriffs auf diese können Objekt- oder Graphdatenbanken relationalen Datenbanken gegenüber eine deutlich bessere Performanz bieten (vgl. die folgenden Abschnitte 4.2.1.1 bis 4.2.1.3), die gewählte Datenbank muss jedoch zunächst in die Architek-

[121] Hier wurden lediglich zwei unterschiedliche Konzepte vorgestellt; tatsächlich existieren jedoch bspw. mit Graph- oder Dokument-Datenbanken weitere Ansätze, die im Vergleich mit objekt-relationalen und objekt-orientierten Datenbanken eigene Vor- und Nachteile aufweisen und sich u.U. besser für die Umsetzung eines Anwendungsfalls eignen.

tur von Tesla eingebunden werden. Gegenüber Frameworks, in die ein Persistenzmechanismus integriert ist, bietet dies zwar deutlich bessere Möglichkeiten zur Modellierung von Datenstrukturen und Zugriffsmethoden (da sämtliche internen Details der verwendeten Klassen berücksichtigt werden können), führt allerdings auch zu einem erheblichen Mehraufwand bei der Konzeption und Umsetzung einer Komponente, sofern nicht auf bereits existierende Implementationen von Rollen zurückgegriffen werden kann.

Im Rahmen der Entwicklung von Tesla wurde daher untersucht, wie die Anforderungen verschiedener Rollen bestmöglich umgesetzt werden können, und wie sich gleichzeitig die Wiederverwertbarkeit der verwendeten Persistenz- und Query-Mechanismen erhöhen lässt. Dazu wurde zunächst eine (einfache, aber universell einsetzbare) Schnittstelle zur programmatischen Konfiguration von Datenbanken integriert, durch die `IAccessAdapter` und `IOutputAdapter` individuell parametrisiert werden können: Anhand des voll qualifizierten Namens einer Klasse wird bei ihrer (von Tesla durchgeführten) Instantiierung eine Parameter-Datei ausgelesen und der Instanz zugänglich gemacht, so dass für eine Datenbankverbindung benötigte Informationen (wie URL der Datenbank, Benutzername oder Passwort) ebenso konfigurierbar sind wie Parameter zur Optimierung von Persistenz- oder Query-Operationen. Zudem wurden für jedes der Persistenz-Frameworks *Hibernate* und *db4o* sowie für die Eigenentwicklung *TunguskaDB* Basisklassen für Ein- und Ausgabeoperationen implementiert, welche die Implementation neuer Adapter vereinfachen: So stellen diese Klassen Methoden bereit, mit denen sowohl auf die allgemeine Funktionalität als auch auf individuelle Fähigkeiten der jeweiligen Datenbank bzw. des jeweiligen Frameworks im Kontext der Verarbeitung von Annotation- und DataObject-Instanzen zugegriffen werden kann. Da auf diese Weise bei der Speicherung von Annotationen i.d.R. auf die Implementation eines neuen `IOutputAdapters` verzichtet und stattdessen eine Default-Implementation verwendet werden kann[122], muss lediglich bei der Implementation eines neuen `IAccessAdapters` zwischen Anforderungen der Rolle und Methoden der datenbankspezifischen Basisklasse vermittelt werden. Dies wird, anhand der Umsetzung für die oben erwähnten Datenbanken und Frameworks, in den folgenden Abschnitten beschrieben, wobei gleichzeitig auch die Eignung des jeweiligen Persistensmechanismus für unterschiedliche Rollen diskutiert wird.

[122] Die Erweiterung einer Default-Implementation ist lediglich dann notwendig, wenn die zu speichernden Daten vor der Serialisierung konvertiert werden müssen, oder wenn zusätzliche Metadaten gespeichert werden müssen. Bei den mehr als 35 bisher verfügbaren Komponenten ist dies nur bei einer Komponente der Fall – dort wurde testweise eine Optimierung zur Verbesserung der IO-Performance implementiert.

4.2.1.1 Hibernate

Mit der Einführung von *Entity Beans* in der *Java Enterprise Edition* (J2EE) im Jahr 1999 stand Java-Entwicklern zwar ein standardisierter Mechanismus für die Umsetzung von objekt-relationalem Mapping zur Verfügung, dieser vereinfachte den Einsatz von Datenbanken jedoch nur bedingt. So musste bspw. die Abbildung von Klassen auf Tabellen mit Hilfe extern definierter Metadaten durchgeführt werden, Zugriffsmethoden für Objektvariablen mussten einer fest vorgegebenen Konvention entsprechen, und obwohl Entity Beans nur innerhalb der JEE verwendet werden konnten, war die Einbindung in eine Anwendung mit hohem Implementationsaufwand verbunden (vgl. Minter & Linwood 2005, S. 7ff). Um diese Nachteile zu umgehen, wurden alternative Ansätze entwickelt, die eine Nutzung relationaler Datenbanken in objektorientierten Programmiersprachen vereinfachen – zu diesen Ansätzen gehört bspw. das ORM-Framework *Hibernate*, das auch in Tesla verwendet wird.

Maßgeblich für diese Entscheidung war u.a.[123] eine deutliche Vereinfachung des OR-Mappings im Vergleich zu Entity Beans, da Hibernate durch Verwendung von Java Annotationen eine Möglichkeit bietet, die für ORM relevanten Metadaten innerhalb des Quellcodes einer Klasse abzulegen. Zudem wurde im Rahmen des Hibernate-Projekts die Datenbank-Abfragesprache *Hibernate Query Language* (HQL) entwickelt, welche eine Abstraktion von datenbankspezifischen SQL-Dialekten ermöglicht.

Im Kontext des TRS bietet sich die Verwendung von Hibernate (analog zur Charakterisierung relationaler Datenbanken in der Einleitung zu diesem Abschnitt) prinzipiell dann an, wenn große Mengen einfacher Datenstrukturen verwaltet werden müssen – dies liegt darin begründet, dass Referenzen zwischen Objekten als Assoziationen zwischen Tabellen gespeichert werden, die bei Abfrage- oder Deserialisierungsoperationen berücksichtigt werden müssen, was die Laufzeit der Operationen negativ beeinflusst. Allerdings ist Hibernate auch bei einfachen Datenstrukturen nicht notwendigerweise bestgeeignet, was darauf zurückzuführen ist, dass es für den Einsatz in Business-Anwendungen entwickelt wurde, in denen Transaktionen zwingend erforderlich sind und auch nebenläufige Lese- und Schreiboperationen berücksichtigt werden müssen. Beides ist in Zusammenhang mit dem TRS nicht der Fall, so dass ein Großteil der von Hibernate gebotenen Funktionalität hier keine

[123] Ein weiterer, wesentlicher Punkt bestand darin, dass Hibernate nicht auf den Einsatz in JEE-Umgebungen beschränkt ist und somit auf einfache Weise in die Architektur von Tesla integriert werden konnte. Hibernate wird daher auch für die Persistierung der intern von Tesla verwendeten Datenstrukturen (bspw. von Experimenten) genutzt. Viele der in Hibernate umgesetzen Vereinfachungen bei der Weiterentwicklung der J2EE übernommen; diese standen jedoch zu Beginn der Entwicklungsphase von Tesla noch nicht zur Verfügung.

Anwendung findet, sich jedoch negativ auf das Laufzeitverhalten von IO-Operationen auswirkt – beispielsweise dann, wenn eine Tagger- oder POS-Komponente mehrere Millionen Objekte pro Sekunde generieren kann. Business-relevante Funktionen, wie eine ausfallsichere und robuste Transaktionsverwaltung, sind in diesem Kontext nicht von Bedeutung sondern führen vielmehr zu einer Verschlechterung der Laufzeit, da nur ein Bruchteil der pro Sekunde erzeugten Daten persistiert werden kann.

Vorteilhaft an der Verwendung Hibernate-basierter Adapter ist hingegen, dass mit HQL eine mächtige und datenbankübergreifend einsetzbare Abfragesprache verfügbar ist, die es bspw. erlaubt, zurückzuliefernde Daten innerhalb der Datenbank anhand von Attributen und Assoziationen filtern und/oder sortieren zu lassen – ist dies für die Umsetzung einer Rolle (bzw. insbesondere für den dort definierten AccessAdapter) relevant, so kann Hibernate als Persistenz-Framework verwendet werden.

In einem solchen Fall kann die Basisklasse `DefaultHibernateAccessAdapter` erweitert werden, um entweder anhand der dort implementierten generischen Methoden die benötigte Funktionalität umzusetzen, oder um mit Hilfe der zur Verfügung gestellten Hibernate-Session[124] eigene Queries zu erzeugen und auszuführen. Während vorgefertigte Methoden wie `InputIterator<Annotation<L>> getAnnotations(Class<L c, Range r, Order o)` Queries auf Basis von Eigenschaften der gespeicherten Annotation-Objekte ermöglichen, ohne dabei Kenntnisse in HQL vorauszusetzen[125], kann das Session-Objekt genutzt werden, um die Funktionalität einer relationalen Datenbank besser auszunutzen: So kann eine Optimierung hinsichtlich der zu speichernden Daten und anschließender Abfragen erreicht werden.

4.2.1.2 db4o

Die Objekt-Datenbank db4O (*Database for Objects*) wurde als optionaler Bestandteil in Tesla integriert, um, entsprechend der Analyse der Vor- und Nachteile von ORDBMS und OODBMS in 4.2, eine Datenbank für komplexe Objektgraphen nutzen zu können. Da db4o unter der *Gnu General Public Licence* lizensiert ist, während Tesla die *Eclipse*

[124] Diese Klasse kapselt die wesentlichen *CRUD*-Funktionen (*Create*, *Read*, *Update* und *Delete*), die für die Kommunikation mit einer relationalen Datenbank notwendig sind. Für eine ausführlichere Beschreibung sei auf Minter & Linwood (2005, Kapitel 9) verwiesen.

[125] Die Parameter der Methode bilden die Semantik des Annotationsgraphen in Tesla ab: Queries können durch ein Range-Objekt auf einen Signalausschnitt reduziert und durch ein Order-Objekt nach linkem oder rechten Anker sortiert werden, während der Class-Parameter als einfacher Filter genutzt werden kann. So kann diese Methode bspw. im Kontext eines Tokenizers verwendet werden, um eine Methode, die sämtliche Wörter eines Satzes zurückgibt (nicht jedoch ebenfalls enthaltene Satzzeichen), umzusetzen.

Public Licence verwendet, ist der Einsatz der Datenbank problematisch, denn Komponenten, die sie zur Speicherung von Annotationen verwenden, müssen ebenfalls unter der GPL lizensiert werden. In Tesla wurde die Verwendung von db4o daher nur für die Komponenten in Betracht gezogen, die aufgrund weiterer verwendeter Bibliotheken bereits unter der GPL standen, wie etwa der *Berkeley Parser* (Anhang B.4.1).

Wie bereits zu Beginn von Abschnitt 4.2 erwähnt, liegt einer der Vorteile einer Objekt-Datenbank darin, dass eine einfachere Integration in eine Programmiersprache möglich ist. In db4o äußert sich dies u.a. darin, dass keine eigene Query-Sprache verwendet werden muss, sondern dass Queries durch Beispielobjekte ermöglicht werden. Eine weitere Abfragemöglichkeit besteht der Definition von Constraints, die zurückzuliefernde Objekte beschreiben. Dies ist bspw. dann notwendig, wenn eine Suche nach Annotationen, die innerhalb eines Intervalls liegen, ausgeführt werden soll (vgl. Listing 4.8).

```
1 private void constraintRange(Query query, Range range) {
2    if(range == null) return;
3    String signalId = range.getSignalId();
4    if(signalId != null) {
5       query.descend("signalId").constrain(signalId);
6    }
7    final int left = range.getLeft();
8    final int right = range.getRight();
9    if(left >= 0 && right >= left) {
10      Constraint leftConstrain = query.descend("leftAnchor").constrain(left);
11      leftConstrain = leftConstrain.equal();
12      if(range.isWithin()) {
13         leftConstrain = leftConstrain.greater();
14      }
15      Constraint rightConstrain = query.descend("rightAnchor").constrain(right);
16      rightConstrain = rightConstrain.equal();
17      if(range.isWithin()) {
18         rightConstrain = rightConstrain.smaller();
19      }
20   }
21 }
```

Listing 4.8: Db4o-Query mit Constraints. In diesem Beispiel wird das Intervall, in dem zurückzugebende Annotationen liegen müssen, spezifiziert.

Grundsätzlich ist es in db4o nicht notwendig, zu speichernde Klassen mit zusätzlichen Metadaten auszuzeichnen (wie bei objekt-relationalem Mapping der Fall) – allerdings ist es für ein effizienteres Laufzeitverhalten von Abfragen nützlich, eine Indexierung relevanter Felder zu aktivieren, was die Vorteile der Datenbank gegenüber Hibernate relativiert, da auch hier datenbankspezifische Auszeichnungen vorgenommen werden müssen. Zudem zeigte sich bei dem Versuch, die vom *Stanford Parser* generierten Daten in db4o zu speichern, dass u.U. weitere Optimierungen notwendig sind: So ist die im *Stanford Parser* ver-

wendete Klasse `edu.stanford.nlp.trees.Tree` von `java.util.AbstractCollection` abgeleitet, unterstützt jedoch nicht sämtliche Methoden, die durch diese Klasse spezifiziert werden – die Deserialisierung derartiger Objekte führt daher zu Laufzeitfehlern in db4o.

Schließlich müssen weitere Parameter individuell, je nach zu speichernden Objekten, konfiguriert werden: So muss etwa für Abfragen definiert werden, bis zu welcher Tiefe der Objektgraph aktiviert wird. Ein zu niedriger Wert führt hierbei dazu, dass die zurückgelieferten Objekte nicht sämtliche notwendigen Informationen enthalten, während sich ein zu hoher Wert in schlechtem Laufzeitverhalten und Speicherbedarf äußert. Da die tatsächlich benötigte Aktivierungstiefe davon abhängt, wie die zurückgelieferten Daten weiterverarbeitet werden sollen, diese Information zum Zeitpunkt der Implementation eines AccessAdapters nicht zwangsläufig vorliegt[126], und da die Datenbank nur mit GPL-kompatiblem Code kombiniert werden kann, wird db4o nur von wenigen Komponenten in Tesla genutzt.

4.2.1.3 TunguskaDB

Die Datenbank *TunguskaDB* wurde im Rahmen der Entwicklung von Tesla entworfen und implementiert[127]. Ziel war es, eine Persistenzarchitektur zu entwickeln, die performante Methoden zur Speicherung flacher Datenstrukturen und zum Zugriff auf diese bietet, und die zudem sowohl einfach zu verwenden als auch für den Einsatz in Tesla optimiert ist. Dazu wurden einige der in Ghemawat *et al.* (2003) festgehaltenen Beobachtungen, die zur Architektur des *Google File System* (GFS) führten, auf Tesla übertragen. So stellten die Autoren bei der Analyse des Ein- und Ausgabeverhaltens der von Google genutzten Anwendungen bspw. fest, dass

> [...] most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once

[126] Etwa dann, wenn eine Baumdatenstruktur mit unbekannter maximaler Tiefe verwendet werden soll, wie die in Abschnitt 5.3.1 beschriebenen NGramm-Bäume.

[127] Das *Tunguska-Ereignis* bezeichnet eine schwere Explosion in der sibirischen Tunguska-Region zu Beginn des letzten Jahrhunderts, bei der ein über 2000 Quadratkilometer großes Gebiet verwüstet wurde. Über die bis heute ungeklärte Ursache der Explosion wurde viel spekuliert; eine der Spekulationen geht davon aus, dass Teslas Experimente zur Hochfrequenz-Energieübertragung die Explosion verursacht haben. Belege für diese Theorie gibt es jedoch nicht, als wahrscheinlicher gilt vielmehr der Einschlag eines Asteroiden oder vulkanische Aktivität (`http://de.wikipedia.org/wiki/Tunguska-Ereignis`). Da die hier vorgestellte Datenbank zwar für das Komponentensystem Tesla entwickelt wurde, grundsätzlich jedoch auch in anderen Kontexten einsetzbar ist, wurde das vermutlich nur indirekt mit Nikola Tesla verbundene Tunguska-Ereignis als Namensgeber gewählt.

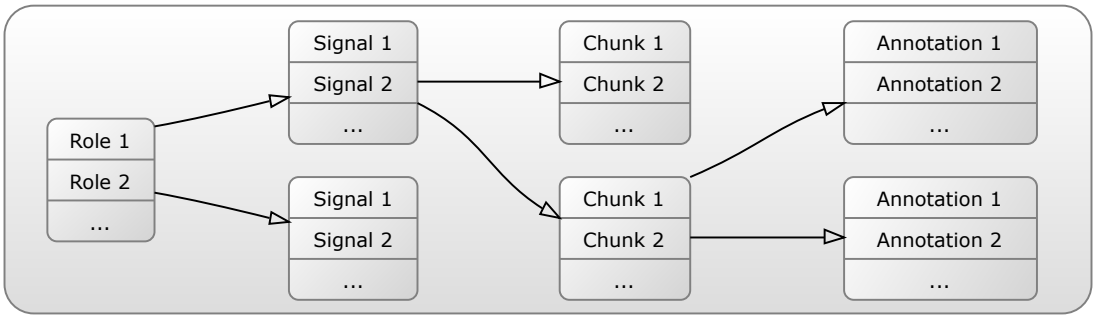

Abbildung 4.18: Hierarchische Speicherverwaltung in TunguskaDB. Die Pfade zu den gespeicherten Daten entsprechen der Struktur des von Tesla verwendeten Annotationsgraphen.

> written, the files are only read, and often only sequentially. (Ghemawat *et al.*, 2003, S. 29)

In Tesla gilt diese Beobachtung bspw. für Rollen wie Tokenizer oder POS-Tagger, deren Daten häufig sequentiell weiterverarbeitet werden – insbesondere aber wird durch das Rollensystem (bzw. die in Abschnitt 4.1 diskutierten Anforderungen an eine virtuelle Laborumgebung) festgelegt, dass gespeicherte Daten nie modifiziert, sondern lediglich gelesen oder (vollständig) gelöscht werden. Dies ermöglicht eine einfache und robuste Systemarchitektur, da bspw. keine Synchronisation paralleler Schreib- und Lesezugriffe und keine Reindexierung modifizierter Datensätze zur Optimierung von Suchanfragen berücksichtigt werden müssen: Aus der Menge der *CRUD*-Operationen (vgl. Fußnote auf Seite 140) werden hier lediglich *Create* und *Read* (sowie eine *Drop*-Operation zur vollständigen Löschung aller Daten einer Komponente) benötigt.

Ebenfalls analog zum GFS werden Daten in der TunguskaDB in *Chunks* aufgeteilt, zu denen in einer separaten, wenig Speicherplatz benötigenden Datenstruktur Metadaten verwaltet werden – in den Details unterscheiden sich beide Systeme jedoch ob der stark unterschiedlichen Anwendungsfälle deutlich. Chunks in TunguskaDB enthalten eine fest definierte Anzahl von `Annotation`- und `DataObject`-Instanzen, die nach linkem Anker aufsteigend sortiert sind, und werden als (durchnummerierte) Dateien in einem Verzeichnis, welches das annotierte Signal repräsentiert, gespeichert (vgl. Abbildung 4.18). Im einfachsten Fall, in dem eine Komponente die Annotationen zu einem Signal in linearer Folge anfordert (bspw. alle Wörter eines Textes), sind somit keine zusätzlichen Verwaltungsoperationen nötig, um die Daten zurückzuliefern.

Zu jedem gespeicherten Chunk wird der kleinste linke und der größte rechte Anker der enthaltenen Annotationen in den Metadaten hinterlegt. Diese Information lässt sich

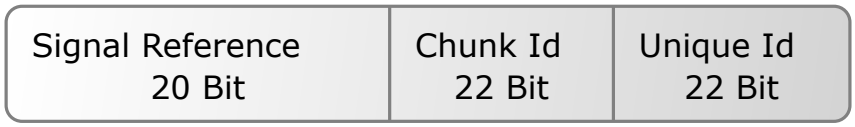

Abbildung 4.19: Aufbau einer von TunguskaDB erzeugten Id. Die insgesamt 64 zur Verfügung stehenden Bits kodieren eine Referenz auf das verwendete Signal sowie die Id des Chunks und die Speicherposition.

nutzen, wenn auf einen Ausschnitt aus einem Signal zugegriffen wird, wie etwa alle Wörter eines Satzes: Erster und letzter Chunk lassen sich anhand der Metadaten rekonstruieren, und da die Daten innerhalb des Chunks sortiert vorliegen, kann der zu analysierende Bereich mit binärer Suche weiter eingegrenzt werden. Geladene Chunks werden in einem Cache vorgehalten, so dass Lesezugriffe innerhalb einer zusammenhängenden Region eines Signals nur wenig Ladezeit erfordern.

Der Zugriff auf Annotationen über ihre Id ist ebenfalls effizient implementiert, da sich aus der Id einer in TunguskaDB gespeicherten Annotation sowohl Signal- als auch Chunk-Id rekonstruieren lassen (vgl. Abbildung 4.19), so dass die Laufzeit eines solchen Zugriffs im Wesentlichen von der Zeit, die für das Laden eines Chunks benötigt wird, abhängt.[128]

Die Serialisierung und Deserialisierung der Daten kann u.a. mit Hilfe der Bibliothek *Protostuff*[129] umgesetzt werden, was die Geschwindigkeit im Vergleich zur traditionellen Serialisierung über die `Serializable`-Schnittstelle deutlich verbessert; zudem wird jeder Chunk komprimiert, wodurch sich eine Platzersparnis von bis zu 40 Prozent ergibt.[130]

Zusätzlich wird zu jedem Chunk eine probabilistische Filterstruktur[131] erzeugt, durch die prognostiziert werden kann, ob eine gesuchte Type-Id in einem Chunk enthalten ist, wodurch Abfragen u.U. beschleunigt werden können. Optimierungen hinsichtlich der verwendeten Datenstrukturen und der benötigten Querymethoden können ebenfalls, falls notwendig, implementiert werden, indem in den (De-)Serialisierungsprozess eingegriffen wird, oder indem zusätzliche Metadaten erzeugt und bei Queries berücksichtigt werden,

[128] Wahlfreie Zugriffe können allerdings dennoch sehr aufwändig sein, da der Chunk-Cache im *worst-case* nie die benötigten Daten enthält, so dass ein Chunk in diesem Fall bei jedem Zugriff vollständig deserialisiert werden muss.

[129] Siehe http://code.google.com/p/protostuff/.

[130] Die (negative) Auswirkung der Kompression auf die Laufzeit ist dabei relativ gering, da die für die Speicherung benötigte I/O-Zeit deutlich verringert wird und die verwendete Bibliothek (https://github.com/ning/compress/wiki) zudem einen sehr effizienten Kompressionsalgorithmus nutzt.

[131] Verwendet wird ein *Bloom-Filter*, der zu jeder in einem Chunk enthaltenen Type-Id ein Bitmuster errechnet und sämtliche Bitmuster aggregiert. Bei der Suche nach Type-Ids kann durch einen Abgleich des Bitmusters mit dem Bitmuster des jeweiligen Chunks bestimmt werden, ob der Chunk ein gesuchtes Element enthalten kann oder nicht. Je nach Anzahl und Häufigkeit der Vorkommen der von einer Komponente generierten Type-Ids kann dies die Menge der zu untersuchenden Chunks einschränken.

so dass bspw. die Anzahl zu untersuchender Chunks reduziert werden kann.

Da beim Speichern von Annotationen keine aufwändigen Indexierungs- oder Verwaltungsoperationen ausgeführt werden müssen, ist ein im Vergleich mit herkömmlichen Datenbanken deutlich höherer Datendurchsatz festzustellen. So benötigte bspw. der Tokenizer *SPre* (siehe Ahnhang B.2.2) auf einem Mac Pro[132] weniger als 5 Minuten, um das vollständige *British National Corpus* mit mehr als 100 Millionen Wörtern zu annotieren und zu speichern, was ca 330.000 Schreiboperationen pro Sekunde entspricht; der weniger präzise, dafür jedoch performantere *Simple Tokenizer* (siehe Anhang B.2.1) erreichte mehr als 400.000 Operationen pro Sekunde. Bei einer zu Testzwecken durchgeführten Umstellung auf Persistierung mit Hibernate sank der Datendurchsatz auf weniger als 5000 Elemente pro Sekunde[133].

Während TunguskaDB einfache und sequentielle Suchanfragen durch die beschriebenen Verfahren effizient bearbeiten kann, ist dies bei komplexen Anfragen nicht der Fall – insbesondere dann, wenn die zurückzuliefernden Annotationen neu sortiert werden müssen. Zudem ist die Datenbank nur für die Speicherung einfacher Objekte ausgelegt, da Objektreferenzen u.U. nicht korrekt abgebildet werden.[134] Schließlich erfordert TunguskaDB, dass die zu speichernden Annotationen sortiert nach linkem Anker eingefügt werden, was bei Verwendung eines entsprechenden OutputAdapters berücksichtigt werden muss.

4.2.2 Annotationsgraph

Wie bereits in Zusammenhang mit der Beschreibung des TRS erwähnt, dient die Klasse `Annotation` dazu, eine Verknüpfung zwischen Signalausschnitt und `DataObject`-Implementation herzustellen (vgl. Seite 97). Da ein `Annotation`-Objekt neben der Angabe von linkem und rechtem Anker auch Informationen über die Komponente und die Rolle, durch die es erzeugt wurde, vorhält, beschreibt die Gesamtheit aller Annotation-Objekte implizit einen vielschichtigen Annotationsgraphen, in dem jede Ebene einer Rolle entspricht (vgl. Abbildung 4.20).

[132] Eingesetzt wurde das Standard-Modell 2008 mit 2 x 2.8 GHz Quad-Core Xeon Prozessoren, 6 GB RAM und herkömmlicher IDE-Festplatte.

[133] Unter Verwendung der Datenbank PostgreSQL 8.3 – weder die Datenbank noch Hibernate wurden hier optimiert, so dass davon auszugehen ist, dass bessere Ergebnisse erzielt werden könnten.

[134] Der zugrundeliegende Mechanismus muss während der Serialisierung über Referenzen auf sämtliche serialisierten Objekte verfügen, da verhindert werden muss, dass zwei Referenzen auf ein Objekt als zwei voneinander unabhängige Kopien gespeichert werden und der Objektgraph nach Deserialisierung nicht dem ursprünglichen Graphen entspricht. TunguskaDB garantiert jedoch nicht, dass zusammengehörige Annotationen im gleichen Chunk gespeichert werden, so dass das geschilderte Problem auftreten kann.

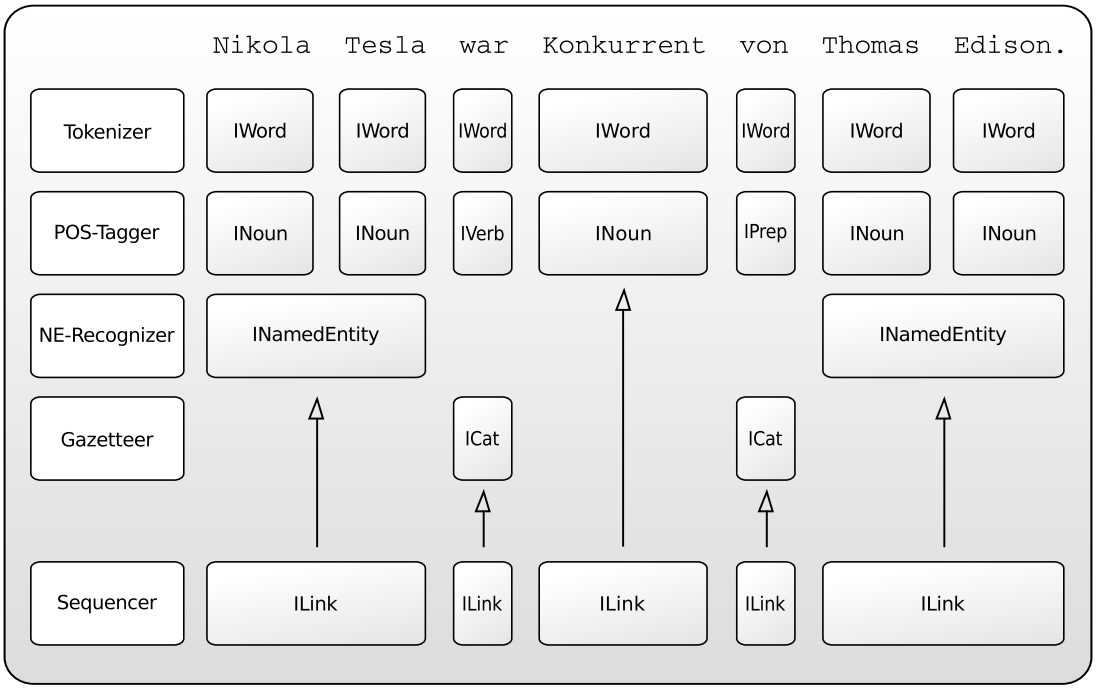

Abbildung 4.20: Schematische Darstellung des Annotationsgraphen in Tesla, in der jede Rolle einer Ebene des Graphen entspricht. Während die Annotationen der ersten drei Rollen ausschließlich am Text verankert sind, werden auf unterster Ebene zusätzlich Querverweise zu Annotationen anderer Komponenten verwendet. Das zugrundeliegende Experiment wird in Abschnitt 5.4.4 beschrieben.

Bezieht sich ein DataObject auf ein vollständiges Signal oder auf sämtliche analysierten Signale, so kann dies dadurch umgesetzt werden, dass den entsprechenden Feldern einer Annotation keine Werte zugewiesen werden.[135] Je nach Anwendungsfall ist es allerdings sinnvoll, Annotationen unmittelbar miteinander zu assoziieren, um etwa sämtliche Vorkommen eines Bigrams zu referenzieren, wie in der in Listing 4.5 auf Seite 123 gezeigten Methode. Dies führt jedoch zu einem technischen Problem: Enthält ein DataObject Referenzen auf Annotationen, die von anderen Komponenten generiert wurden, so kann nicht davon ausgegangen werden, dass diese Komponenten das gleiche Persistenzframework verwendet haben, so dass sich die Referenzen nicht unmittelbar persistieren lassen. Gleichzeitig muss verhindert werden, dass die referierten Objekte als Kopie persistiert werden, denn andernfalls würde dies – neben einem erhöhten Speicher- und Laufzeitbedarf – zu Fehlern führen, falls die Objekte nicht von dem verwendeten Persistenzframework unterstützt werden (etwa weil die von Hibernate benötigten Metadaten nicht verfügbar

[135] Weitere Einschränkungen, wie etwa die Annotation einer Document Selection, sind zur Zeit nicht implementiert.

sind).

Aus diesem Grund ist die Variable der Klasse `Annotation`, die auf ein DataObject verweist, als `transient` markiert – das referenzierte Objekt wird bei Serialisierungsoperationen nicht automatisch gespeichert oder geladen, sondern muss explizit von dem verwendeten OutputAdapter (de-)serialisiert werden. Da dies nur für auf höchster Ebene referenzierte Objekte geschieht, werden DataObject-Instanzen, auf die durch Querverweise Bezug genommen wird, ausgelassen. Mit Hilfe dieser Konstruktion konnte eine datenbankübergreifende Form des *Lazy-Loading* realisiert werden: Versucht eine Komponente, über die Methode `Annotation.getDataObject()` auf ein DataObject zuzugreifen, so überprüft die Methode zunächst, ob das Objekt bereits geladen wurde. Ist dies nicht der Fall, kann anhand der übrigen, im Annotationsobjekt verfügbaren Informationen ermittelt werden, welcher AccessAdapter verwendet werden muss, um das Objekt (dessen Id identisch mit der Id der Annotation und somit bekannt ist) zu laden. Da das Interface `IAccessAdapter` die Methode `getAnnotationById(long id)` definiert, kann dieses Vorgehen auf sämtliche AccessAdapter und somit auch auf sämtliche Persistenz-Frameworks angewendet werden.[136] Um von konkreten Daten zu abstrahieren wurde ein weiteres Feld in den Basisdatentyp Annotation eingefügt, das typbezogenen Komponenten ermöglicht, mit dem Output beliebiger tokenbezogener Komponenten umzugehen. Jede Annotation, die von einer Komponente produziert wird, kann mit einer Type-Id ausgezeichnet werden. Ist diese Id bei zwei Annotationen identisch, so beschreiben diese den gleichen Type. Dies ermöglicht es, die Kategoriezugehörigkeit von Annotationen auf generische Weise auszuwerten, da eine Komponente auf diese Weise erzeugte Annotationen mit einem individuellen Typisierungskonzept verknüpfen kann: Der Tokenizer *SPre* weist Type-Ids bspw. auf Basis des annotierten Strings zu, während die Type-Id der vom BNC-Reader generierten POS-Tags einer Kodierung der zugewiesen Part-Of-Speech-Kategorie entspricht. Komponenten, die Type-Ids von Annotationen weiterverarbeiten, übernehmen so automatisch das Typisierungskonzept der konsumierten Komponenten.

[136] Es bleibt allerdings noch zu untersuchen, ob diese Lösung für komplexe praktische Anwendungsfälle hinreichend ist, oder ob sie u.U. zu inperformant ist, denn das Query-Potential des verwendeten Persistenzframeworks wird bei iterativen Zugriffen über die Id eines Objekts, die der verwendete Lazy-Loading-Mechanismus ausführt, nicht ausgeschöpft, und die Datenbank kann lediglich durch Caching und/oder evtl. implementierte probabilistische Modelle den Zugriff auf die Daten beschleunigen. Sequentielle Zugriffe über die Id eines Objekts sind bspw. in TunguskaDB performant, da der betroffene Chunk in einem solchen Fall i.d.R. bereits geladen wurde, während verteilte Zugriffe zu permanenten Nachladen der Chunks führen und deutlich langsamer sind.

4.2.3 Funktionstests

Nachdem in den vorherigen Abschnitten erläutert wurde, wie die Implementation von Rollen mit Hilfe von Basis-AccessAdaptern vereinfacht werden kann, stellt sich die Frage, wie eine solche Implementation getestet werden kann, um auszuschließen, dass der Quellcode Fehler enthält, die u.U. nur in Kombination mit einigen Komponenten auftauchen[137]. Daher wurde eine Schnittstelle entworfen, mit der Tests für Rollen entwickelt werden können – hierfür konnte das Test-Framework *JUnit*[138] verwendet und in das Konzept des Rollensystems integriert werden.

```
1 public interface IMultiValueCategoryAccessAdapter<T extends IMultiValueCategory> extends
      ICategoryAccessAdapter<T> {
2    ...
3    class AdapterTest implements IAccessAdapterTest<IAccessAdapter<DataObject>> {
4
5          private IAccessAdapter<DataObject> adapter;
6
7          public void initialize(IAccessAdapter<DataObject> adapter) {
8             this.adapter = adapter;
9          }
10
11         @SuppressWarnings({ "unused"})
12         @Test
13         private void testgetAllCategories() {
14            Set<String> allCategories = adapter.getAllCategories(null);
15            assertNotNull("getAllCategories returned null!", allCategories);
16            assertTrue("No categories found!", allCategories.size() > 0);
17         }
18         ...
19   }
20 }
```

Listing 4.9: Auszug eines Tests einer Rolle in Tesla. Die innerhalb des Interfaces definierte Klasse `AdapterTest` kann mit einer Implementation des Interfaces initialisiert werden, um anschließend mit Hilfe von *JUnit* ihre Funktionalität zu testen.

Um Tests für eine Rolle zu entwickeln, kann, wie in Listing 4.9 gezeigt, ein IAccessAdapter-Subinterface um eine interne Klasse erweitert werden, in der die Tests definiert werden. Um sie ausführen zu lassen, kann anschließend die Tesla-Komponente *JUnit Tests* (vgl. Anhang B.6.1) zu einem Experiment hinzugefügt und mit der zu testenden Rolle verknüpft werden. Zur Laufzeit dieser Komponente werden sämtliche Tests, die in der Interfacehierarchie der zu testenden Rolle definiert wurden, detektiert und – nach Initialisierung mit dem verwendeten AccessAdapter – ausgeführt.

[137] Konsumiert eine Komponente eine Rolle R, so ist nicht davon auszugehen, dass sämtliche Methoden, die AccessAdapter und DataObject von R definieren, von der Komponente verwendet werden. Dies kann dazu führen, dass Programmierfehler in *selten aufgerufenen* Methoden erst spät entdeckt werden.

[138] Siehe http://www.junit.org/.

Derartige Tests weichen konzeptuell von Unit- oder Integrationstests ab[139], da die Ergebnisse vom Experiment abhängen und auf den Versuchsaufbau bezogen interpretiert werden müssen: So kann es bspw. vorkommen, dass die *Gazetteer*-Komponente (siehe Anhang B.7.1) keine Annotationen generiert, weil die im Lexikon gespeicherten Wortsequenzen nicht im analysierten Korpus enthalten waren – in diesem Fall würde der in Listing 4.9 abgebildete Test in Zeile 16 fehlschlagen, ohne dass notwendigerweise ein Implementationsfehler vorliegt.

Die Integration nicht-deterministischer Rollentests ist somit lediglich eine Ergänzung zu Modul- und Integrationstests, durch die Funktionalität und Effizienz implementierter Rollen geprüft werden kann, die jedoch zusätzlich auch der Dokumentation einer Rolle dient: Anhand der Annahmen, die in den Tests zu einer Rolle überprüft werden, wird das erwartete Verhalten der Rolle beschrieben, was das Testen einer Reimplementation einer Rolle in einem anderen Persistenzframework ebenso wie das Testen einer neuen Komponente, die eine existierende Rolle erfüllen soll, vereinfacht.[140] Da die Tests auf JUnit basieren und die wichtigsten Funktionen des Frameworks[141] genutzt werden können, sind Syntax und Semantik der Tests Entwicklern, die bereits JUnit verwendet haben, unmittelbar geläufig.

4.2.4 Diskussion

In Abschnitt 4.2.1 wurde gezeigt, wie verschiedene Datenbanken mit individuellen Vor- und Nachteilen auf einheitliche Weise in Tesla integriert werden konnten. Während das TRS die öffentlichen Schnittstellen zwischen Komponenten definiert, ist der Persistenzmechanismus einer Komponente der internen Funktionalität zuzuordnen – die in 4.2.2 beschriebene, dynamische Auflösung von Referenzen zwischen Annotationen gewährleistet einen transparenten, Datenbank-unabhängigen Zugriff auf den durch ein Experiment

[139] vgl. bspw. die Definition von Massol & Husted (2003, S. 6):

> A unit test examines the behavior of a distinct *unit of work*. Within a Java application, the "distinct unit of work" is often (but not always) a single method. By contrast, *integration tests* and *acceptance tests* examine how various components interact. A *unit of work* is a task that is not directly dependent on the completion of any other task.

[140] In Kombination mit der Kommentierung der durch eine Rolle definierten Interfaces können die Tests als eine weitere Form des *literate Programming*) angesehen werden, da sie das erwartete Verhalten einer Rollenimplementation beschreiben.

[141] Hierzu werden die Auszeichnung einer Test-Methode mit der Annotation `@org.junit.Test`, die Verwendung der Klasse `org.junit.Assert` sowie das erwartete Auftreten eines Fehlers (durch Parametrisierung der `@Test`-Annotation) gezählt. Darüber hinausgehende Funktionen von JUnit, wie etwa Test-Suites, werden nicht unterstützt.

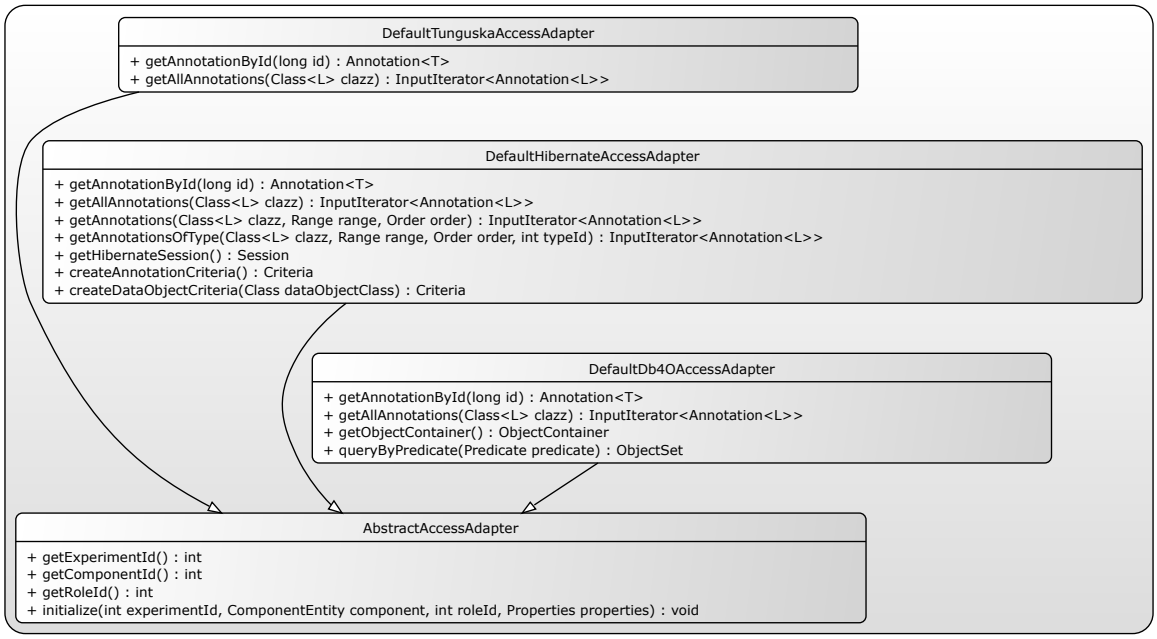

Abbildung 4.21: Vergleich der AccessAdapter-Basisklassen verschiedener Persistenzmechanismen. Neben universellen, an den von Tesla verwendeten Annotationsgraphen angepassten Query-Methoden bieten die Klassen auch Zugang zu datenbankspezifischen Queries, mit denen komplexe Abfragen effizient umgesetzt werden können.

modellierten Objektgraphen. So können Persistenz- und Querymechanismen entsprechend der Anforderungen einer Komponente gewählt werden, was die Umsetzung komplexer Anwendungsfälle vereinfacht – die in 4.2.3 vorgestellten Funktionstests stellen dabei sicher, dass unabhängig vom verwendeten Persistenzframework und unabhängig von der konkreten Implementation einer Rolle die Kompatibilität und Austauschbarkeit einer Komponente gewährleistet ist.

Aus dem in Tesla verfolgten Ansatz ergibt sich, dass Entwickler weder bei Konzeption und Implementation von Rollen, DataObjects und AccessAdaptern noch bei Umsetzung einer Idee in Form einer Komponente durch das Framework eingeschränkt werden, so dass (aus technischer Sicht) jede beliebige Anforderung umgesetzt werden kann.

Es soll nicht unerwähnt bleiben, dass das TRS auch zu einem (im Vergleich mit einfachen Modellen wie dem Annotationsgraphen in GATE) erhöhten Entwicklungsaufwand führen kann, da u.U. individuelle Eigenarten des verwendeten Persistenzframeworks berücksichtigt werden müssen (etwa die Annotation zu persistierender Klassen für Hibernate, die Festlegung der Aktivierungstiefe in db4o oder die Vorsortierung nach linkem Anker in TunguskaDB). Die bereits implementierten Basisklassen für die Verwendung der hier

vorgestellten Datenbanken vereinfachen dies zwar, indem sie Funktionen bereitstellen, die auf die Anforderungen von Tesla ausrichtet sind – grundlegende Kenntnisse des jeweiligen Persistenzframeworks sind jedoch notwendig.

4.3 Architektur des Tesla Servers

Zu Beginn der Konzeption von Tesla wurde untersucht, ob der Server auf Basis existierender Anwendungen, wie etwa des Applikationsservers *JBoss*[142] oder eines Servlet-Containers wie *Tomcat*[143] umgesetzt werden kann. Analog zur Integration des Tesla Clients in Eclipse wäre ein solches Vorgehen insofern vorteilhaft gewesen, als dass die Funktionalität, die die genannten Frameworks bieten, unmittelbar hätte genutzt werden können: So integriert JBoss bspw. Clustering-Technologien, mit denen eine installierte Anwendung auf mehrere Rechner verteilt und den Anforderungen entsprechend skaliert werden kann. Es stellte sich jedoch heraus, dass dies mit einer deutlichen Beschränkung der Flexibilität von Tesla verbunden gewesen wäre, da zentrale Aspekte eines Komponentensystems nicht hätten umgesetzt werden können, ohne Eingriffe in die Architektur der genannten Anwendungen durchzuführen.[144]

Stattdessen wurde der Tesla Server als eigenständige Anwendung implementiert, die unter einer beliebigen JavaSE-Umgebung ausgeführt werden kann und – abgesehen von einer Datenbank, die zur Persistierung von Experimentkonfigurationen benötigt wird – keine weiteren Anforderungen an die Softwareumgebung stellt. Die Server-Applikation nutzt das im folgenden Abschnitt beschriebene *Spring Framework*[145], um die Vorteile einer eigenen Anwendung (wie etwa die Kontrolle über zu ladende Klassen und die Konfiguration von Umgebungsvariablen) mit der Funktionalität existierender Bibliotheken (wie dem bereits in Abschnitt 4.2 beschriebenen *Hibernate*) zu verbinden. Abbildung 4.22 veranschaulicht den Startprozess des Servers.

Zu Beginn der Startphase des Servers werden allgemeine Konfigurationsoptionen interpretiert, die der Anwendung als Parameter übergeben wurden, wie die Konfiguration des

[142] Siehe http://www.jboss.org/.
[143] Online unter http://tomcat.apache.org/.
[144] Dies ist u.a. in den – notwendigen – Sicherheitskonzepten der genannten Anwendungen begründet, die bspw. verhindern, dass eine ausgeführte Anwendung auf die von einer anderen Anwendung definierten Java-Klassen zugreift. Es zeigte sich jedoch auch, dass bspw. die genannten Clustering-Möglichkeiten von JBoss nicht ohne weiteres für zeitintensive Operationen, wie sie bei Ausführung einer Komponente auftreten, genutzt werden können: In der Evaluationsphase wurden ausgelastete Knoten im Cluster von der zentralen JBoss-Instanz fälschlicherweise als unerreichbar interpretiert und aus dem Cluster ausgeschlossen.
[145] Siehe http://www.springsource.org/.

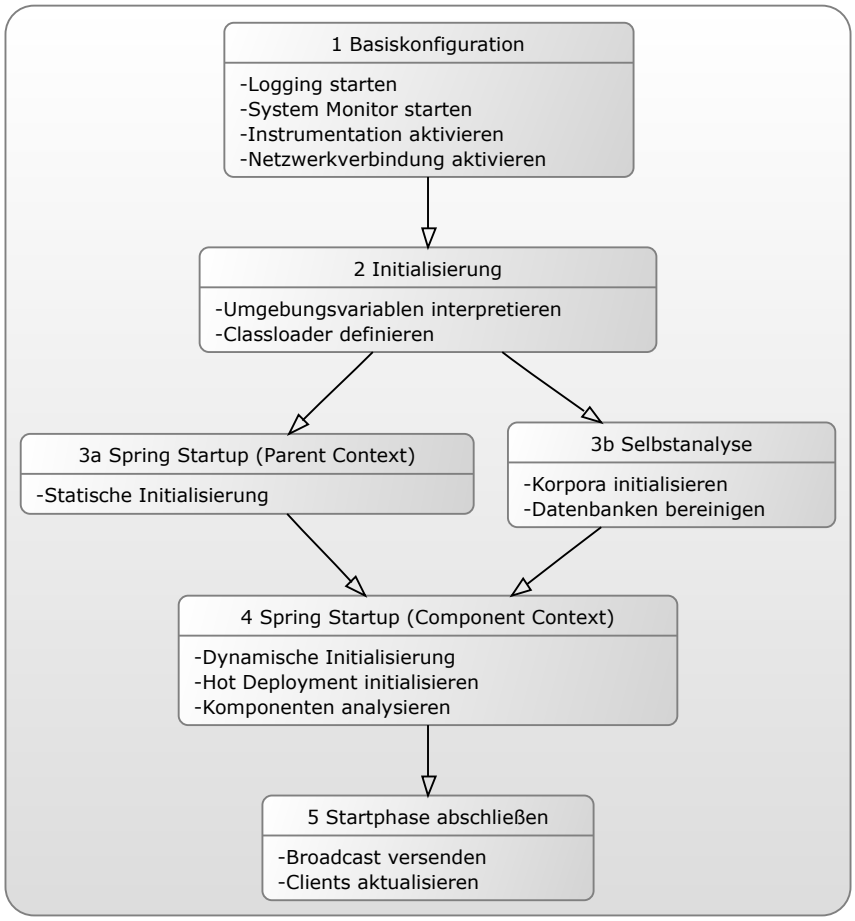

Abbildung 4.22: Startsequenz des Tesla Servers.

Logging-Frameworks und der Netzwerkadresse, über die der Server erreichbar sein soll. In der zweiten Phase wird der *Classpath* des Servers auf Basis der Konfiguration erzeugt und gesetzt; insbesondere werden die Java-Archive, in denen Komponenten definiert sind, zum Classpath hinzugefügt. Diese werden – im Gegensatz zu den weiteren Bibliotheken, die für den Betrieb des Servers benötigt werden – separat verwaltet und können bei Bedarf zur Laufzeit aktualisiert werden. In Schritt 3a und 4 wird das Spring-Framework initialisiert, durch das die wesentlichen Server-Features von Tesla umgesetzt werden.

Die Konfiguration von Spring ist in zwei separaten Kontexten definiert, um die relativ zeitaufwändige Initialisierung allgemeiner Serverdienste (bspw. Hibernate) von Tesla-spezifischen Funktionen zu trennen. Parallel zu Schritt 3a werden die vom Server verwendeten Ressourcen überprüft und bspw. Korpora (re-)indexiert, sowie evtl. vorhandene defekte Datenbanken gelöscht.[146] Nach Abschluss beider Sequenzen ist die allgemeine In-

[146]Bei Ausführung einer Komponente protokolliert Tesla deren Ausführungszustand, so dass rekonstruiert werden kann, ob die von der Komponente generierten Daten vollständig sind, oder ob die Ausführung

itialisierung des Systems beendet, und die komponenten-abhängige Initialisierung wird in Schritt 4 fortgesetzt. Dies beschleunigt nicht nur den Start des Servers, sondern ermöglicht es, bei Modifikation einer Komponente lediglich die Schritte 4 und 5 neu auszuführen, um die Änderungen in den laufenden Server zu übernehmen. Der Tesla Server ist somit nicht nur *Hot Deployment*-fähig, sondern zudem in der Lage, innerhalb von wenigen Sekunden eine Aktualisierung durchzuführen[147], da nur wenige Dateien neu geladen werden müssen.

Nachdem Spring vollständig gestartet wurde, sendet der Server eine *Broadcast*-Nachricht an Clients, die für die Verwendung mit dem Server konfiguriert wurden, um diese über die Verfügbarkeit des Servers zu informieren. Diese verbinden sich daraufhin mit dem Server und aktualisieren ggfs. die lokal gespeicherten Komponentendefinitionen – Abschnitt 4.3.2 beschreibt die Client-Server-Kommunikation im Detail, im folgenden Abschnitt wird jedoch zunächst die dafür erforderliche Infrastruktur vorgestellt.

4.3.1 Das Spring Framework

Das Spring Framework wurde als leichtgewichtige Alternative zu Frameworks wie der J2EE entwickelt. Ziel war es, den Einfluss des Frameworks auf die zu modellierende Anwendung möglichst gering zu halten, um deren Architektur nicht unnötig zu beeinflussen. Johnson (2003, S. 167), Initiator des Projekts, stellt dazu fest:

> A framework differs from a class library in that committing to a framework dictates the architecture of an application. Whereas user code that uses a class library handles control flow itself, using class library objects as helpers, frameworks take responsibility for control flow, calling user code.

Smeets & Ladd (2007) beschreiben das Problem anhand der JDBC-Bibliothek wie folgt:

> JDBC [...] influences the design of an application in such a way that the focus of the design shifts away from its original goals toward trying to use the API in the application. In fact, because the JDBC API is so intrusive, application developers should not spend their time trying to use it correctly. The same can be said for many other APIs in the Java platform.

Zur Lösung dieses Problems integriert Spring das Entwurfsmuster der *Dependency Injection*, einer teilweisen Realisierung des *Inversion of Control*-Paradigmas: Spring stellt

der Komponente aufgrund eines Fehlers abgebrochen wurde und die produzierten Daten verworfen werden müssen.

[147] Die benötigte Zeit ist abhängig von der Anzahl der Komponenten, lag aber i.d.R. deutlich unter der Zeit, die bspw. von Tomcat oder JBoss für ein Hot Deployment benötigt wird.

eine Schnittstelle zwischen externen APIs (wie JDBC) und der Anwendunglogik zur Verfügung, mit deren Hilfe Abhängigkeiten zwischen den einzelnen Modulen (im Kontext von Spring als *Spring Beans* bezeichnet) durch externe Konfiguration aufgelöst und die Abhängigkeiten vom Framework zur Laufzeit der Anwendung injiziert werden können. Spring Beans sind Java-Objekte, die vom Framework erzeugt und, wenn sie nicht mehr benötigt werden, auch wieder zerstört werden – Spring verwaltet also den Lebenszyklus der Spring Beans, sorgt aber insbesondere dafür, dass die Beans konfiguriert werden. Muss ein Spring Bean instantiiert werden, so werden zunächst die Beans, von denen dieses abhängig ist, instantiiert (was ggfs. zu weiteren Instantiierungen führt), um dem Bean anschließend Referenzen zu den angeforderten Objekten zu übergeben.

```
...
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="${jdbc.driver}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>

<bean id="sessionFactory" class="de.uni_koeln.spinfo.tesla.server.TeslaSessionFactoryBean">
  <property name="dataSource">
    <ref bean="dataSource" />
  </property>
  <property name="hibernateProperties">
   ... <!-- weitere Parameter zur Konfiguration von Hibernate -->
  </property>
</bean>
...
```

Listing 4.10: Auszug einer Konfigurationsdatei von Spring.

Listing 4.10 veranschaulicht dies anhand eines Auszugs der Spring-Konfiguration von Tesla, in dem ein Teil der für die Verwendung von Hibernate benötigten Abhängigkeiten deklariert wird. Das unter der Id `sessionFactory` definierte Spring-Bean bietet bspw. die Möglichkeit, Hibernate-Sessions (vgl. Abschnitt 4.2.1.1) zu erzeugen, die anschließend innerhalb des Frameworks genutzt werden können. Da hierfür u.a. eine Datenquelle definiert und konfiguriert werden muss, wird über die Eigenschaft `dataSource` ein weiteres Bean referenziert, welches die benötigte Funktionalität kapselt und zugleich eine Möglichkeit zur externen Konfiguration bietet.

Soll innerhalb einer Javaklasse auf ein Spring-Bean zugegriffen werden, sind – je nach Anwendungsfall – zwei Möglichkeiten denkbar. Zum einen kann ein *Lookup* im von Spring verwalteten Kontext durchgeführt werden, da mit der Id eines Beans eine eindeutige Referenzierung möglich ist. Zum anderen kann, falls das Objekt durch das Spring Fra-

mework erzeugt wurde, der Injection-Mechanismus von Spring verwendet werden. Die zweite Methode ist dabei i.d.R. zu empfehlen, da so gewährleistet wird, dass auch diese Klasse von Spring verwaltet wird und von der Framework-Funktionalität, bspw. in Bezug auf Transaktionsmanagement oder Authentifizierung, profitieren kann. Dazu ist es nicht zwingend notwendig, die Konfigurationsdatei von Spring zu erweitern – ähnlich wie Hibernate verwendet auch Spring (seit Veröffentlichung von Version 2.5 im Jahr 2006) Java Annotationen, um eine weniger fehleranfällige Alternative zur Deklaration von Spring Beans in der Konfiguration zu bieten. Dazu wird eine Klasse als *@Service* annotiert; zudem werden die Felder, denen Spring Beans zugewiesen werden sollen, mit der Annotation *@Autowired* versehen. Listing 4.11 zeigt dies anhand eines Ausschnitts der Klasse `EvaluationExporterBean`, die in Tesla für den Export von Daten (vgl. Abschnitt 4.1.6) verantwortlich ist.

```
1 @Service(EvaluationExporter.NAME)
2 @Secured( { "ROLE_GUEST", "ROLE_USER" })
3 public class EvaluationExporterBean implements EvaluationExporter {
4
5     @Autowired
6     @Qualifier(AdapterFactory.NAME)
7     private AdapterFactory adapterFactory;
8
9     public RemoteInputStream getConvertedXmlData(final XmlExportDescription
10    description) throws IOException {
11    ...
12    }
13    ...
14 }
```

Listing 4.11: Deklaration eines Spring Beans mit Java Annotationen

Das Beispiel verdeutlicht, wie durch das Konzept der Dependency Injection eine Vereinfachung der Entwicklung erreicht werden kann: Dank der Annotationen ist es hier nicht notwendig, Code zur Erzeugung, Konfiguration und Zuweisung einer `AdapterFactory`-Instanz für die Variable *adapterFactory* zu integrieren oder sonstige Methoden des Frameworks aufzurufen – wird eine Bean vom Typ `EvaluationExporterBean` erzeugt, werden derartige Schritte automatisch vom Spring Framework durchgeführt.

Für die Entscheidung, Spring in Tesla zu verwenden, war vor allem ausschlaggebend, dass es flexibel an die Anforderungen angepasst werden konnte: So wird nicht nur die Persistierung von Teslas internen Datenstrukturen (wie etwa Experimenten) über Spring realisiert, sondern auch die im nächsten Abschnitt vorgestellte Kommunikation zwischen Client und Server.

4.3.2 Client-Server Kommunikation

Zur Kommunikation zwischen Client und Server werden in Tesla zwei Techniken verwendet, die den unterschiedlichen Kommunikationsrichtungen entsprechen: So müssen nicht nur Befehle von einem Client an den Server übertragen werden, vielmehr müssen umgekehrt auch Rückmeldungen des Servers an sämtliche Clients, die mit dem Server verbunden sind, in Echtzeit zurückgegeben werden können.

Die Ausführung von Befehlen wird mit Hilfe von *Remote Method Invocation* (RMI) umgesetzt: Serverseitig werden Dienste (bspw. die Speicherung eines neuen Experiments) durch Interfaces definiert und für clientseitige Aufrufe bereitgestellt. Durch entsprechende Konfiguration der hierfür verwendeten Spring-Beans (siehe auch Listing 4.11) wird zudem eine Authentifizierungsschnittstelle eingebunden, so dass nur registrierte Anwender auf die Dienste des Servers zugreifen können.[148] Diese Kommunikationsrichtung wird synchron ausgeführt: Der Client ist während des Aufrufs einer Methode blockiert, bis diese vollständig ausgeführt wurde. So ist es zwar möglich, Rückgabewerte zu verwenden (und bspw. die Liste aller serverseitig verfügbaren Komponenten an den Client zu übergeben); ein derartiges Vorgehen ist jedoch für häufig eintretende Ereignisse, wie etwa die Aktualisierung von Zustandsmeldungen, nicht umsetzbar (zumal mehr als ein Client mit einem Server verbunden sein kann). Daher verwendet der Server als zweiten Kommunikationskanal ein asynchron und unidirektional arbeitendes Benachrichtigungssystem, durch das es möglich ist, an registrierte Clients Nachrichten zu senden. Je nach Relevanz wird dabei das TCP- oder UDP-Protokoll verwendet: Zustandsmeldungen, etwa über Prozessorauslastung oder verfügbaren Arbeitsspeicher, werden (falls nicht anders konfiguriert) in regelmäßigen Abständen über UDP verschickt – daher können Nachrichten verloren gehen, was jedoch nicht relevant für die Funktionalität von Client und Server ist. Wichtige Nachrichten, bspw. bezüglich der Modifikation einer Document Selection, werden hingegen über TCP an jeden verbundenen Client gesendet, um sicherzustellen, dass der im Client abgebildete Zustand des Servers dem aktuellen Zustand entspricht.

Beide Richtungen der Kommunikation sind dabei völlig unabhängig vom clientseitig verwendeten Eclipse-Framework – dies ermöglicht es, den Tesla-Server auch in andere

[148] Die Authentifizierung wird mit Hilfe des *Java Authentication and Authorisation Service* (JAAS, siehe auch http://www.oracle.com/technetwork/java/javase/jaas/index.html) durchgeführt, wodurch sie an die Anforderungen der jeweiligen Nutzerumgebung angepasst werden kann – bspw. ist eine Integration in eine bereits bestehende LDAP-Infrastruktur möglich. JAAS bietet die Möglichkeit, unterschiedliche Benutzergruppen zu definieren, von der Tesla zur Zeit jedoch nur eingeschränkt Gebrauch macht: Hier wird zwischen Gästen, die lediglich Lesezugriff auf Experimente und Korpora haben, und Anwendern mit Vollzugriff auf das System unterschieden.

Kontexte einzubetten und bspw. ausgewählte Dienste über Web-Services zur Verfügung zu stellen. Zudem konnte die Funktionalität des Servers so unabhängig vom Client getestet werden.[149]

4.4 Fazit: Tesla als Framework für strukturalistisch motivierte Verfahren

In diesem Kapitel wurde ein Framework-Entwurf vorgestellt, der sich von den in Kapitel 3 beschriebenen Systemen u.a. dadurch unterscheidet, dass Entwickler bei der Konzeption und Implementation neuer Komponenten bzgl. der zu verwendenden Datenstrukturen nicht eingeschränkt werden, sondern den vollständigen Funktionsumfang einer Programmiersprache verwenden können. Der im Zusammenhang mit UIMAs *Common Analysis Structure* geäußerte Einwand, dass eine derartige Flexibilität zwingenderweise mit stark erhöhtem Entwicklungsaufwand einhergehen würde (vgl. Seite 81), konnte zumindest teilweise widerlegt werden: Durch Bereitstellung geeigneter IO-Schnittstellen für unterschiedliche Persistenzmechanismen (vgl. Abschnitt 4.2), sowie durch Erweiterung der IDE-Funktionalität von Eclipse (Abschnitt 4.1.7.2) konnte der notwendige Mehraufwand stark reduziert werden; zudem vereinfacht die hier vorgestellte Architektur die Integration bereits existierender Werkzeuge, da häufig deutlich weniger Aufwand zur Konvertierung von Ein- und Ausgabedatenstrukturen benötigt wird (insbesondere dann, wenn keine komplexen Query-Mechanismen unterstützt werden müssen und die von einem Werkzeug verwendeten Datenstrukturen bspw. mit db4o oder TunguskaDB persistiert werden können).

Die innerhalb des *Tesla Role System* zusammengefassten und in Abschnitt 4.1.4 vorgestellten Konzepte stellen eine wohldefinierte Schnittstelle zum Daten- und Funktionsaustausch zwischen Komponenten bereit, die nicht nur die Austauschbarkeit und Wiederverwendbarkeit von Komponenten erhöht, sondern es auch ermöglicht, fein granulierte Rollen zu entwerfen und so Komponenten zu entwickeln, deren konsumierte oder produzierte Datenstrukturen mit Hilfe der in Kapitel 3 diskutierten Frameworks nicht oder nur unter großem Aufwand abgebildet werden könnten. Allerdings kann, wie bereits in Abschnitt 4.1.4.2 erwähnt, die Spezifikation einer neuen Rolle u.U. ein relativ langwieriger Prozess werden, da möglichst viele unterschiedliche Anwendungsfälle berücksichtigt

[149] Über 5000 JUnit-Tests stellen zur Zeit sicher, dass die Funktionalität des Servers und der bereits implementierten Komponenten den Anforderungen entspricht (siehe `http://tesla.spinfo.uni-koeln.de/nightlies/junit.html`).

werden sollten, und da eine strikte Trennung von Definition der Rolle und ihrer Implementation durch eine Komponente erfolgen muss. Zudem hat die Erfahrung mit dem TRS gezeigt, dass die Flexibilität des Ansatzes zumindest beim aktuellen Entwicklungsstand die Übersichtlichkeit beeinträchtigt, da sich Zusammenhänge zwischen unterschiedlichen Rollen und Komponenten nicht immer unmittelbar erschließen. Dieser Nachteil ließe sich jedoch durch Optimierung der graphischen Benutzeroberfläche von Tesla beheben.

Jedes der in dieser Arbeit besprochenen Frameworks ist je nach Verwendungszweck unterschiedlich gut einsetzbar, wie auch die Zusammenfassung in Tabelle 4.1 zeigt: In Abschnitt 3.2.5 wurde bspw. festgehalten, dass UIMA für den Einsatz innerhalb etablierter Produktionsabläufe am besten geeignet ist, während, wie zu Beginn von Abschnitt 3.3 erläutert, TextGrid in erster Linie eine interdisziplinäre Austauschplattform für geisteswissenschaftliche Forschungsprojekte (insbesondere für die dort anfallenden Daten) bereitstellt. Ausgangspunkt der Untersuchung war jedoch die Fragestellung, mit welchem Framework die in Abschnitt 2.3 formulierten Anforderungen bestmöglich umgesetzt werden können, d.h. wie experimentelles Arbeiten mit neuen Verfahren und Anwendungsfällen in einer virtuellen Forschungsumgebung umgesetzt und evaluiert werden kann, und wie vergleichsweise komplexe Datenstrukturen und Methoden (wie in Abschnitt 2.2.2 beschrieben) in ein solches Framework eingebunden werden können. Die vorangegangenen Kapitel haben gezeigt, dass die in Tesla integrierten Konzepte eine Umsetzung dieser Anforderungen ermöglichen, da u.a. das in Abschnitt 4.1.4.1 dargestellte Vererbungskonzept von Rollen die Möglichkeit bietet, Kompatibilität und Wiederverwertbarkeit von Komponenten zu maximieren.

Um dies auch praktisch zu illustrieren, wird im folgenden Kapitel abschließend demonstriert, wie ein solches experimentelles Arbeiten am Beispiel von durch strukturalistische Hypothesen motivierten Verfahren, wie in Kapitel 2 beschrieben, in Tesla umgesetzt werden kann.

	GATE	UIMA	TextGrid	Tesla
Lizenz	LGPL	Apache Licence	LGPL	EPL
Programmiersprache	Java	Java, C++, Prolog (weitere implementierbar)	SOAP-Schnittstelle	Java (Scala, Jython)
Annotationsmodell	Key-Value-Paare: Komplexe Datentypen zur Laufzeit, primitive Datentypen serialisierbar	Komplexe Datentypen durch CAS-Komposition	TEI/kein konkretes Austauschformat definiert	Java-Interfaces
Persistenz-Modell	Unterstützung verschiedener SQL-Datenbanken (Oracle, PostGres) sowie Lucene	Beliebig implementierbare Schnittstelle (CAS-Consumer); Unterstützung von Lucene und Apache Solr vorhanden	Unterstützung verschiedener Datenbanken (u.a. Exist)	beliebige Datenbanken integrierbar (durch Adapter); Unterstützung von Hibernate, db4o und TunguskaDB implementiert
Architektur	Desktop-Anwendung; kostenpflichtige Cloud-Services werden angeboten	Server	Service-orientierter Client	Client-Server
IDE	keine	Unterstützung bei der CAS-Generierung	keine	Integration in die Eclipse-IDE
Benutzeroberfläche	Swing-Client	Swing- und SWT-Fragmente	Eclipse RCP Anwendung	Eclipse Plugins
Finanzierung und Entwicklung	Entwickelt im Rahmen einer Dissertation, anschließend diverse Drittmittel	zunächst IBM, dann Betreuung durch Apache Software Foundation als Open-Source-Projekt	BMBF im Rahmen der Grid-Initiative	Entwickelt im Rahmen dieser Dissertation und Hermes (2011).

Tabelle 4.1: Gegenüberstellung der Evaluation von GATE, UIMA, TextGrid und Tesla.

5 Strukturalistische Analyse mit Tesla

> The scientific man does not aim at an immediate result. He does not expect that his advanced ideas will be readily taken up. His work is like that of the planter — for the future. His duty is to lay the foundation for those who are to come, and point the way. He lives and labors and hopes.
>
> *(Nikola Tesla)*

In diesem Kapitel wird anhand des in Abschnitt 2.1 vorgestellten Frameworks ABL gezeigt, wie die Integration strukturalistischer Methoden in Tesla unter Berücksichtigung der in 2.3 aufgeführten Anforderungen durchgeführt werden kann. Dabei werden drei Schwerpunkte gesetzt:

- **Evaluationsformen:** Wie in den Kapiteln 1 und 2 diskutiert, kann sich die Evaluation computerlinguistischer Verfahren als schwierig erweisen – die Reproduktion von Ergebnissen kann zudem durch zahlreiche Faktoren (wie unzugängliche Korpora oder ungenaue Versuchsbeschreibungen) erschwert werden. Stark vereinfachende bzw. zusammenfassende Evaluationsmaße, wie etwa Precision und Recall, lassen sich zudem häufig nur dann interpretieren, wenn auch Vergleichswerte zur Verfügung gestellt werden. Ein Schwerpunkt liegt daher auf der Entwicklung einer geeigneten Evaluationsmöglichkeit für Alignment-Verfahren, die die genannten Kritikpunkte ausräumt und es u.a. ermöglichen, die in diesem Kapitel aufgeführten Ergebnisse mit Hilfe randomisierter Referenzverfahren zu interpretieren.
- **Anwendung und Analyse von Alignment-Verfahren:** Mit Hilfe der entwickelten Evaluationskomponenten werden verschiedene in ABL umgesetzte Verfahren hinsichtlich ihrer Eignung für die Detektion syntaktischer Strukturen analysiert. Insbesondere wird untersucht, wie die Generierung von Strukturhypothesen während der *Align*-Phase (vgl. Abschnitt 2.2) optimiert werden kann, und wie sich der

bei widersprüchlichen Strukturen notwendige Auswahlprozess in der *Select*-Phase verbessern lässt. Im Zuge der Untersuchungen werden insgesamt fünf Korpora verwendet, so dass die durchgeführte Analyse eine umfassende Beurteilung der betrachteten Verfahren ermöglicht.

- **Tesla als computerlinguistisches Labor:** Aus obigen Punkten ergibt sich der dritte Schwerpunkt dieses Kapitels: Ein Framework, das für experimentelles computerlinguistisches Arbeiten entworfen wurde, muss nicht nur zulassen, dass komplexe Experimente, die aus zahlreichen spezialisierten Komponenten bestehen, modelliert werden können, sondern auch hinsichtlich Evaluations- und Analysemöglichkeiten hohe Flexibilität bieten, um wissenschaftliche Fragestellungen bestmöglich beantworten zu können – dieses Kapitel evaluiert daher auch Tesla als computerlinguistisches Labor, das den sich im Folgenden ergebenden Anforderungen in Bezug auf Versuchsaufbau und -ausführung, Analyse, Evaluation und Reproduktion genügen muss.

In Abschnitt 5.1 werden die hier verwendeten Korpora vorgestellt und untersucht. Daran anschließend wird in Abschnitt 5.2 diskutiert, auf welche Art eine Interpretation der von einem Alignment-Verfahren erzielten Ergebnisse ermöglicht werden kann, und wie die Ergebnisse im Vergleich zur bereits in Abschnitt 2.2.1 beschriebenen *Right*-Strategie zu interpretieren sind.

Abschnitt 5.3 behandelt die Funktionsweise der in ABL4J implementierten Alignment-Verfahren und diskutiert deren Vor- und Nachteile; in Abschnitt 5.3.1 wird zudem ein ergänzendes Verfahren vorgestellt, das effizientere und verbesserte Prozessierungsmöglichkeiten zulässt. In Abschnitt 5.4 wird die Evaluation dieser Verfahren durchgeführt; dabei werden neuen Hypothesen und Fragestellungen, die sich aus der Evaluation ergeben, aufgegriffen.

In Abschnitt 5.5 wird gezeigt, wie sich Alignment-Verfahren auf weitere computerlinguistische Forschungsbereiche anwenden lassen, indem sie exemplarisch sowohl zur morphosyntaktischen als auch zur semantischen Kategorisierung von Einzelwörtern genutzt werden.

Abschnitt 5.6 fasst schließlich die Ergebnisse zusammen – der Fokus liegt dabei auf Evaluation und Analyse der Alignment-Verfahren, während Tesla als computerlinguistisches Labor separat im letzten Kapitel dieser Arbeit diskutiert wird.

5.1 Korpora

Für eine Evaluation von ABL erscheint es zunächst naheliegend, Korpora zu verwenden, die eine exakte Reproduktion der bspw. in van Zaanen (2002) beschriebenen Ergebnisse ermöglichten sollten, wie etwa das dort untersuchte ATIS-Korpus (vgl. Abschnitt 2.2). Würde die Reproduktion der Daten dabei gelingen, könnten jedoch nur eingeschränkt auf die Eignung der Verfahren für die Verarbeitung weiterer Korpora geschlossen werden, da es sich bei ATIS um ein *Sublanguage*-Korpus handelt – wie die in Abschnitt 2.3 beschriebenen Arbeiten am *Medical Language Processor* nahelegen, kann eine derartige Einschränkung zwar aus pragmatischen Gründen sinnvoll sein, eine allgemeine Beurteilung eines Verfahrens ist aufgrund der mangelnden Übertragbarkeit der Resultate jedoch kaum möglich. Zudem ist das ATIS-Korpus ein Bestandteil der *Penn Treebank*, die nicht dem in Abschnitt 4.1.1 beschriebenen Prinzip des *Open Access* unterliegt, sondern für die eine Lizenz erworben werden muss – dies widerspricht dem Anspruch der Reproduzierbarkeit, der in dieser Arbeit verfolgt wird.

Um die Leistungsfähigkeit der untersuchten Verfahren besser beurteilen zu können, werden daher im Folgenden insgesamt fünf Korpora verwendet, die sich in mehreren Merkmalen unterscheiden: Zur Analyse gesprochener Sprache werden die Baumdatenbanken *Tübingen Treebank of Spoken English* (TüBa-E/S) und *Tübingen Treebank of Spoken German* (TüBa-D/S) genutzt; Schriftsprache wird anhand der *Tübingen Treebank of Written German* (TüBa-D/Z), welche aus Zeitungstexten der *tageszeitung* (taz) aufgebaut wurde, sowie anhand eines Ausschnitts des BNC[150] evaluiert. Einige sich im Laufe der Analyse ergebende Fragestellungen werden zudem am CHILDES-Korpus untersucht, das Transkriptionen von Gesprächen zwischen Kindern und Bezugspersonen enthält. Mit Ausnahme des BNC sind alle untersuchten Korpora (für nicht-kommerzielle, wissenschaftliche Verwendung) frei einsetzbar – die im Folgenden beschriebenen Experimente können daher mit geringem Aufwand reproduziert und analysiert werden.

Während die ersten drei Korpora manuell annotiert wurden und so als Gold-Standard verwendet werden können, bieten BNC und CHILDES keine Informationen zur Satzstruktur, sondern lediglich morphosyntaktische Annotationen. Allerdings zeigt die Analyse der Auszeichnungen in den Tübinger Datenbanken, dass auch dort nicht sämtliche struktu-

[150] Um ein englischsprachiges Äquivalent zum TüBa-D/Z nutzen zu können, werden sämtliche im BNC vorhandenen Artikel der Zeitung *The Guardian* verwendet, indem aus einer mit Teslas *Corpus Manager* durchgeführten Volltext-Suche (vgl. Abbildung 4.3 in Abschnitt 4.1.3) nach *The Guardian, Electronic Edition* eine neue *Document Selection* erzeugt wurde. Dieser Ausschnitt des BNC wird im Folgenden als *BNC-G* bezeichnet.

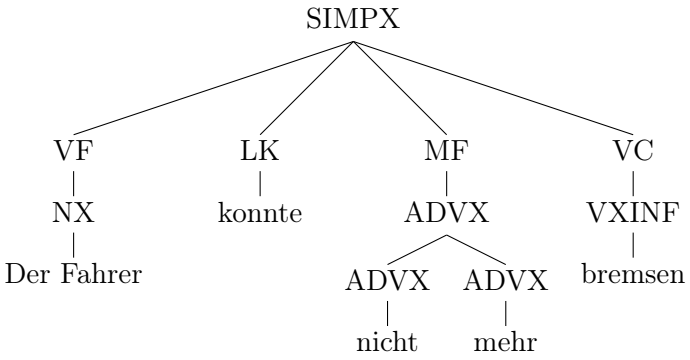

Abbildung 5.1: Beispiel einer unvollständigen syntaktischen Auszeichnung im TüBa-D/Z: Die Konstituenten *nicht mehr bremsen* und *konnte nicht mehr bremsen* sind hier nicht explizit ausgezeichnet.

rellen Beziehungen zwischen den Satzteilen ausgezeichnet wurden (vgl. Abbildung 5.1).

Daher wird die Auswertung der Verfahren dadurch erweitert, dass die vom Berkeley Parser generierten Strukturen als weitere Referenz herangezogen werden. Zwar ist dies nicht äquivalent zur Evaluation anhand eines Gold-Standards, jedoch bietet die Evaluation an einem unvollständig annotierten Gold-Standard ebenfalls nur eingeschränkte Aussagekraft – zudem kann auf diesem Weg ermittelt werden, inwieweit die untersuchten, unüberwacht arbeitenden Verfahren mit einem anhand von Beispielen trainierten Verfahren konkurrieren können. Schließlich ist in den in Abschnitt 5.4 vorgestellten Experimenten die kategorielle Auszeichnung aufgedeckter Strukturen irrelevant, so dass korrekt erkannte, aber fehlerhaft bezeichnete Strukturen (etwa die Auszeichnung einer Struktur durch VP anstelle von NP) die Ergebnisse nicht verfälschen können. Abbildung 5.2 zeigt die vom Berkeley Parser generierte Baumstruktur des Beispiels aus Abbildung 5.1.

Die Korpora TüBa-D/S und TüBa-E/S können als *Sublanguage*-Korpora interpretiert werden, da die dort festgehaltenen Transkriptionen im Wesentlichen auf Gespräche zur Terminvereinbarung beschränkt sind. Das CHILDES-Korpus nimmt hingegen eine Sonderrolle ein, da es sich hier um Transkriptionen der Gespräche zwischen Kindern und ihren Bezugspersonen handelt – thematisch ist es im Gegensatz zu den Tübinger Korpora nicht auf eine Domäne beschränkt, die syntaktische Struktur ist jedoch vergleichsweise einfach (siehe auch Abschnitt 5.4.3).

Tabelle 5.1 fasst die quantitativen Eigenschaften der untersuchten Korpora zusammen. Dabei ist anzumerken, dass Tokens, die Satzzeichen repräsentieren, hier und im Folgenden nicht berücksichtigt werden – dies hätte nicht nur das Type-Token-Verhältnis beeinflusst,

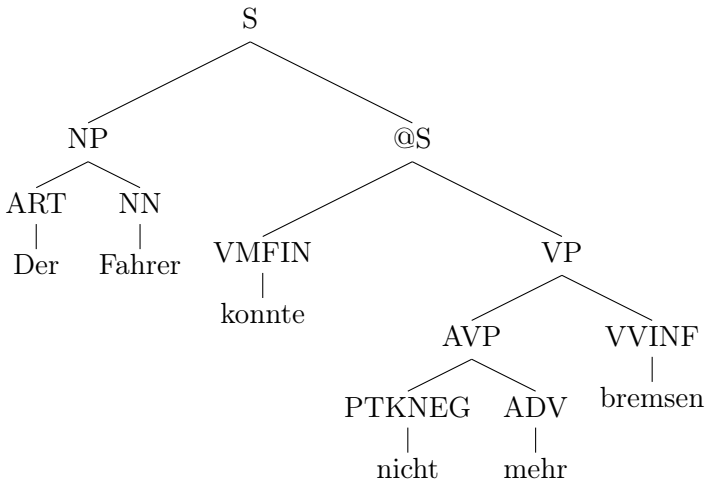

Abbildung 5.2: Beispiel der syntaktischen Auszeichnung durch den Berkeley Parser anhand des Satzes aus Abbildung 5.1. Der Vergleich zeigt, dass der Berkeley Parser für das dargestellte Beispiel detailliertere strukturelle Auszeichnungen generiert.

Korpus	Sätze	Wörter	Types	Strukturen (alle)	Strukturen (non-triv)
TüBa-D/S	38.342	304.223	6.682	466.266	129.952
TüBa-E/S	29.672	252.323	3.479	284.038	112.638
TüBa-D/Z	45.200	677.663	88.895	947.810	394.789
BNC-G	43.148	872.869	51.500	–	–
CHILDES	704.904	2.770.971	25.211	–	–

Tabelle 5.1: Überblick über die im Rahmen der hier durchgeführten Experimente verwendeten Korpora. Unter *Strukturen* werden hier alle manuell annotierten Auszeichnungen, die auf eine sequentielle Zeichenfolge abgebildet werden können, zusammengefasst. Separat aufgeführt sind zudem alle nicht-trivialen Strukturen (s.u.).

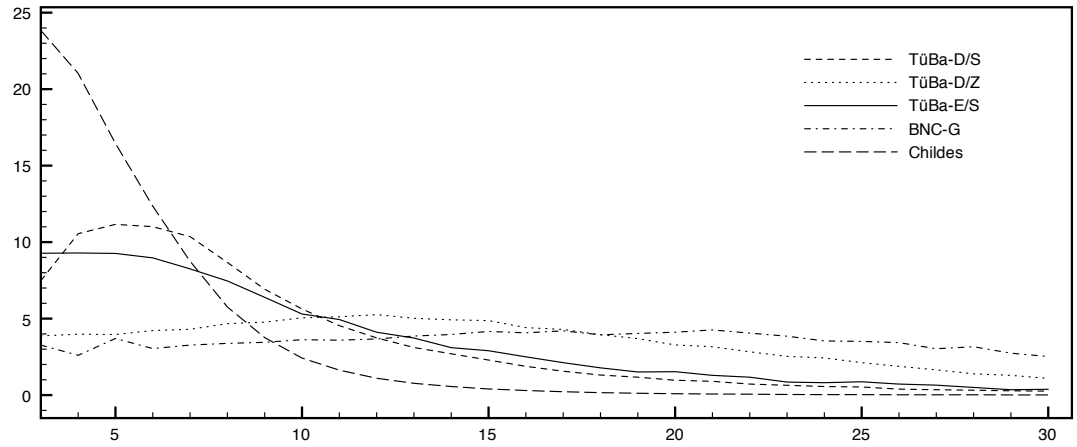

Abbildung 5.3: Häufigkeitsverteilung der Satzlängen in den untersuchten Korpora. Berücksichtigt werden alle Sätze der Länge 3 bis 30 (X-Achse). Zusätzlich zu den bereits erwähnten Korpora ist hier bereits das CHILDES-Korpus dargestellt, das in Abschnitt 5.4.3 vorgestellt wird.

sondern sich auch (negativ) auf die untersuchten Verfahren ausgewirkt.[151] Die Tabelle zeigt, dass sich die Korpora hinsichtlich ihrer Merkmale deutlich voneinander unterscheiden: Während bspw. die aus Schriftsprache erzeugten Korpora durchschnittlich etwa 15 bzw. 20 Tokens pro Satz enthalten, liegt die durchschnittliche Satzlänge bei TüBa-D/S und TüBa-E/S (8 bzw. 8,5 Tokens pro Satz) deutlich niedriger – Abbildung 5.3 zeigt den prozentualen Anteil der verschiedenen Satzlängen der Korpora.

Da die Berücksichtigung *sehr kurzer* Sätze (vgl. dazu die Überlegungen im folgenden Abschnitt 5.2) die Aussagekraft von Precision und Recall beeinträchtigt, wurden in den hier durchgeführten Experimenten nur Sätze berücksichtigt, die aus mindestens drei Wörtern bestehen. Zudem wurde, um eine bessere Vergleichbarkeit der Korpora zu ermöglichen, die maximale Länge akzeptierter Sätze auf 30 Wörter festgesetzt.

Auch im Verhältnis von Types und Tokens variieren die Korpora stark, wie die Daten in Tabelle 5.1 zeigen: In TüBa-D/S und TüBa-E/S wird jede Wortform durchschnittlich 45,5 bzw. 72,5 mal verwendet, in TüBa-D/Z und BNC-G hingegen nur 7,5 bzw. 17 mal. Abbildung 5.4 zeigt die Verteilung der Worthäufigkeiten im TüBa-D/S, die annähernd

[151] Eine Berücksichtigung von Satzzeichen würde zur Bildung von Hypothesen führen, die nicht syntaktisch oder semantisch, sondern ausschließlich orthographisch motiviert wären. Bereits in Abschnitt 2.2 wurde auf Saussures Beobachtung, dass es sich bei Schrift und Sprache um zwei verschiedene Systeme handele, hingewiesen (vgl. Seite 30f.) – die Berücksichtigung von Satzzeichen würde ein Strukturierungsmerkmal in die Untersuchung mit einbeziehen, das sprachlich nicht existiert. Satzzeichen würden zudem auch eine verstärkte Übergenerierung von Hypothesen bewirken, wie anhand der Überlegungen in Abschnitt 5.4.3 gezeigt wird.

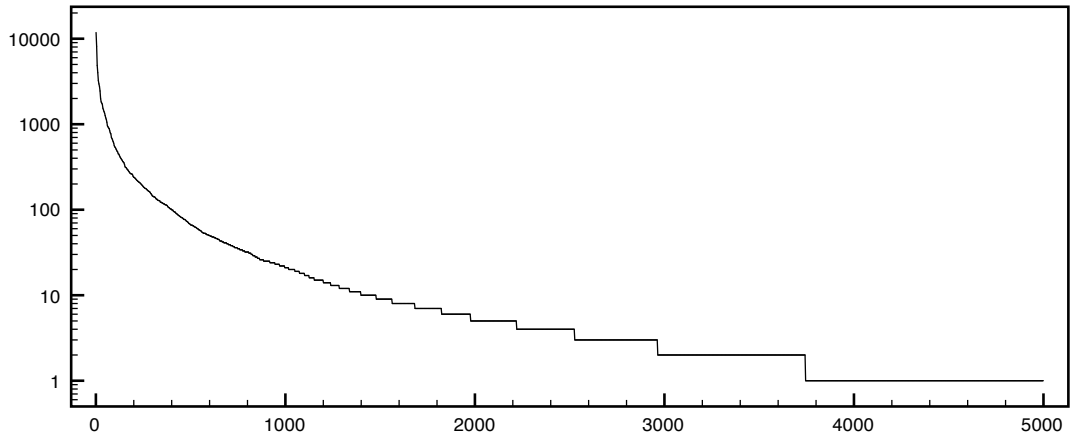

Abbildung 5.4: Verteilung der Worthäufigkeiten im TüBa-D/S. Auf der X-Achse sind die Wörter nach Häufigkeit absteigend sortiert, auf der Y-Achse ist die absolute Häufigkeit (logarithmisch skaliert) angegeben.

dem Zipfschen Gesetz (vgl. Manning & Schütze 1999, S. 23) entspricht.

Neben der Beschränkung der untersuchten Sätze hinsichtlich ihrer Länge sowie der Entfernung von Interpunktion wird keine weitere Filterung der Annotationen, die von den *Reader*-Komponenten (vgl. Abschnitt 4.1.3.1 sowie Anhang B.1) der analysierten Korpora produziert werden, durchgeführt. Die zu evaluierenden Strukturen werden jedoch ebenso wie die Referenzstrukturen einer zusätzlichen Prozessierung unterzogen, um Duplikate[152] zu filtern, durch die das Ergebnis verzerrt werden kann. Analog zu der von Klein & Manning (2002, Abschnitt 4) beschriebenen Vorgehensweise werden zudem alle *trivialen* Strukturen ignoriert, d.h. Strukturen, die ein einzelnes Wort oder einen vollständigen Satz umfassen. Wie Tabelle 5.1 zeigt, ist mehr als die Hälfte der manuell ausgezeichneten Strukturen in den untersuchten Korpora trivial – dies bedeutet, dass ein naives Verfahren, welches lediglich triviale Strukturen auszeichnet, vergleichsweise gute, jedoch wenig aussagekräftige Ergebnisse erzielen würde (vgl. auch van Zaanen & Geertzen 2008).

Abbildung 5.5 zeigt eine vereinfachte Darstellung des initialen Versuchsaufbaus, der sich aus den hier beschriebenen Überlegungen ergibt, und der im folgenden Abschnitt hinsichtlich der zu erwartenden Resultate analysiert wird. Bei allen hier und im Folgenden dargestellten Experimenten handelt es sich um schematische Darstellungen, in denen nicht sämtliche tatsächliche benötigten Komponenten abgebildet sind, auch wurde auf die Darstellung aller notwendigen Verknüpfungen zwischen konsumierten und produzierten

[152] Diese können bspw. dann auftreten, wenn eine Wortsequenz mehrfach annotiert wird (etwa als Nomen und als Nominalphrase).

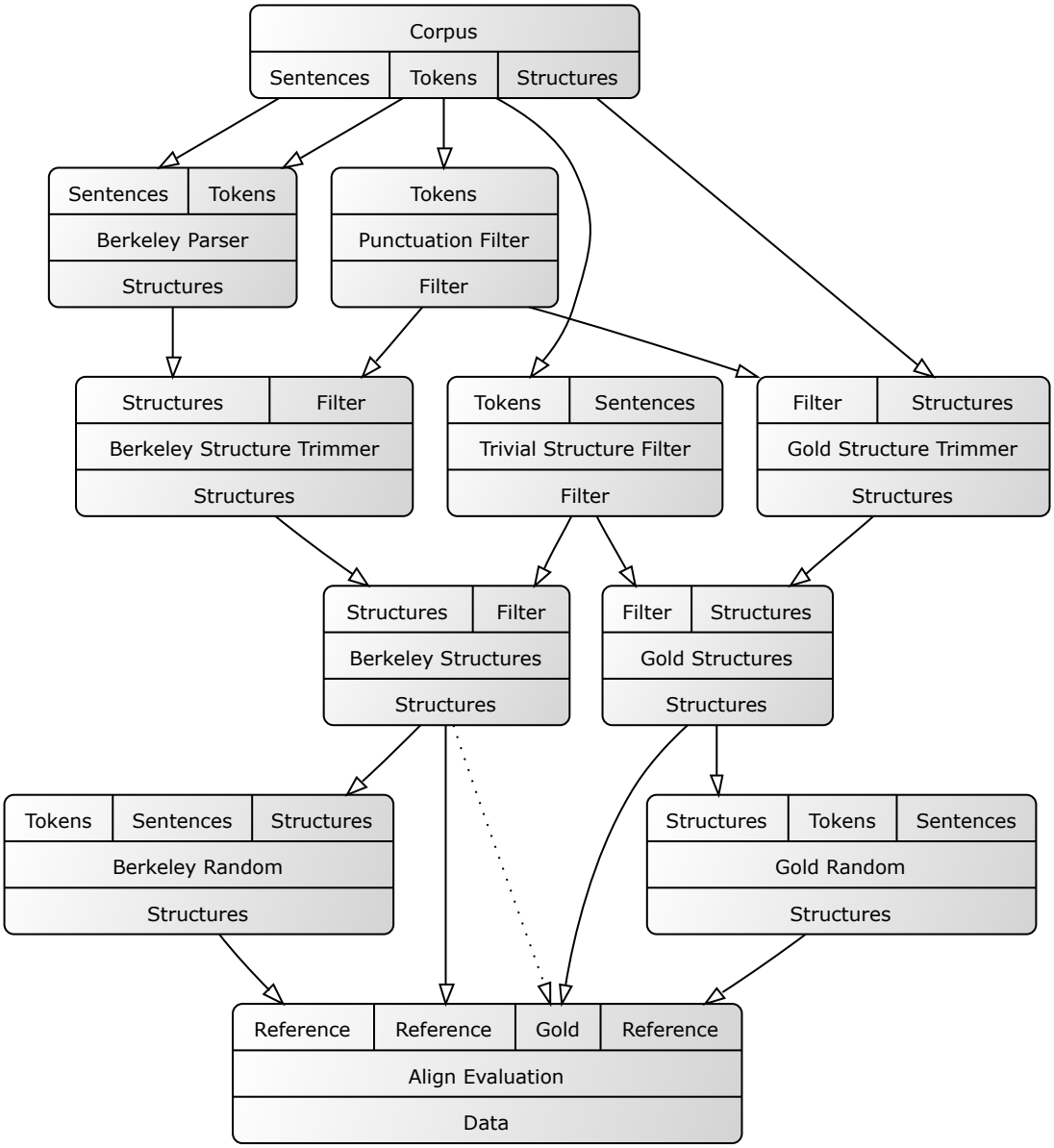

Abbildung 5.5: Schematische Abbildung der Präprozessierung der hier analysierten Korpora. Verschiedene Filter-Komponenten werden verwendet, um die beschriebene Filterung von Interpunktionszeichen und trivialen Strukturen durchzuführen (auf Abbildung des Satzlängen-Filters und einige Verknüpfungen wurde zwecks Übersichtlichkeit verzichtet). Die punktierte Verbindung repräsentiert die alternative Evaluierung anhand der vom Berkeley Parser aufgedeckten Strukturen. Die Komponenten *Berkeley Random* und *Gold Random* werden für die in Abschnitt 5.2 beschriebene Berechnung von Erwartungswerten verwendet.

Rollen verzichtet – für eine Reproduktion der Experimente sei auf Anhang C verwiesen.

Die in Abbildung 5.5 dargestellten Filter-Komponenten sind in Bezug auf die produzierte Rolle identisch[153], unterscheiden sich jedoch in ihrer Implementation: So erzeugt bspw. die in Abbildung 5.5 als *Trivial Structure Filter* bezeichnete Komponente einen Filter auf Basis der Positionen aller Wörter, während der *Punctuation Filter* die Interfaces der konsumierten Annotationen analysiert und hier so konfiguriert wurde, dass alle Annotationen, deren DataObject von `IPunctuationToken` abgeleitet ist, verworfen werden (für eine ausführlichere Beschreibung dieser und weiterer Filter siehe Anhang B.3).

5.2 Erwartungswerte

Bevor eine Evaluation der Verfahren durchgeführt werden kann, muss zunächst untersucht werden, welche Ergebnisse mindestens erwartet werden können, um so eine untere Schranke zu definieren, von der sich die analysierten Verfahren abgrenzen müssen. Analog dazu kann auch eine obere Schranke definiert werden, denn es ist nicht anzunehmen, dass eines der unüberwachten Alignment-Verfahren Ergebnisse produzieren kann, die besser als die eines Verfahrens sind, das auf zusätzliches linguistisches Wissen zurückgreift. Daher werden unterschiedliche Vergleichswerte ermittelt: Zum einen wird untersucht, wie hoch Precision und Recall des Berkeley-Parsers gegenüber dem Gold-Standard (vgl. Abschnitt 4.1.3) sind, zum anderen wird ein als *Random* bezeichnetes Verfahren implementiert, das ausschließlich zufallsgesteuert Hypothesen über Strukturgrenzen generiert. Die Anzahl der generierten Hypothesen wird dabei anhand einer Referenz ermittelt: Für jede Satzlänge wird zunächst die durchschnittliche Anzahl vorhandener Referenzstrukturen berechnet, anschließend wird für jeden Satz eine entsprechende Menge von Strukturen (mit zufällig ermittelten Grenzen) erzeugt. Abbildung 5.6 zeigt beispielsweise, dass in Sätzen der Länge 20 im TüBa-D/S durchschnittlich etwa 10 Strukturen annotiert werden, entsprechend erzeugt auch das *Random*-Verfahren für diese Sätze 10 Strukturen.

Der Vorteil des Verfahrens liegt darin, dass es sich automatisch an die Produktivität der Referenzkomponente anpasst, die bei den hier untersuchten Verfahren abhängig von der

[153] Im Gegensatz zu den Annotationen, die von Tokenizern oder POS-Taggern generiert wurden, sind die in der Rolle *Filter* referenzierten `IFilter`-DataObjects nicht mit Textstellen verankert: Stattdessen bieten sie die Methoden `boolean accepts(Annotation a)` und `boolean rejects(Annotation a)`, die von Komponenten, die eine oder mehrere Instanzen eines Filters konsumieren, verwendet werden können, um die zu verarbeitenden Annotationen einzuschränken. Filter sind somit ein einfaches Beispiel dafür, wie das *Tesla Role System* genutzt werden kann, um Funktionen, die nicht innerhalb eines *klassischen* Annotationsgraphen umgesetzt werden können, zu integrieren. Aus Anwendersicht ist daher keine Unterscheidung zwischen Filtern und den bisher beschriebenen Komponenten notwendig, da sie sich, wie Abbildung 5.5 zeigt, analog zu diesen in Experimenten verwenden lassen.

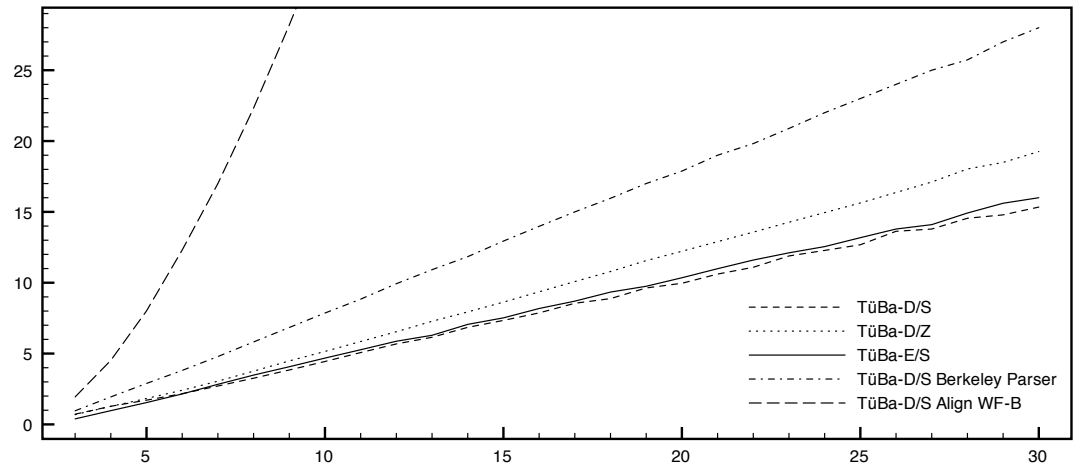

Abbildung 5.6: Durchschnittliche Anzahl struktureller Auszeichnungen (Y-Achse) in Bezug auf die Satzlänge (X-Achse). Neben den manuell erstellten Annotationen der Tübinger Korpora wird für das TüBa-D/S auch die Anzahl der vom Berkeley Parser generierten Strukturen sowie (ausschnittsweise) die Anzahl der Hypothesen, die eines der im folgenden Abschnitt untersuchten Alignment-Verfahren (WF-B) erzeugt, dargestellt.

Satzlänge, der Anzahl prozessierter Sätze und dem Type/Token-Verhältnis ist. Die Tabellen 5.2 und 5.3 zeigen anhand des TüBa-D/S, dass sich die derart ermittelten Ergebnisse nutzen lassen, die Qualität weiterer Verfahren zu bewerten: So erreicht das beschriebene *Random*-Verfahren einen F-Score[154] von 9,58, wenn der Gold-Standard als Referenz verwendet wird, und einen F-Score von 12,08, falls die vom Berkeley-Parser generierten Strukturen vom *Random*-Verfahren genutzt werden – die Ergebnisse können als untere Schranke betrachtet werden, an der weitere Verfahren gemessen werden müssen.

Wie Tabelle 5.2 zeigt, liefert der Berkeley Parser einen deutlich vom Zufallsverfahren abweichenden Recall, da die manuellen Auszeichnungen größtenteils den Bereichen entsprechen, die auch vom Parser als Strukturen markiert wurden. Werden hingegen die Precision und die absolute Anzahl generierter Hypothesen analysiert, zeigt sich, dass der Parser hier keine guten Werte liefern kann, da fast doppelt so viele Strukturen ausgezeichnet wurden, wie im Gold-Standard tatsächlich vorhanden sind. Dies kann jedoch nicht als

[154]Um Precision und Recall verschiedener Verfahren besser vergleichen zu können, wird häufig ein als F-Score bezeichnetes Maß verwendet, in dem beide Werte (ggfs. mit unterschiedlicher Gewichtung) kombiniert werden. Hier und im Folgenden bezeichnet der F-Score das harmonische Mittel beider Werte, wie bspw. in Manning & Schütze 1999, Kapitel 8.1 definiert durch

$$F\text{-}Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{5.1}$$

	Hypothesen	Precision	Recall	F-Score
Gold	117.865	100	100	100
Gold Random	118.509	9,55	9,6	9,58
Berkeley Parser	206.403	36,27	63,52	46,18
Berkeley Random	210.105	9,43	16,81	12,08

Tabelle 5.2: Evaluation unterschiedlicher Vergleichsverfahren anhand der manuell ausgezeichneten Strukturen im Gold-Standard TüBa-D/S. Neben dem Berkeley Parser sind die Ergebnisse von zwei Instanzen der hier beschriebenen Random-Komponente ausgeführt, die sich bezüglich der konsumierten Referenzen unterscheiden: Die als *Gold Random* bezeichnete Komponente nutzt die Annotationen des Gold-Standards zur Berechnung der Anzahl zu produzierender Strukturen, während die *Berkeley Random*-Komponente die vom Berkeley Parser generierten Strukturen verwendet. Abweichungen zu Tabelle 5.1 ergeben sich dadurch, dass hier nur Sätze mit 3 bis 30 Wörtern berücksichtigt werden.

Nachteil des Parsers interpretiert werden, sondern ist vielmehr auf die unvollständige Annotation des TüBa-D/S zurückzuführen (siehe Abschnitt 5.1).

Die Ergebnisse der *Random*-Verfahren lassen sich mathematisch wie folgt erklären: Für eine Sequenz von n Symbolen existiert genau eine Möglichkeit, eine Struktur der Länge n zu annotieren, zwei Möglichkeiten für eine Struktur der Länge $n-1$, drei für die Länge $n-2$ usw., und schließlich n Möglichkeiten für Strukturen der Länge 1. Die Summe aller Möglichkeiten liegt also bei $\sum_{i=1}^{n} i$, was nach Gauß'scher Summenformel umgeformt werden kann zu $n \cdot \frac{n+1}{2}$. Da triviale Strukturen hier nicht berücksichtigt werden sollen, müssen das erste und das letzte Element der Summe ausgeschlossen und die Formel entsprechend angepasst werden:

$$\sum_{i=2}^{n-1} i \equiv (n-1) \cdot \frac{n}{2} - 1 \tag{5.2}$$

Wird mit $n = 8$ die durchschnittliche Satzlänge des TüBa-D/S in die Formel eingesetzt, ergeben sich so 27 unterschiedliche Möglichkeiten, eine strukturelle Auszeichnung für einen aus acht Wörtern bestehenden Satz zu generieren. Aus Tabelle 5.1 lässt sich entnehmen, dass im Gold-Korpus durchschnittlich 3,4 Strukturen pro Satz ausgezeichnet wurden: Ein zufallsbasiertes Verfahren müsste also in etwa 12,6 Prozent aller Fälle eine korrekte Struktur generieren. Die Abweichung zwischen diesem Wert und dem in Tabelle 5.1 ermittelten Ergebnis von 9,59 Prozent lässt sich darauf zurückführen, dass es sich bei der Verwendung der durchschnittlichen Satzlänge um eine starke Vereinfachung handelt:

	Hypothesen	Precision	Recall	F-Score
Berkeley Parser	206.377	100	100	100
Berkeley Random	210.105	15,19	15,46	15,33

Tabelle 5.3: Auswertung der vom *Random*-Verfahren generierten Strukturen bezüglich der syntaktischen Auszeichnung durch den Berkeley Parser am TüBa-D/S.

Die Wahrscheinlichkeit, zufällig eine im Gold-Standard enthaltene Struktur zu generieren, sinkt polynomiell mit steigender Satzlänge, was in der Berechnung nicht berücksichtigt wird.[155]

Die im Folgenden untersuchten Verfahren müssen die hier ermittelten Erwartungswerte übersteigen. In Abschnitt 2.1.3 wurde jedoch bereits auf die formale Analyse von Gold (1967) hingewiesen – Gold beweist dort, dass lediglich formale Sprachen mit finiter Kardinalität[156] anhand ausschließlich positiver Beispiele fehlerfrei erlernt werden können, während für die Konstruktion einer Grammatik für reguläre Sprachen bereits ein komplexeres, als *Informant* bezeichnetes System benötigt wird, das positive oder negative Rückmeldung zu beliebigen Sätzen geben kann und das es einem Lernalgorithmus ermöglicht, Hypothesen zu validieren oder zu falsifizieren. Sofern der Beweis von Gold nicht widerlegt wird, ist daher ausgeschlossen, dass ein ausschließlich auf Texten operierender Lernalgorithmus anhand der hier untersuchten Korpora eine fehlerfreie Grammatik generiert – die im TüBa-D/S festgehaltenen Gespräche können zwar, da es sich in erster Linie um Terminvereinbarungen und geschäftliche Absprachen handelt, als eine Form der *Sublanguage* im Sinne Harris (vgl. Abschnitt 2.1.2) interpretiert werden, sie sind jedoch nicht formal eingeschränkt und enthalten bspw. auch Einschübe oder fragmentarische Äußerungen. Dies muss daher bei der folgenden Analyse berücksichtigt werden: Anhand der hier verwendeten Korpora kann lediglich analysiert werden, ob die Qualität der untersuchten Verfahren

[155] Nach dieser Überlegung sollten ebenfalls 12,6 Prozent der vom *Berkeley Random*-Verfahren generierten Strukturen korrekt sein, insgesamt also etwa 26.900, woraus sich ein zu erwartender Recall von 23 Prozent ergibt. Allerdings steigt die Anzahl der vom Berkeley Parser generierten Strukturen in Abhängigkeit zur Satzlänge deutlich stärker als im Gold-Standard (vgl. Abbildung 5.6), weshalb der ermittelte Recall entsprechend niedriger liegt. Dies ist auch dann der Fall, wenn nicht die im Gold-Standard enthaltenen Auszeichnungen als Referenz verwendet werden, sondern die vom Berkeley Parser generierten Strukturen, wie in Tabelle 5.3 zusammengefasst. Hier enthält jeder Satz durchschnittlich 5,4 korrekte Hypothesen, so dass die Wahrscheinlichkeit, zufallsbasiert eine dieser Hypothesen zu erzeugen, bei etwa 20 Prozent liegt. Etwa 42.000 der vom *Random*-Verfahren generierten Hypothesen sollten demnach in der Referenzmenge gefunden werden, was einem zu erwartenden Recall von 20 Prozent entspricht, der ebenfalls über dem empirisch ermittelten Ergebnis liegt.

[156] Dies bezeichnet Sprachen, die eine endliche Menge möglicher Sätze umfassen, was bspw. durch die Beschränkung von Vokabular und maximaler Satzlänge ermöglicht werden kann.

signifikant von den Ergebnissen des *Random*-Verfahrens abweichen, eine annähernd fehlerfreie strukturelle Auszeichung ist hingegen nicht zu erwarten.

Ebenfalls kann nicht davon ausgegangen werden, dass die hier untersuchten Verfahren bessere Ergebnisse als das *Right*-Verfahren liefern werden, da dieses eine Strategie verfolgt, die zwei linguistisch motivierte Annahmen umsetzt: Zum einen sind die generierten Strukturbäume stets binär, zum anderen sind sie dabei rechtsverzweigend – dies führt dazu, dass insbesondere im Vergleich mit den vom *Berkeley Parser* generierten Strukturbäumen (die ebenfalls binär und i.d.R. rechtsverzweigend sind) ein hoher Grad an Übereinstimmung erreicht wird.[157] Beispiel 5.1 veranschaulicht dies anhand der bereits in Abbildung 5.2 dargestellten Analyse durch den *Berkeley Parser* (5.1.a), die der Analyse durch *Right* (5.1.b) gegenübergestellt ist, ohne dass triviale Strukturen, d.h. Wort- und Satzgrenzen, berücksichtigt bzw. abgebildet werden.

5.1 a. [Der Fahrer] [konnte [[nicht mehr] bremsen]]
b. Der [Fahrer [konnte [nicht [mehr bremsen]]]]

Sowohl Precision als auch Recall liegen in diesem Fall bei 50 Prozent, da zwei der vier von *Right* und *Berkeley Parser* generierten Strukturen identisch sind. Zwar nimmt der Erfolg von *Right* mit zunehmender Länge und Komplexität der analysierten Sätze ab, dennoch führen die zugrundeliegenden Annahmen dazu, dass das Verfahren gegenüber den im folgenden Abschnitt vorgestellten Alignment-Ansätzen einen bessere Ausgangsposition einnehmen kann, und dass es hinsichtlich des erreichten F-Scores von keinem Alignment-Verfahren übertroffen wird. Es ist jedoch nicht das Ziel dieser Arbeit, ein möglichst hochwertiges Verfahren zur Auszeichnung syntaktischer Strukturen zu entwickeln, sondern vielmehr zu analysieren, unter welchen Voraussetzungen und mit welcher Form der Prozessierung und Konfiguration Alignment-basierte Ansätze möglichst gute Ergebnisse erzielen können. Zwar ist es durchaus vorstellbar, dass Alignment-Ansätze und *Right*-Strategie so kombiniert werden können, dass der sich daraus ergebende Ansatz den einzelnen Verfahren überlegen ist; vor einer derartigen Umsetzung muss jedoch eine ausführliche Analyse und Evaluation von Alignment-Verfahren durchgeführt werden – dies ist Inhalt der folgenden

[157] Die Argumentation folgt hierbei Klein & Manning (2002, Kapitel 4), die ebenfalls die *Right*-Strategie (dort als *RBRANCH* bezeichnet) zur Beurteilung eines unüberwacht arbeitenden Verfahrens zur Strukturaufdeckung verwenden:

> RBRANCH is often used as a baseline for supervised systems, but exploits a systematic right-branching tendency of English. An unsupervised system has no *a priori* reason to prefer right chains to left chains [...]. A system need not beat RBRANCH to claim partial success at grammar induction.

Abschnitte.

5.3 Untersuchte Alignment-Verfahren

Die Komponente *ABL Align* wird mit unterschiedlichen Konfigurationen analysiert, um die Eignung verschiedener Alignment-Verfahren zu untersuchen. Zudem werden mehrere Vergleichsverfahren ausgeführt, so dass insgesamt sechs Methoden zur Erzeugung von Strukturhypothesen betrachtet werden:

- Das bereits im vorherigen Abschnitt verwendete *Random*-Verfahren, durch das es ermöglicht wird, Vergleichswerte für jedes untersuchte Verfahren zu berechnen,
- die in Abschnitt 2.2.1 beschriebene *Right*-Strategie, die als weitere Referenz für die Interpretation der Ergebnisse herangezogen wird,
- eine als *Left* bezeichnete Methode, die analog zum *Right*-Verfahren arbeitet, jedoch im Gegensatz zu diesem die Endposition einer Hypothese variiert und die Startposition fixiert,
- das in Abschnitt 2.2.1 vorgestellte *Wagner-Fischer-Alignment*,
- *All*-Alignment, welches nicht auf der Berechung einer Edit-Distance basiert, sondern auf Basis der Schnittmenge aller Symbole, die in zwei Sequenzen enthalten sind, sämtliche möglichen Alignments generiert (vgl. van Zaanen 2002, Abschnitt 4.1.2), und
- das *Suffix*-Verfahren (s.u.), das mit Hilfe eines modifizierten Suffix- bzw. N-Gramm-Baums ein mit Wagner-Fischer- und All-Alignment vergleichbares Detektionsverfahren umsetzt, dabei jedoch ein deutlich effizienteres Laufzeitverhalten bietet.

Mit Ausnahme der *Random*- und *Suffix*-Methoden handelt es sich bei allen der oben genannten Verfahren um Java-Adaptionen der in ABL 1.0 implementierten Algorithmen.[158]

[158] Bei der Portierung wurde versucht, die Ergebnisse von ABL exakt zu reproduzieren, soweit dies akzeptabel war: So verwendet ABL bspw. in dem hier nicht weiter besprochenen *Both*-Alignment eine C-Bibliothek zur Generierung von Zufallszahlen. Während der Portierung wurde diese zwar über die *Java Native Interface*-Schnittstelle (JNI) eingebunden, anschließend jedoch wieder entfernt. Weiterhin wurden zwei Änderungen am Code von ABL durchgeführt, um Kompatibilität zu erreichen: Zum einen enthält die vom Wagner-Fischer-Verfahren verwendete Klasse `Sub_dis` einen Initialisierungsfehler, zum anderen musste die Rundungsgenauigkeit von Java und C angeglichen werden. Letztere Änderung kann zwar als Eingriff in die Funktionsweise der Ursprungsanwendung betrachtet wer-

Die Konfiguration der Align-Komponenten entspricht der Standard-Konfiguration von ABL: Die Optionen *Exhaustive* und *Don't Merge* (siehe Anhang B.5.1) wurden deaktiviert; als Grundlage der Hypothesengenerierung wurde die Annahme, dass der bei einem Alignment abweichende Teil eines Satzes eine Struktur definiert (*Part Type unequal*, siehe ebenfalls Anhang B.5.1), verwendet.

Das Suffix-Verfahren basiert hingegen auf einem völlig anderen Ansatz und wurde daher in einer separaten Komponente umgesetzt, die keine Abhängigkeiten zu den ABL-Komponenten aufweist – stattdessen werden hier die in Abschnitt 4.1.4 beschriebenen Vorteile des *Tesla Role System* genutzt, um sowohl die Kompatibilität zwischen unterschiedlich implementierten Komponenten zur Strukturaufdeckung zu erreichen als auch die Funktionalität der Basisrolle zu erweitern. Dieses Verfahren wird im Folgenden kurz vorgestellt.

5.3.1 Suffix Alignment

In Abschnitt 5.4 wird sich zeigen, dass die niedrigen Precision-Werte der untersuchten Edit-Distance-Verfahren zum großen Teil auf die Übergenerierung von Hypothesen zurückzuführen sind; zudem wird durch das Laufzeitverhalten dieser Verfahren die Evaluation unterschiedlicher Konfigurationen erschwert: Die bereits in Abschnitt 2.2.1 erwähnte Laufzeit von $O(n^2m^2)$ bei n Sätzen mit m Wörtern führt in der praktischen Anwendung dazu, dass bspw. die Ausführung des Wagner-Fischer-Alignments auf dem TüBa-D/Z-Korpus mehrere Stunden dauert, während ein Verfahren mit linearer Laufzeit, wie etwa *Left*, nur wenige Minuten benötigt. Dieser Nachteil kann jedoch durch eine Anpassung der verwendeten Algorithmen und Datenstrukturen behoben werden: So beschreiben Geertzen & Zaanen (2004) ein Verfahren, das ebenso wie die Edit-Distance-Verfahren auf dem Alignment von Sätzen basiert, jedoch mit $O(n)$ ein lineares Laufzeitverhalten bietet. Dazu verwenden sie einen generalisierten Suffixbaum, der im Folgenden kurz anhand eines Beispiels erläutert wird – für eine ausführliche Beschreibung sei auf Gusfield (1997), zur Erläuterung effizienter Algorithmen zum Aufbau eines Suffixbaumes auf Ukkonen (1995) und Farach (1997) verwiesen.

Ein (nicht generalisierter) Suffixbaum ist ein Baum, in dem jedes Suffix einer Symbolkette als Pfad von der Wurzel zu einem Blatt abgebildet ist. Dies ermöglicht u.a. eine sehr

den, es ist jedoch davon auszugehen, dass dies keine relevanten Auswirkungen auf die erzielten Ergebnisse hat. Durch JUnit-Tests an Beispieltexten wurde gewährleistet, dass die hier verwendeten ABL-Implementationen der Verfahren *Left*, *Right* und *Wagner-Fischer* die Ergebnisse von ABL exakt reproduzieren – beim *All*-Alignment variiert die Reihenfolge von Kategoriebezeichnungen (vgl. Listing 2.1 auf Seite 35) leicht, dies hat jedoch auf die Strukturdetektion keine Auswirkungen.

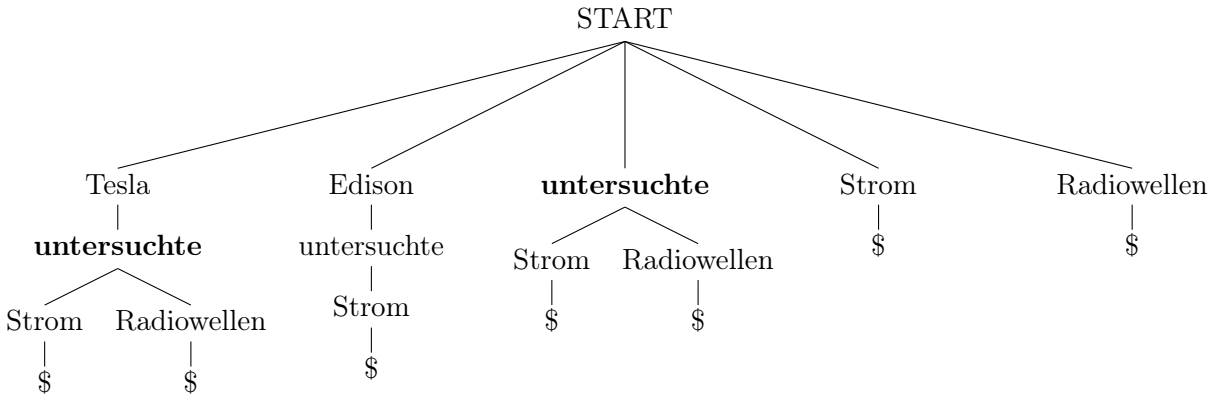

Abbildung 5.7: Generalisierter Suffixbaum zu den Sätzen *Tesla untersuchte Strom*, *Tesla untersuchte Radiowellen* und *Edison untersuchte Strom*.

effiziente Suche nach einem gegebenen Muster, da die dafür benötigte Laufzeit lediglich von der Länge des Musters abhängt.

In einem generalisierten Suffixbaum können verschiedene Symbolketten gleichzeitig gespeichert (und somit auch durchsucht) werden – um die Symbolketten voneinander abzugrenzen, wird das Ende jeder Kette durch ein Terminierungssymbol markiert. Der in Abbildung 5.7 dargestellte Suffixbaum enthält alle Suffixe der Sätze *Tesla untersuchte Strom*, *Tesla untersuchte Radiowellen* und *Edison untersuchte Strom*.

Geertzen & Zaanen (2004) stellen fest, dass die inneren Knoten eines Suffixbaumes, denen zwei oder mehr Kindknoten folgen (bspw. die markierten Knoten in Abbildung 5.7), zur Generierung struktureller Hypothesen genutzt werden können: Ein solcher Knoten markiert das Ende einer mehrfach detektierten Teilkette, die anschließend auf unterschiedliche Weise fortgesetzt wird, und eignet sich somit als Hypothese über den Beginn einer syntaktischen Struktur. Analog dazu kann ein *Präfixbaum*, in dem die Symbole jeder Kette in umgekehrter Reihenfolge enthalten sind, für die Bildung von Hypothesen über das Ende einer Struktur genutzt werden (vgl. Abbildung 5.8).

Anhand beider Bäume können die in Beispiel 5.2 aufgeführten Hypothesen generiert werden – das Ergebnis ist hier identisch mit dem in Abschnitt 2.2.1 vorgestellten Wagner-Fischer-Alignment, da in beiden Fällen die Abweichungen zwischen *matchenden* Symbolen detektiert und als Grundlage zur Hypothesenbildung verwendet werden. Da jedoch Suffixbäume in linearer Zeit konstruiert und durchsucht werden können, ist das hier skizzierte Verfahren deutlich effizienter.

5.2 a. [Tesla]$_2$ untersuchte [Strom]$_1$
b. [Edison]$_2$ untersuchte [Strom]$_1$

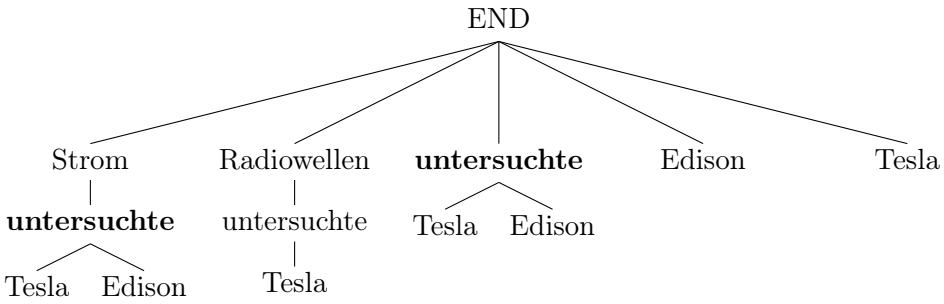

Abbildung 5.8: Generalisierter Präfixbaum zu den Sätzen *Tesla untersuchte Strom*, *Tesla untersuchte Radiowellen* und *Edison untersuchte Strom*. Auf das Terminierungssymbol wurde hier zugunsten der Übersichtlichkeit verzichtet.

c. [Tesla]$_2$ untersuchte [Radiowellen]$_1$

Die Betrachtung der Bäume in Abbildung 5.7 und 5.8 zeigt jedoch auch, dass es nicht notwendig ist, einen Suffixbaum vollständig zu erzeugen, um Strukturhypothesen aufzustellen: Es genügt, den Baum bis zur zweiten Ebene aufzubauen, da jeder verzweigende innere Knoten konstruktionsbedingt auch als unmittelbarer Kindknoten der Wurzel vorkommen muss. Für den hier beschriebenen Anwendungsfall ist die Erzeugung eines *Bigramm-Baumes*[159] daher zunächst ausreichend. Dies vermeidet nicht nur den algorithmischen Aufwand, der zur effizienten Erzeugung eines Suffixbaumes benötigt wird, sondern reduziert gleichzeitig auch den benötigten Speicherbedarf.[160] Daher wurden drei Tesla-Komponenten implementiert, die unabhängig von den ABL4J-Komponenten verschiedene Variationen dieses Ansatzes umsetzen können:

- Die *NGram Tree Generator*-Komponente erzeugt generalisierte Suffix- oder Präfixbäume, wobei die maximale Tiefe des Baums beschränkt werden kann.
- Eine als *Boundaries Detector* bezeichnete Komponente verarbeitet die vom *NGram Tree Generator* produzierten Datenstrukturen und erzeugt (gewichtete) Hypothesen für Start- und Endpositionen struktureller Auszeichnungen.
- Die *Hypothesis Generator*-Komponente nutzt schließlich diese Informationen, um Strukturhypothesen zu generieren.

[159] Der Begriff *Bigramm* bezeichnet eine aus zwei Elementen bestehende, eindeutige Subsequenz und stellt einen Spezialfall eines N-Gramms dar – die Untersuchung aller Bigramme eines Korpus kann bspw. in einer Kollokationsanalyse genutzt werden, um Wortpaare zu finden, die unverhältnismäßig oft in Kombination auftauchen, wie etwa *New York* oder *Los Angeles* (vgl. auch Manning & Schütze 1999, S. 31ff).

[160] Alternativ zu Suffixbäumen lassen sich auch Suffix-Arrays verwenden, die ähnlich effizient sind, jedoch deutlich weniger Speicher benötigen (vgl. bspw. Stehouwer & van Zaanen 2010a).

Abbildung 5.9 veranschaulicht, wie die Komponenten aufeinander aufbauen und die für die Generierung gewichteter struktureller Hypothesen notwendigen Informationen schrittweise aggregieren: Zunächst werden die zu prozessierenden Sequenzen in N-Gramm-Bäume überführt, wie bereits beschrieben. Für jeden Knoten wird dabei eine Liste von Positionen verwaltet, durch die unmittelbar auf die Teilsequenzen, die der jeweilige Knoten repräsentiert, zugegriffen werden kann (Abbildung 5.9 I). Anschließend werden die in den Knoten gespeicherten Daten auf die Ausgangssequenzen übertragen: Wie Abbildung 5.9 II veranschaulicht, sind danach sämtliche möglichen Start- und Endpositionen in jedem Satz annotiert. Jede dieser Annotationen enthält Informationen darüber, in welchen Sequenzen das jeweilige Symbol vorkommt, welche Position es innerhalb dieser Sequenz einnimmt und wie groß der maximale gemeinsame Kontext im jeweiligen Fall ist. Letztere Information wird dabei anhand der Tiefe der Knoten des N-Gramm-Baumes ermittelt, indem dann, wenn eine Bezugsposition mehrfach im Baum enthalten ist, der tiefste Knoten verwendet wird. In Abbildung 5.9 I und II kann dies anhand der Knoten zu *erfand* verdeutlicht werden: Auf Tiefe 1 sind drei Positionen mit dem Knoten assoziiert, auf Tiefe 2 nur noch die Positionen in den Sequenzen A und B. Für das Vorkommen von *erfand* in Sequenz A ergibt sich somit, dass der maximale gemeinsame Kontext bezüglich B aus zwei Elementen besteht, während er bezüglich C lediglich ein Element lang ist (vgl. die in Abbildung 5.9, II dargestellte Annotation von *erfand* in Sequenz A). Der *Hypothesis Generator* erzeugt schließlich anhand der vom *Boundaries Detector* generierten Positionsangaben Strukturhypothesen, indem für jede Sequenz alle Kombinationen aus Start- und Endpositionen dahingehend untersucht werden, ob die dort referenzierten Positionen eine gültige Schnittmenge bilden, d.h. ob mindestens ein Paar von Start- und Endposition aus der gleichen Sequenz stammt und in gültiger Reihenfolge (Start $<$ Ende) vorliegt. Ist dies der Fall, so wird, wie Abbildung 5.9 III zeigt, eine entsprechende Hypothese generiert und mit allen gültigen Positionspaaren verknüpft – diese können im Folgenden als *Belege* für eine Hypothese interpretiert und ggfs. zur Gewichtung von Hypothesen verwendet werden (siehe Abschnitt 5.4.2).

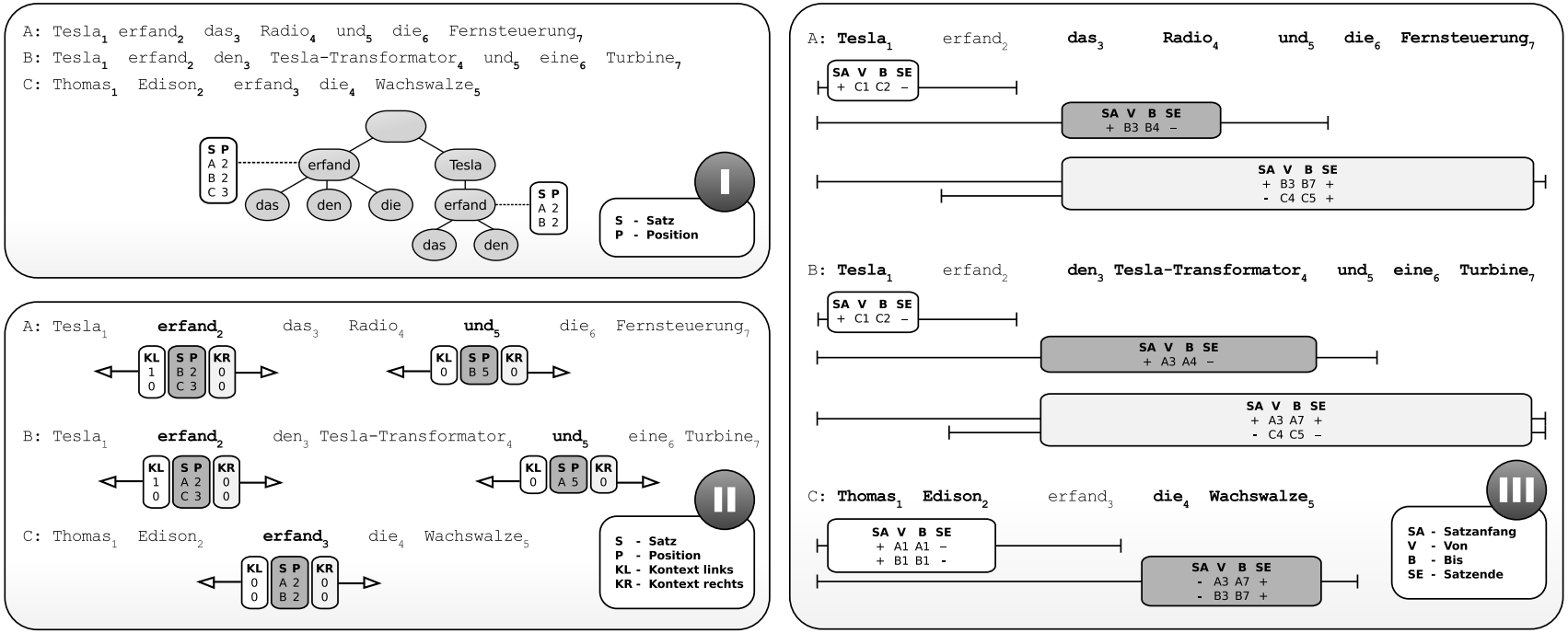

Abbildung 5.9: Generierung von Strukturhypothesen mit N-Gramm-Bäumen. Aus dem Ausgangstext wird zunächst ein N-Gramm-Baum erzeugt, dessen Knoten mit Positionsangaben verknüpft werden (I). Die *Boundaries Detector*-Komponente überträgt die in Suffix- und Präfixbaum enthaltenen Informationen auf den Text und markiert so potentielle Start- und Endpositionen von Hypothesen (II). Anhand dieser Annotationen werden durch den *Hypothesis Generator* strukturelle Auszeichnungen vorgenommen, in denen individuelle Belege, die für die Erzeugung der Strukturen verwendet wurden, durch Positionsangaben referenziert und mit Informationen zum gemeinsamen Kontext angereichert sind (III). Die finale Annotation ist kompatibel zu den Daten der ABL4J-Komponenten, ermöglicht jedoch zusätzlich den direkten Zugriff auf sämtliche Belege, die zur Generierung der Hypothese führten.

Dieses Vorgehen mag zunächst umständlich erscheinen – es bietet jedoch den Vorteil, dass nach Ausführung des *Boundaries Detector* sämtliche Kontextinformationen zu möglichen Strukturhypothesen lokal, d.h. pro Sequenz, vorliegen, so dass eine detaillierte Betrachtung der Indizien, die zur Bildung einer Hypothese führten, möglich wird. Zudem können die Datenstrukturen anschließend iterativ verarbeitet werden, ohne dass es notwendig ist, sämtliche Alignments im Arbeitsspeicher zu halten. Da die Erzeugung dieser Strukturen ebenfalls pro Sequenz durchgeführt werden kann, wird der Speicherbedarf des Verfahrens in erster Linie durch die zugrundeliegenden N-Gramm-Bäume bestimmt.

Anhand der drei Komponenten lassen sich einige der in Abschnitt 4.1.4 unter theoretischen Aspekten beschriebenen Möglichkeiten des *Tesla Role System* (TRS) veranschaulichen: So konnte durch das TRS bspw. die Rolle *NGram Tree* definiert werden. Die im DataObject spezifizierten Methoden vereinfachen u.a. die Traversierung derartiger Baumstrukturen und bieten gleichzeitig verschiedene Funktionen, die auch für die Anwendung in weiteren, nicht zwingend linguistischen Kontexten geeignet sind.[161] Da Datenstrukturen im TRS keinen Einschränkungen unterliegen, können die Bäume unmittelbar gespeichert und von anderen Komponenten weiterverarbeitet werden, ohne dass ihre komplexe interne Struktur in ein anderes Format konvertiert werden muss. Der *Hypothesis Generator* demonstriert, wie Vererbung im TRS eingesetzt werden kann: Die produzierte Rolle *Context-aware Hypotheses* ist von der von *ABL Align* produzierten Rolle abgeleitet und erweitert die generierten Datenstrukturen so, dass zu jeder Strukturhypothese Informationen über die Belegstellen, die für die Erzeugung der Hypothese verantwortlich waren, vorliegen (vgl. Abbildung 5.9 III). Auf diese Weise kann das in Abschnitt 2.2.2 anhand der Datenstrukturen von ABL4J beschriebene Problem der universellen Einsetzbarkeit von Datenstrukturen umgangen werden, ohne dass der Anspruch an Vergleichbarkeit aufgegeben werden muss. Abbildung 5.10 zeigt Integration und Interaktion der hier beschriebenen Komponenten in einem Tesla-Experiment.

5.4 Versuchsaufbau und Analyse

Das hier vorgestellte Experiment veranschaulicht die Unterstützung von einheitlicher Präund Postprozessierung durch Tesla ebenso wie die Austauschbarkeit von Komponenten: In jedem Experiment werden mehrere Instanzen der *ABL-Align*-Komponente verwendet, die identische Daten konsumieren und die auf identische Art und Weise evaluiert werden, die

[161] So wird bspw. in Hermes (2011, Abschnitt 6.3) diese Komponente genutzt, um sich wiederholende Muster in verschlüsselten Texten zu detektieren und daraus Rückschlüsse über Möglichkeiten zur Dechiffrierung des Manuskripts zu ziehen.

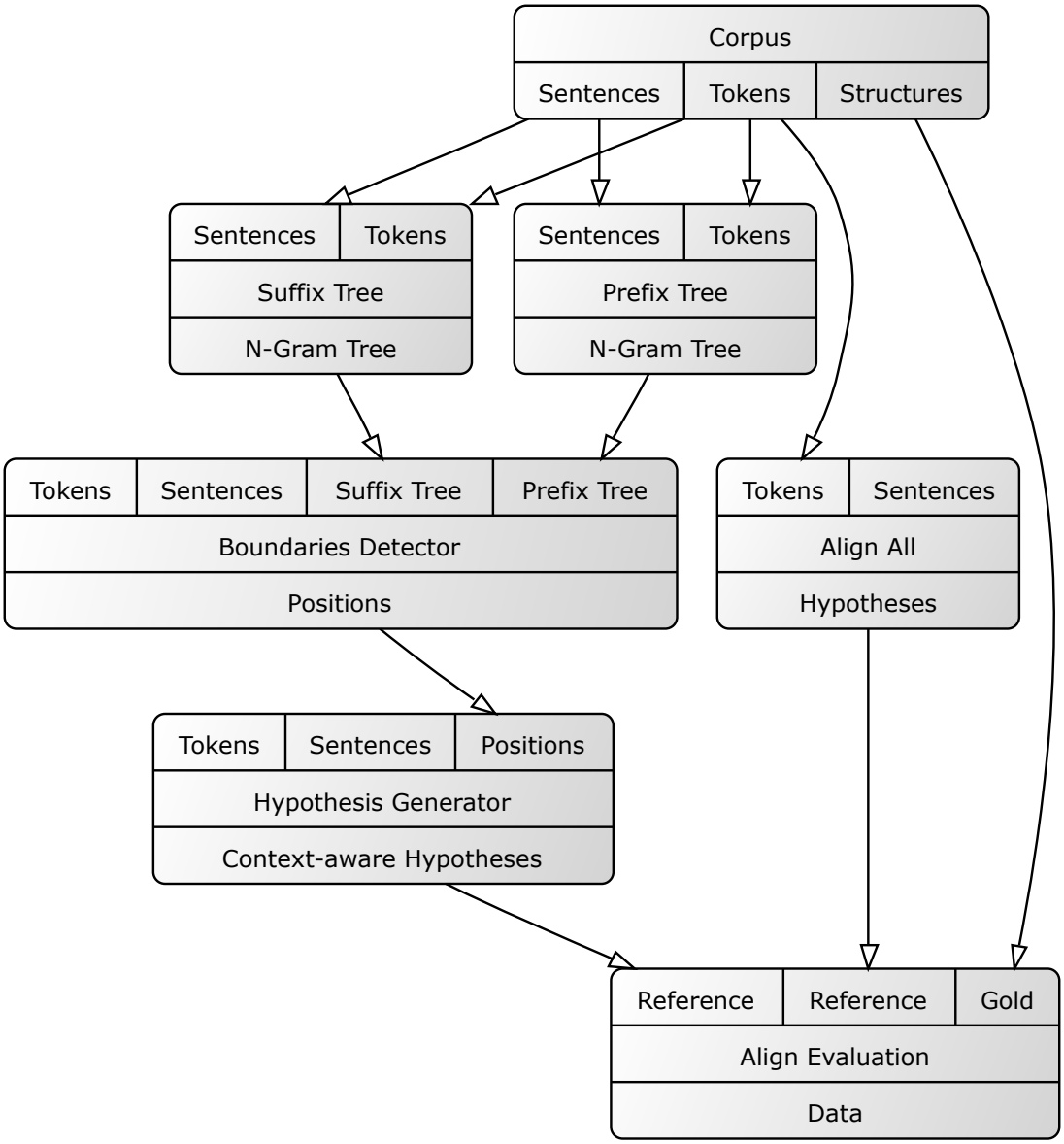

Abbildung 5.10: Versuchsaufbau zur Evaluation von Alignment-Verfahren. Die Komponenten *NGram Tree Generator*, *Boundaries Detector* und *Hypothesis Generator* bieten eine Prozessierungskette, deren Ergebnis kompatibel zur von *ABL Align* produzierten Rolle ist. Der *Hypothesis Generator* realisiert eine Subrolle, die zusätzliche Informationen über die generierten Hypothesen spezifiziert. *Align All* steht hier stellvertretend für alle evaluierten ABL-Align-Verfahren; die in Abbildung 5.5 gezeigten Komponenten zur Filterung der zu verarbeitenden Korpora werden hier zwecks Übersichtlichkeit (ebenso wie die zusätzliche Auswertung der Ergebnisse anhand des Berkeley Parsers) nicht dargestellt.

sich jedoch in ihrer Konfiguration unterscheiden und verschiedene Alignment-Verfahren verwenden. Zudem werden der Berkeley Parser und die im vorherigen Abschnitt beschriebene *Suffix*-Komponente evaluiert, die hinsichtlich der von ihnen produzierten Rollen kompatibel zu ABL sind und daher mit gleicher Postprozessierung ausgeführt werden können. Weiterhin wird für jedes Verfahren die in Abschnitt 5.2 vorgestellte *Random*-Komponente ausgeführt, die als weitere Vergleichsmöglichkeit herangezogen werden kann.

Alle Verfahren werden sowohl an den Strukturen im Gold-Korpus evaluiert als auch mit den vom Berkeley Parser detektierten Strukturen verglichen, wobei sämtliche Strukturen, wie in Abschnitt 5.1 beschrieben, von Interpunktionszeichen und Duplikaten bereinigt sowie bezüglich trivialer Strukturen gefiltert werden – das vollständige Experiment besteht aus fast 50 Komponenten.

Tabellen 5.4 und 5.5 fassen die Ergebnisse der Analyse des TüBa-D/S zusammen – der Aufbau des Experiments entspricht dabei der bereits in Abbildung 5.5 dargestellten Versuchsanordnung, die lediglich um zusätzliche Instanzen der Komponenten *ABL Align* und *Random* sowie der oben erwähnten *Suffix*-Komponenten erweitert wurde. Dabei ist anzumerken, dass in diesem Abschnitt lediglich die *Align*-Phase von ABL (vgl. Abschnitt 2.2) untersucht wird, in der von *All*, *WF-B* und *Suffix* widersprüchliche bzw. inkompatible Strukturen generiert werden können. Der Schwerpunkt der im folgenden Abschnitt 5.4.1 durchgeführten Analyse liegt auf dem der Recall der Verfahren, der in dieser Phase zwangsläufig[162] am höchsten ist. In Abschnitt 5.4.2 wird anschließend der F-Score der Verfahren untersucht.

5.4.1 Generierung syntagmatischer Hypothesen

Tabelle 5.4 fasst die Evaluation am Gold-Standard zusammen. Um eine bessere Beurteilung der Daten zu ermöglichen, wurde die in Abschnitt 5.2 vorgestellte *Random*-Komponente für jedes untersuchte Verfahren ausgeführt. Die Ergebnisse des *Random*-Verfahrens für *Left*, *Right* und *Berkeley Parser* sind jedoch zusammengefasst, da sie sich in keiner Untersuchung unterscheiden. Hinsichtlich *Left* und *Right* ist dies zwangsläufig der Fall, da beide Verfahren für einen aus n Wörtern bestehenden Satz stets $n-2$ Hypothesen generieren.[163] Dass auch das für den *Berkeley Parser* verwendete *Random*-Verfahren iden-

[162] Widersprüche, d.h. überlappende Strukturen, werden in der *Select*-Phase von ABL dadurch aufgelöst, dass einige der Strukturen entfernt werden (vgl. Abschnitt 5.4.2 für die Beschreibung unterschiedlicher Auswahlstrategien). Da bei diesem Schritt falsche Entscheidungen getroffen werden, führt dies zu einer Verringerung des Recalls.

[163] Dies erklärt auch die exakte Übereinstimmung der von *Left*, *Right* und *Random Left/Right* generierten Anzahl von Hypothesen: Die von *Random* vorgenommene Berechnung des Mittelwerts für Sätze der

	Hypothesen	Precision	Recall	F-Score
Gold	117.865	100	100	100
Gold Random	118.509	9,55	9,6	9,58
Berkeley	206.403	36,27	63,52	46,18
Left	210.105	4,11	7,33	5,27
Right	210.105	**22,14**	39,46	**28,36**
Random L/R/BP	210.105	9,43	16,81	12,08
WF-B	1.022.090	9,25	80,21	16,59
Random WF-B	1.019.749	8,43	72,89	15,1
All	1.211.620	8,69	89,39	15,85
Random All	1.212.222	8,11	83,45	14,79
Suffix	**1.253.541**	8,56	**91,02**	15,65
Suffix Random	1.252.679	8,04	85,43	14,69

Tabelle 5.4: Evaluation der untersuchten Verfahren anhand der im TüBa-D/S annotierten Strukturen. Für jede Spalte ist der höchste Wert, den eines der untersuchten Verfahren erzielt hat, hervorgehoben.

tische Ergebnisse liefert, ist hingegen darauf zurückzuführen, dass die durchschnittliche Anzahl von Hypothesen, die der Parser für einen Satz der Länge n generiert, in jedem betrachteten Fall nah an $n-2$ $(\pm 0,49)$ liegt, so dass rundungsbedingt gleiche Referenzwerte für *Random* erzeugt werden.

Die besten Ergebnisse werden durch das *Right*-Verfahren erzeugt, welches nicht nur einen im Vergleich zum entsprechenden *Random*-Verfahren deutlich besseren Recall liefert, sondern zudem auch eine Precision zwischen 14 (TüBa-D/Z) und 31 (TüBa-E/S) Prozent erreicht, die die Ergebnisse der anderen Verfahren um 5 (TüBa-D/Z) bis 22 (TüBa-E/S) Prozentpunkte übersteigt.[164] Dies ist darauf zurückzuführen, dass syntaktische Strukturen im Deutschen ebenso wie im Englischen i.d.R. rechtsverzweigend sind (bzw. in den untersuchten Korpora entsprechend ausgezeichnet wurden). Entsprechend liefert die *Left*-Methode hier keine guten Ergebnisse – Precision und Recall sind schlechter als die vom *Random*-Verfahren generierten Daten. Die Ergebnisse von *Left* und *Right* stimmen in diesem Punkt mit den in van Zaanen (2000a, Abschnitt 4.2) beschriebenen Ergebnissen bei der Analyse englischer Korpora überein und entsprechen auch den Beobachtungen von Klein & Manning (2002, Abschnitt 4).

Die höchsten Recall-Werte werden von den Alignment-Verfahren *All*, *Suffix* und *WF-B*

Länge n ergibt hier stets eine ganze Zahl, so dass weder auf- noch abgerundet werden muss.

[164] Für die vollständige Auswertung aller Experimente siehe Anhang A.1.

	Hypothesen	Precision	Recall	F-Score
Berkeley	206.403	100	100	100
Left	210.105	6,89	7,01	6,95
Right	210.105	**55,24**	56,23	**55,73**
Random L/R/BP	210.105	15,16	15,46	15,33
WF-B	1.022.090	17,69	87,58	29,43
Random WF-B	1.019.749	14,19	70,11	23,6
All	1.211.620	15,96	**93,67**	27,27
Random All	1.212.222	13,82	81,17	23,62
Suffix	**1.253.541**	15,58	94,6	26,75
Suffix Random	1.252.679	13,74	83,4	23,6

Tabelle 5.5: Evaluation der untersuchten Verfahren anhand der vom Berkeley Parser im TüBa-D/S detektierten Strukturen. Für jede Spalte ist der höchste Wert, den eines der untersuchten Verfahren erzielt hat, hervorgehoben.

erreicht. Dies ist jedoch insbesondere der hohen Produktivität der Verfahren (es werden fünf bis sechs mal mehr Hypothesen generiert, als Strukturen im Gold-Standard ausgezeichnet sind) geschuldet: Die ermittelten Recall-Werte liegen lediglich fünf bis sieben Prozentpunkte über den Ergebnissen der jeweiligen *Random*-Verfahren, die zudem eine ähnliche Precision erreichen.

In Tabelle 5.5 ist die Auswertung der Verfahren anhand des Berkeley Parsers zusammengefasst. Tendentiell entsprechen die Daten der Evaluation am Gold-Standard; so liefert das *Right*-Verfahren erneut die besten Ergebnisse, während *All*, *WF-B* und *Suffix* die höchsten Recall-Werte erzeugen. Im Unterschied zu Tabelle 5.4 sind die Ergebnisse jedoch deutlich besser – bspw. wird der F-Score des Right-Verfahrens nahezu verdoppelt. Der F-Score der übrigen Verfahren (mit Ausnahme von *Left*) steigt ebenfalls um über zehn Prozentpunkte, da sich sowohl Recall als auch Precision verbessern. Letzteres ist jedoch nicht verwunderlich, da die Anzahl der Referenzhypothesen im Vergleich zum Gold-Korpus deutlich höher liegt, zudem zeigen die von den *Random*-Verfahren erreichten Werte, dass die Verfahren *All*, *WF-B* und *Suffix* zwar signifikant bessere Ergebnisse produzieren, sich jedoch dennoch nur um 11 bis 17 Prozentpunkte von *Random* abheben.

Die Evaluation der Verfahren an TüBa-D/Z, TüBa-E/S und BNC liefert ähnliche Werte: Die Anwendung des *Right*-Verfahrens führt in jeder Analyse zum besten F-Score; *All*, *WF-B* und *Suffix* generieren als produktivste Verfahren den höchsten Recall. Allerdings unterscheiden sich die absoluten Ergebnisse teilweise deutlich: Wie die Überlegungen zu

	Hypothesen	Precision	Recall	F-Score
Berkeley	475.782	100	100	100
Left	475.948	6,59	6,6	6,6
Right	475.948	**38,65**	38,66	**38,65**
Random L/R/BP	475.948	11,88	11,89	11,89
WF-B	1.766.190	15,58	57,84	24,55
Random WF-B	1.766.182	10,86	40,3	17,1
All	2.374.955	14,12	**70,51**	23,53
Random All	2.375.019	10,51	52,47	17,51
Suffix	**2.469.203**	13,84	71,8	23,2
Suffix Random	2.468.898	10,48	54,38	17,57

Tabelle 5.6: Evaluation der untersuchten Verfahren anhand der vom Berkeley Parser im TüBa-D/Z detektierten Strukturen. Für jede Spalte ist der höchste Wert, den eines der untersuchten Verfahren erzielt hat, hervorgehoben.

Satzlänge und Anzahl möglicher Hypothesen in Abschnitt 5.1 und 5.2 vermuten ließen, sind die Resultate bei Anwendung der Verfahren auf schriftsprachliche Korpora schlechter. Tabelle 5.6 veranschaulicht dies exemplarisch für das TüBa-D/Z mit Evaluation am Berkeley Parser, für die weiteren tabellarischen Auswertungen sei auf Anhang A.1 verwiesen.

Der auffälligste Unterschied (im Vergleich von TüBa-D/Z gegenüber TüBa-D/S) betrifft das *Right*-Verfahren, das bei allen ermittelten Werten fast 20 Prozentpunkte verliert. Die Verfahren *All*, *WF-B* und *Suffix* erreichen hingegen eine nur um wenige Prozentpunkte abweichende Precision, auch der Recall dieser Verfahren nimmt deutlich schwächer ab. Im Vergleich mit den *Random*-Verfahren verbessert sich der Recall sogar – im Gegensatz zu *Right* sind die untersuchten Verfahren theoretisch dazu in der Lage, auch in Sätzen mit komplexer syntaktischer Struktur (und unabhängig von der Position im Satz) korrekte Hypothesen zu detektieren.

Insgesamt zeigen die Ergebnisse jedoch, dass die untersuchten Methoden bei den verwendeten Korpora (und ohne weitere Präprozessierung) keine Daten generieren, die den Einsatz als ein Verfahren zur Strukturaufdeckung rechtfertigen könnten, bzw. die darauf hinweisen, dass eine Integration in eine praktisch einsetzbare Anwendung sinnvoll wäre. Dies muss jedoch nicht zwangsläufig an den Verfahren liegen, sondern könnte auch auf die Eigenschaften der Korpora zurückgeführt werden: So könnte etwa argumentiert werden, dass das Vokabular zu groß bzw. die Type/Token-Relation zu niedrig ist (vgl. Tabelle 5.1), und dass die Ergebnisse verbessert werden können, wenn die in den analysierten Korpora

	Left	Right	WF-B	All	Suffix
Left	–	0	95,75	97,33	97,81
Right	0	–	97,7	98,68	99,06
WF-B	19,68	20,08	–	98,93	99,75
All	16,88	17,11	83,46	–	99,41
Suffix	16,39	16,6	81,34	96,09	–

Tabelle 5.7: Auswertung der von den untersuchten Verfahren generierten Strukturen untereinander nach Anwendung auf das TüBa-D/S: Jeder Wert entspricht der Precision, die das in der Zeile genannte Verfahren bzgl. des in der jeweiligen Spalte genannten Verfahrens hat. So werden bspw. 16,88 Prozent der von *All* erzeugten Strukturen vom *Left*-Verfahren und 83,46 Prozent von *WF-B* generiert. Wird die Tabelle in umgekehrter Relation interpretiert, lässt sich der Recall von zwei Verfahren ablesen – bspw. sind die von *Left* produzierten Hypothesen zu 97,33 Prozent Bestandteil der von *All* erzeugten Strukturen.

enthaltenen Daten entsprechend aufbereitet würden. In Abschnitt 5.4.4 wird diese Überlegung erneut aufgegriffen; zunächst bleibt jedoch zu überprüfen, wie sich die verschiedenen Verfahren qualitativ unterscheiden, da bisher ausschließlich quantitativ untersucht wurde, wie hoch Precision und Recall der generierten Daten bezüglich der Referenzstrukturen sind. In Tabelle 5.7 ist daher (exemplarisch anhand des TüBa-D/S) angegeben, wie hoch der Anteil der Strukturen des in einer Zeile aufgeführten Verfahrens bezüglich der in den Spalten genannten Verfahren ist.

Die Daten in Tabelle 5.7 zeigen, dass ein Großteil der von den Verfahren generierten Strukturen identisch ist[165]: Nahezu alle der von *Left* und *Right* detektierten Strukturen werden auch von *All* und *WF-B* extrahiert, in Bezug auf das *Suffix*-Verfahren sind es ca. 90 Prozent. Dies kann erneut auf die im Vergleich zu *Left* und *Right* deutlich höhere Produktivität der Verfahren zurückgeführt werden. Bezüglich der drei Alignment-Verfahren untereinander bietet sich ein ähnliches Bild: So werden fast alle Strukturen auch von *All* und *Suffix* generiert, lediglich *WF-B* zeigt hier eine Abweichung.

Damit genügt es grundsätzlich, in den weiteren Untersuchungen lediglich eines der hier verwendeten Verfahren zu nutzen, was sowohl einen pragmatischen als auch einen theoretischen Vorteil mit sich bringen würde: Die Laufzeit des *Suffix*-Verfahrens ist deutlich besser als die von *WF-B* und *All* (vgl. Abschnitt 5.3), zudem handelt es sich bei der von *Suffix* produzierten Rolle um eine Erweiterung der von ABL4J angebotenen Rolle, die (wie in

[165] Dies gilt nicht für *Left* und *Right* untereinander – werden triviale Strukturen nicht berücksichtigt, sind die von den Verfahren generierten Strukturen zwangsläufig schnittmengenfrei.

Abbildung 5.9 gezeigt) eine detailliertere Analyse der Strukturhypothesen ermöglicht. Da das im nächsten Abschnitt untersuchte *Select*-Verfahren von diesen Analysemöglichkeiten keinen Gebrauch macht, wird auf die Evaluation von *WF-B* und *All* nicht vollständig verzichtet – es wird allerdings davon abgesehen, sie zur Analyse der Zeitungskorpora zu verwenden.

5.4.2 Filterung syntagmatischer Hypothesen mit *ABL Select*

Wie die im vorherigen Abschnitt aufgeführten Ergebnisse zeigen, handelt es sich bei den Alignment-Verfahren *WF-B*, *All* und *Suffix* um stark übergenerierende Ansätze, die deutlich mehr Hypothesen generieren als im Gold-Standard vorhanden sind bzw. vom *Berkeley Parser* detektiert werden (vgl. Abbildung 5.6). Diese Hypothesen sind zudem teilweise widersprüchlich (vgl. Abschnitt 2.2) – in diesem Abschnitt werden daher Verfahren untersucht, mit denen Inkompatibilitäten von Strukturhypothesen (d.h. sich überlappende und somit widersprüchliche Strukturen, vgl. Abschnitt 2.2.1) aufgelöst und die Kardinalität der Hypothesenmenge reduziert werden können.

In *ABL4J* stehen dafür drei verschiedene *Select*-Verfahren zur Verfügung, die sich in ihrer Strategie zur Auflösung von Widersprüchen unterscheiden: *First* verfolgt die Annahme, dass eine früh gelernte Hypothese gegenüber später generierten Hypothesen zu bevorzugen ist. *Constituents* und *Terms* (von van Zaanen oftmals auch als *Branch* und *Leaf* bezeichnet) verwenden hingegen quantitative Bewertungsverfahren: Für zwei inkompatible Hypothesen wird ermittelt, wie häufig die in den Strukturen enthaltenen Wörter als Hypothesen markiert wurden. Beim *Terms*-Verfahren wird dieser Wert anschließend durch die Anzahl aller Hypothesen geteilt, während das *Constituent*-Verfahren lediglich diejenigen Hypothesen berücksichtigt, die der gleichen Kategorie (vgl. Abschnitt 2.4)zugeordnet wurden.

In der hier durchgeführten Untersuchung wird ausschließlich das *Terms*- bzw. *Leaf*-Verfahren analysiert: Die untersuchten Korpora bieten keinen Anlass dafür, die *First* zugrundeliegende Annahme zu verfolgen, da die enthaltenen Sätze nicht hinsichtlich syntaktischer Komplexität o.ä. sortiert sind[166]. Das *Constituents*-Verfahren wird u.a. deshalb

[166] van Zaanen (2002, Abschnitt 4.2.1) räumt ebenfalls ein, dass die zugrundeliegende Annahme nur wenig plausibel ist, sofern nicht speziell angepasste Korpora verwendet werden:

> The assumption that hypotheses learned earlier are correct may perhaps be only likely for human language learning. It so happens that the type of sentences in a corpus are generally not comparable to the type of sentences human hear when they are learning a language. Futhermore, the implication that once an incorrect hypothesis has been learned, it can never be corrected, is (cognitively) highly implausible.

nicht weiter untersucht, weil es gegenüber *Terms* deutlich schlechtere Ergebnisse liefert, wie auch van Zaanen (2002, Abschnitt 5.1.3) beschreibt und den Grund auf zwei Faktoren zurückführt:

> The hypothesis universe [d.h. die Menge der Hypothesen] contains many correct, but also incorrect hypotheses which are used in the computation of the probabilities. It may be the case that when using more precise statistics, the incorrect hypotheses have a larger impact on the final probability, yielding worse results. Apart from this, [...the branch select method] relies on the non-terminal types of the hypotheses. However, the types are clustered in an imperfect way [...] which introduces an extra margin of error.

Da das *Constituents*-Verfahren zudem nicht adäquat mit einem zufallsbasierten Verfahren verglichen werden kann (den von *Random* generierten Hypothesen können keine Kategoriebezeichnungen zugewiesen werden, so dass eine Auswertung von *Constituents* nicht möglich ist), wird von einer weiteren Untersuchung abgesehen.

Abbildung 5.11 zeigt den Versuchsaufbau (einschließlich der im folgenden Abschnitt vorgestellten *Suffix Select*-Komponente), während die Tabellen 5.8 und 5.9 die Auswertung der für das TüBa-D/S generierten Strukturen zusammenfassen. Um eine genauere Interpretation der Daten zu ermöglichen, werden hier zwei zufallsbasierte Verfahren eingesetzt: Zum einen wird das bereits im vorherigen Abschnitt verwendete *Random*-Verfahren genutzt, um zu untersuchen, wie sich das *Select*-Verfahren verhält, wenn es eine zufällig erzeugte Menge teilweise widersprüchlicher Strukturen verarbeitet – die Ergebnisse werden für jedes untersuchte Alignment-Verfahren aufgeführt und als *Select Random* bezeichnet. Zum anderen wird ermittelt, welche Ergebnisse ein Verfahren erzielt, dass sich lediglich an der Produktivität eines Select-Verfahrens orientiert (d.h. die Menge der zu generierenden Hypothesen für jede Satzlänge aus dem Select-Verfahren übernimmt), daber aber dennoch zufallsbasiert widerspruchsfreie Hypothesen erzeugt – dieses Vergleichsmaß wird ausschließlich für das *Select Suffix*-Verfahren berechnet und als *Random Non-Crossing* in den tabellarischen Zusammenfassungen aufgeführt.

Der wesentliche Unterschied zwischen den beiden zufallsbasierten Verfahren besteht darin, dass *Select Random* Vergleichswerte bereitstellt, anhand derer evaluiert werden kann, wie gut die Hypothesenmenge, die durch ein Align-Verfahren generiert wurde, von *Select* bewertet werden kann, während es die von *Random Non-Crossing* erzeugten Re-

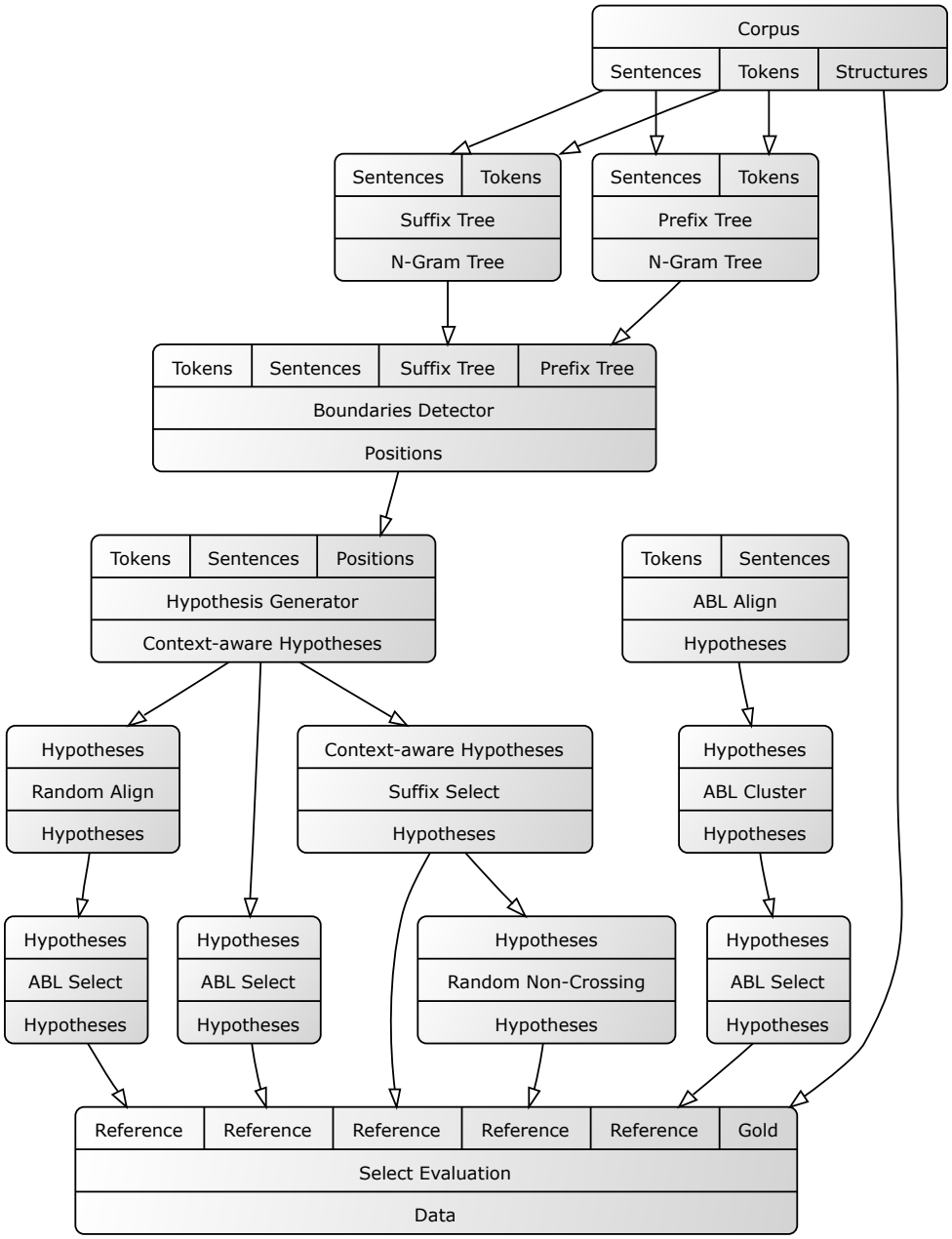

Abbildung 5.11: Schematischer Versuchsaufbau zur Analyse verschiedener *Select*-Verfahren in Tesla. Neben *ABL Select* wird auch das *Suffix Select*-Verfahren untersucht, welches Zusatzinformationen über Kontextlänge und Fundstellen von Hypothesen, die durch die Rolle *Context-aware Hypotheses* zur Verfügung gestellt werden, verwendet. Da es sich hierbei um eine Subrolle der von *ABL Select* konsumierten Rolle handelt, kann dieses Verfahren ebenfalls zur Selektion der vom *Hypothesis Generator* erzeugten Strukturen genutzt werden.

	Hypothesen	Precision	Recall	F-Score
Gold	117.865	100	100	100
Berkeley	206.403	36,27	63,52	46,18
Unique Right	210.105	22,14	39,46	28,36
Select WF-B	109.495	**23,88**	22,19	23
Select Random WF-B	104.404	22,04	19,52	20,71
Select All	128.651	23,41	25,55	24,43
Select Random All	134.772	21,83	24,95	23,28
Select Suffix	137.234	23,06	**26,85**	**24,81**
Select Random Suffix	142.007	21,72	26,17	23,73
Random Non-Cross.	137.259	12,78	14,88	13,75

Tabelle 5.8: Evaluation der Verfahren *WF-B*, *All* und *Suffix* nach der Select-Phase. Als Referenz dienen die manuell annotierten Strukturen im TüBa-D/S.

sultate ermöglichen, die Qualität des *Select*-Verfahrens im Kontrast zu einer zufälligen, nicht überlappenden Strukturierung zu bewerten.

Der Vergleich der von *Align* und *Select* erzielten Ergebnisse bei der Evaluation anhand der im TüBa-D/S ausgezeichneten Strukturen (Tabellen 5.4 und 5.8) zeigt, dass der F-Score durch das *Terms*-Bewertungsverfahren um sechs bis acht Prozentpunkte verbessert werden konnte – zudem ist die durch *Select* ausgewählte Hypothesenmenge widerspruchsfrei; sie könnte also als dazu verwendet werden, eine eindeutige Repräsentation der internen Satzstruktur aufzubauen, die – unter technischen Gesichtspunkten – äquivalent zur Analyse eines Parsers ist, was in Form einer gemeinsamen (Basis-) Rolle im TRS abgebildet werden könnte. Für eine aus praktischer Sicht sinnvolle Umsetzung ist der erzielte F-Score jedoch weiterhin zu gering, auch wenn er mit ca. 22 bis 24 Prozent deutlich über dem von *Random Non-Crossing* erreichten Ergebnis liegt.

Wie Tabelle 5.8 jedoch auch zeigt, weichen die Ergebnisse des *Select*-Verfahrens nach Anwendung auf die Hypothesen eines *Align*-Verfahrens nur unwesentlich von den Ergebnissen ab, die *Select* für zufällig erzeugte Strukturen generiert – die Tabellen in Anhang A.2 verdeutlichen, dass dies auch bei Anwendung der Verfahren auf die übrigen Korpora der Fall ist. Die oben zitierte Annahme van Zaanens, dass die Übergenerierung während der Align-Phase dazu führt, dass in der *Select*-Phase zu viele falsche Hypothesen berücksichtigt werden, kann auch hier als mögliche Erklärung der Ergebnisse herangezogen werden – wie in Abschnitt 5.4.1 gezeigt werden konnte, beträgt der Unterschied zwischen dem F-Score von Alignment- und zufallsbasiertem Verfahren teilweise weniger als 2 Pro-

	Hypothesen	Precision	Recall	F-Score
Berkeley	206.403	100	100	100
Right	210.105	55,24	56,23	55,73
Select WF-B	109.495	31,77	16,86	22,03
Select Random WF-B	104.404	25,75	13,02	17,3
Select All	128.651	30,09	18,76	23,11
Select Random All	134.712	26,06	17,01	20,58
Select Suffix	137.234	29,69	19,74	23,72
Select Random Suffix	142.047	26,11	17,97	21,29
Random Non-Cross.	137.259	20,95	13,93	16,74

Tabelle 5.9: Evaluation der Verfahren *WF-B*, *All* und *Suffix* nach der *Select*-Phase. Als Referenz dienen die vom Berkeley Parser erzeugten Strukturen im TüBa-D/S.

zentpunkte (etwa bei Evaluation anhand der in TüBa-D/S und TüBa-E/S enthaltenen Strukturen, vgl. Tabelle A.1 und A.3 in Anhang A.1).

Weiterhin fällt auf, dass die durch *Select* erzeugten Hypothesenmengen deutlich weniger Strukturen enthalten, als bspw. vom *Berkeley Parser* generiert wurden – bei Auswertung mit Hilfe des Parsers (Tabelle 5.9) ergibt sich daraus zwangsläufig ein deutlich schlechterer Recall. Im folgenden Abschnitt wird daher ein alternatives *Select*-Verfahren untersucht, das eine Auswahl von Hypothesen anhand einer Kontextanalyse und -bewertung vornimmt, während gleichzeitig versucht wird, die Übergenerierung von Hypothesen einzuschränken.

5.4.3 Filterung syntagmatischer Hypothesen mit N-Gramm-Bäumen

Konstruktionsbedingt bietet das *Suffix*-Verfahren gegenüber *WF-B* und *All* neben dem bereits erwähnten deutlich besseren Laufzeitverhalten einen weiteren wesentlichen Vorteil, denn aus den zugrundeliegenden N-Gramm-Bäumen lässt sich unmittelbar die Breite des gemeinsamen Kontextes vor und nach einer Hypothese ablesen. Wie anhand der Abbildungen 5.7 und 5.8 erläutert, markieren Knoten mit mehr als einem Kindknoten mögliche Start- bzw. Endpositionen einer Strukturhypothese. Die Tiefe des Knotens entspricht dabei der Länge des Kontextes: So kann die *Boundaries Detector*-Komponente bei der Übertragung der in den Bäumen enthaltenen Daten auf die zugrundeliegenden Sätze diese Information ebenfalls speichern, wodurch dem *Hypothesis Generator* eine Möglichkeit zur Bewertung von Hypothesen zur Verfügung steht, die den in ABL4J verwendeten *Select*-Verfahren nicht gegeben ist (vgl. Abbildung 5.9). Die Tiefe der Knoten kann grundsätzlich

bereits bei der Ausführung des *Boundaries Detector* dazu genutzt werden, die Kardinalität der Hypothesenmenge einzuschränken, indem eine minimale Tiefe im N-Gramm Baum (und damit eine minimale Kontextbreite) spezifiziert wird, die für die Erzeugung einer Hypothese erreicht werden muss. Damit lässt sich ein grundsätzliches Problem der bisher untersuchten Ansätze umgehen: *WF-B* und *All* generieren bereits dann Hypothesen, wenn zwei einzelne Wörter in gleicher Reihenfolge in zwei oder mehr Sätzen vorhanden sind. Wie in Abbildung 5.4 dargestellt, folgt die Verteilung der Worthäufigkeiten jedoch annähernd dem Zipf'schen Gesetz: Die vier am häufigsten im TüBa-D/S vorkommenden Wörter (in absteigender Reihenfolge *ja*, *ich*, *das* und *wir*) sind durchschnittlich etwa 10.000 mal in den 38.000 Sätzen des Korpus enthalten. Ein Großteil der generierten Hypothesen kann daher auf hochfrequente Wörter zurückgeführt werden[167], die zudem – wie das Wort *ja* zeigt – nur bedingt für die Detektion syntaktischer Strukturen geeignet sind. Dies erklärt auch, weshalb bei der in Abschnitt 5.1 beschriebenen Präprozessierung eine Filterung von Satzzeichen vorgenommen wurde: So enthalten die 38.000 im TüBa-D/S enthaltenen Sätze neben den Satzendzeichen ungefähr 18.000 weitere Interpunktionssymbole, die ohne Filterung zusätzlich zur Übergenerierung beitragen würden.

Wie bereits in Abschnitt 5.3.1 beschrieben, können *Boundaries Detector* und *Hypothesis Generator* so konfiguriert werden, dass für zwei Sequenzen nur dann eine Hypothese erzeugt wird, wenn der gemeinsame Kontext aus zwei oder mehr Wörtern besteht – statt Einzelwörtern sind in diesem Fall Bigramme (bzw. beliebig lange N-Gramme) Grundlage

[167] Analysiert ein Alignment-Verfahren die Sequenz $\ldots x s_1 \ldots s_n y \ldots$ und generiert die Hypothese, dass es sich bei $s_1 \ldots s_n$ um eine Struktur handelt, dann setzt dies voraus, dass die Symbolfolge $x \ldots y$ mindestens ein weiteres Mal im Korpus enthalten sein muss. Die Wahrscheinlichkeit, dass zwei Symbole x und y, die in einer Sequenz in linearer Folge auftauchen, ein weiteres Mal in den restlichen Sätzen gefunden werden können, sei hier vereinfacht definiert als

$$P(x,y) = \frac{P(x) \cdot P(y)}{2} \tag{5.3}$$

$P(x)$ und $P(y)$ repräsentieren dabei die unbedingte Wahrscheinlichkeit dafür, dass eine Sequenz das Symbol x bzw. y enthält – diese lässt sich für jedes Symbol i durch n_i/n_s berechnen, wobei n_i die Anzahl der Vorkommen des Symbols i und n_s die Anzahl der untersuchten Sequenzen darstellen. Die Definition von $P(x,y)$ basiert auf der Annahme, dass die Vorkommen der Symbole voneinander unabhängig sind (was eine Vereinfachung darstellt, da dies für natürliche Sprache nicht zutrifft), und in der Hälfte der Fälle die lineare Reihenfolge der Symbole falsch ist (d.h. das Symbol y vor dem Symbol x vorkommt und daher von einem Alignment-Verfahren nicht berücksichtigt würde). Für die vier am häufigsten im TüBa-D/S vorkommenden Wörter gilt also $P(x,y) = (\frac{10}{38} \cdot \frac{10}{38})/2 \approx 0,035$ – dies entspricht der Wahrscheinlichkeit, dass ein beliebiger Satz die Symbolfolge $x \ldots y$ enthält, so dass bei 38.000 Sätzen erwartet werden kann, dass die Symbolfolge etwa 1.300 mal auftaucht. Die genannten Wörter wären somit für etwa $2^4 \cdot 1.300 = 20.800$ Hypothesen verantwortlich, was trotz Vereinfachung relativ nah am empirisch ermittelten Ergebnis (fast 18.400 Hypothesen, siehe auch Anhang C.4) liegt.

der Hypothese. Dies führt zu einer starken Verkleinerung der Hypothesenmenge, wobei allerdings nicht nur falsche, sondern auch korrekte Hypothesen verworfen werden.

Da in den vom *Hypothesis Generator* erzeugten Hypothesen die Information darüber, welche Kontexte bei der Generierung berücksichtigt wurden, vorhanden ist, kann die Bewertung von Hypothesen auch von einer eigenständigen Komponente durchgeführt werden. Dies wurde mit einer als *Suffix Select* bezeichneten Komponente umgesetzt: Für die Selektion inkompatibler Hypothesen können hier unterschiedliche Bewertungsverfahren für individuelle Hypothesen mit Verfahren zur satzbezogenen Bewertung und Auswahl von Hypothesenmengen kombiniert werden – beide Modifikationen werden durch Implementation eines Interfaces durchgeführt, das in Teslas Experiment-Editor ausgewählt werden kann (analog zu der in Abschnitt 4.1.5.2 beschriebenen Vorgehensweise).

Zur Bewertung einer einzelnen Hypothese werden unterschiedliche Implementationen des Interfaces `IWeightCalculator` verwendet: Die als *Width* bezeichnete Variante gewichtet Hypothesen anhand der maximalen Kontextbreite, die gefunden werden konnte, während *Width & Occurrences* (W/O) die Kontextbreite W mit der Anzahl Alignments dieser Breite O durch die Formel $2^W log_e(O)$ kombiniert[168]. *W/O (Best)* berücksichtigt hierbei lediglich die Alignments mit maximaler Kontextbreite, während *W/O (All)* die Summe über alle Alignments berechnet. Die Verfahren nutzen nur einen Teil der verfügbaren Informationen[169], sie eignen sich aber dennoch dazu, den oben beschriebenen Einfluss hochfrequenter Wörter bei der Auswahl widersprüchlicher Hypothesen zu berücksichtigen. Zudem ist festzuhalten, dass die Grundlage der Bewertung deutlich von den im vorherigen Abschnitt beschriebenen Verfahren abweicht, denn anders als in den von van Zaanen entwickelten Verfahren wird die Bewertung hier nicht anhand der Häufigkeit der in einer Hypothese enthaltenen Wortfolge ermittelt, sondern ausschließlich auf Basis der Belege, die zur Erzeugung der Hypothese führten – das Verfahren ist somit deutlich stärker an die in Abschnitt 2.1.1 beschriebene Diskursanalyse nach Harris angelehnt.

Hinsichtlich des Auswahlverfahrens, mit dem eine widerspruchsfreie Teilmenge aus allen potentiellen Hypothesen eines Satzes gebildet wird, findet hier keine Variation statt: Der verwendete Algorithmus berechnet für jede Teilmenge die Summe der Einzelbewertungen

[168] Durch die hier verwendete Formel wird erreicht, dass die W und O mit unterschiedlicher Gewichtung in die Bewertung einfließen: Die Kontextbreite wird im Verhältnis zur Anzahl von Fundstellen deutlich stärker berücksichtigt. Ob Variationen dieser Bewertungsfunktion die Ergebnisse positiv beeinflussen können, ist in separaten Untersuchungen zu evaluieren – dank der Verwendung des `IWeightCalculator`-Interfaces ist der dafür erforderliche Entwicklungsaufwand vergleichsweise gering.

[169] Nicht berücksichtigt wird bspw. die Häufigkeit, mit der ein N-Gramm, das den linken oder rechten Kontext einer Hypothese definiert, im Korpus vorkommt – derartige Optimierungen könnten die Bewertung von Hypothesen u.U. weiter verbessern.

	Hypothesen	Precision	Recall	F-Score
Gold	117.865	100	100	100
Right	210.105	22,14	39,46	28,36
Select Width	**184.780**	17,54	27,51	21,42
Select W/O (Best)	173.582	20,77	**30,59**	**24,74**
Select W/O (All)	178.557	19,74	29,9	23,78
Select (Terms)	75.770	**26,09**	16,77	20,42
Random N.C. (Best)	175.809	12,06	17,99	14,44
Random N.C. (Terms)	76.064	13,75	8,87	10,79

Tabelle 5.10: Evaluation des *Suffix Select*-Verfahrens anhand der im TüBa-D/S ausgezeichneten Strukturen.

und wählt die Menge aus, bei der das beste Ergebnis erzielt werden konnte.[170] Dies führt u.a. dazu, dass das Verfahren mehr Hypothesen auswählt als das zuvor verwendete *Terms Select* und somit (bei Vergleich mit den vom *Berkeley Parser* ausgezeichneten Strukturen) prinzipiell auch einen höheren Recall erreichen kann.

In den Tabellen 5.10 und 5.11 sind die Ergebnisse unterschiedlicher Konfigurationen der Komponente zusammengefasst – der *Boundaries Detector* ist in dem untersuchten Experiment so konfiguriert, dass Bigramme als Grundlage für die Erzeugung von Hypothesen verwendet werden, so dass eine Generierung von Hypothesen ausschließlich auf Basis hochfrequenter Einzelwörter ausgeschlossen wird. Auf eine Anwendung der Select-Verfahren auf zufällig erzeugte Hypothesen muss hier verzichtet werden, da *Suffix Select* für die Bewertung einer Hypothese Informationen über Häufigkeit und Breite identischer Kontexte benötigt, diese jedoch vom Random-Verfahren nicht zur Verfügung gestellt werden können. Das im vorherigen Abschnitt verwendete *Random Non-Crossing* kann hingegen auch hier als Vergleichsverfahren genutzt werden, zudem wird auch das *Terms Select*-Verfahren erneut eingesetzt – so kann gleichzeitig analysiert werden, wie sich die Einschränkung der minimalen Kontextbreite auf dieses Verfahren auswirkt.

Während die Evaluation an den manuellen Auszeichnungen im TüBa-D/S hinsichtlich des F-Scores keine wesentlichen Veränderungen zeigt, ergibt sich bei einem Vergleich

[170] Die Laufzeit des Verfahrens hängt stark von der Anzahl der Hypothesen, die für einen Satz generiert wurden, ab. Daher wurden zwei weitere Methoden implementiert, die heuristische Strategien verfolgen, um mit effizienterem Laufzeitverhalten eine akzeptable Lösung zu generieren; weitere Verfahren können durch Implementation des Interfaces `ISelectAlgorithm` umgesetzt und der Komponente zur Verfügung gestellt werden. Da in den hier durchgeführten Untersuchungen die Kardinalität der Hypothesenmenge jedoch (durch Festlegung der maximalen Satzlänge auf 30 Wörter) noch in akzeptabler Zeit verarbeitet werden kann, wurde auf die Verwendung der heuristischen Verfahren verzichtet.

	Hypothesen	Precision	Recall	F-Score
Berkeley	206.403	100	100	100
Right	210.105	55,24	56,23	55,73
Select Width	**184.780**	37,74	33,79	35,66
Select W/O (Best)	173.582	**47,1**	**39,61**	**43,03**
Select W/O (All)	178.557	44,88	38,83	41,63
Select (Terms)	75.770	45,76	16,8	24,57
Random N.C. (Best)	175.809	19,51	16,62	17,95
Random N.C. (Terms)	76.064	22,75	8,38	12,25

Tabelle 5.11: Evaluation des *Suffix Select*-Verfahrens anhand der vom Berkeley Parser im TüBa-D/S detektierten Strukturen.

mit den vom Berkeley Parser generierten Strukturen in Tabelle 5.11 ein anderes Bild: Mit Ausnahme von *Terms* erreichen alle Verfahren deutlich bessere Ergebnisse als zuvor, auch wenn diese weiterhin unterhalb der von *Right* definierten Schranke bleiben. Bei Betrachtung des *Terms*-Verfahrens fällt auf, dass die Precision durch Beschränkung der minimalen Kontextbreite stark verbessert werden konnte – das vergleichsweise schlechte Ergebnis ist erneut darauf zurückzuführen, dass das Verfahren nicht genügend Strukturen wählt, um annähernd die Produktivität des *Berkley Parsers* zu erreichen.

Die Anwendung des Verfahrens auf weitere Korpora (vgl. Anhang A.2) zeigt, dass keine generelle Verbesserung erreicht werden kann: Beim TüBa-D/Z (vgl. Tabellen A.19 und A.20 in Anhang A.2) liegt der F-Score zwischen 9,5 und 11,5 (Gold) bzw. 16,8 und 19,2 (*Berkeley Parser*) und damit weiterhin deutlich unter dem durch *Right* erreichten Wert (17,25 bzw. 38,65). Der F-Score der drei kontextbezogenen Verfahren unterscheidet sich bei den meisten Korpora nur unwesentlich, so dass eine Beurteilung der Bewertungsfunktionen nicht möglich ist, ohne weitere Untersuchungen durchzuführen. Auffällig ist hingegen, dass die Precision des *Terms*-Verfahrens bei allen Analysen vergleichsweise hoch liegt und die der kontextbezogenen Verfahren teilweise deutlich übersteigt – bei Anwendung auf das TüBa-E/S liegt die Precision mit 35 bzw. 49 Prozent bis zu 10 Prozentpunkte höher als die der übrigen Verfahren und sogar über dem von *Right* erreichten Wert (vgl. Tabellen A.17 und A.18 in Anhang A.3.2).

Eine weitere Vergrößerung der minimalen Kontextbreite zur Verbesserung der Qualität erweist sich als ungeeignet für die hier analysierten Korpora, wie exemplarisch für das TüBa-D/S untersucht wurde: Bereits bei Beschränkung auf Strukturen mit 3 gemeinsamen Wörtern im linken und rechten Kontext wurden von den in Tabelle 5.10 und 5.11

aufgeführten Verfahren weniger als 130.000 Hypothesen generiert. Obwohl die Precision von *Suffix Select* auf bis zu 23 (Gold) bzw. 54 Prozent (Berkeley Parser) stieg, konnte der F-Score daher nicht verbessert werden.

Die u.a. in van Zaanen & Geertzen (2008) festgehaltenen Ergebnisse können bisher somit nur teilweise bestätigt werden: Der dort angegebene F-Score von fast 47 kann hier nur für das TüBa-D/S (annähernd) erreicht werden. Die Ergebnisse legen die Vermutung nahe, dass der Erfolg Alignment-basierter Strukturaufdeckungsverfahren stark von der Komplexität der syntaktischen Struktur der in den analysierten Korpora enthaltenen Sätze abhängt.[171] Diese Beobachtung mag zunächst trivial erscheinen und nicht die Schlussfolgerung zulassen, dass nicht die untersuchten Verfahren, sondern die untersuchten Korpora angepasst werden sollen: Dies würde dem allgemein gehaltenen Anspruch der in Abschnitt 2.1 vorgestellten Überlegungen widersprechen. Andererseits erscheint es wenig plausibel, die Hypothese aufzustellen, dass annähernd fehlerfreie syntaktische Strukturen ausschließlich durch Analyse von Zeitungstexten oder Transkriptionen von Telefonaten zur Terminvereinbarung erlernt werden können. Die kognitive Plausibilität der hier untersuchten Verfahren wurde bisher nicht diskutiert, und es würde den Rahmen dieser Arbeit sprengen, wenn der Frage, ob bzw. inwieweit Alignment-Verfahren einen Teil des menschlichen Spracherwerbsprozesses abbilden, adäquat nachgegangen würde. Stattdessen sei hier stellvertretend auf Brodsky *et al.* (2007) verwiesen, die diese Überlegung im Kontext eines alignment-basierten Lernalgorithmus aufgreifen:

> There is a great deal of evidence suggesting that parents produce structured dialogues when talking with very young children. Parents' speech to young children is highly repetitive and often includes clusters of partial self-repetitions – *variation sets* – when speaking to young children acquiring language. [...V]ariation sets seem to be ideal environments for learning lexical items and constituent structures. By holding most of the utterance constant, while altering it slightly [...], parents may allow children to *discover* lexical items, syntactic constituents, and their place in the syntax, vis-à-vis comparison and contrast, as envisaged (in the context of the discovery of grammar by linguists) by Zellig Harris [...].

[171] So weist bspw. auch Cramer 2007, Abschnitt 3.1 darauf hin, dass die u.a. in van Zaanen & Geertzen (2008) verwendete ATIS-Baumdatenbank relativ kurze Sätze enthält (durchschnittlich 7,5 Wörter pro Satz), die zudem einer einfachen Syntax folgen.In Ermangelung einer Zugriffsmöglichkeit auf die zugrundeliegende *Penn Treebank*, welches nicht mit den in Abschnitt 4.1.1 erwähnten Open-Access-Prinzipien kompatibel ist, kann dies jedoch nicht überprüft werden.

	F-Score				
Satzlänge	3 – 6	7 – 10	11 – 14	15 – 22	3 – 22
Berkeley	100	100	100	100	100
Right	61,33	56,4	48,39	40,32	57,28
Select Width	50,27	41,47	34,72	28,54	44,69
Width Random	33,2	19,33	14,19	10,36	25,28
Select W/O (Best)	51,89	44,12	37,44	30,13	47
W/O (Best) Random	31,41	18,84	13,28	9,99	23,82
Select W/O (All)	52,41	44,17	37,35	30,46	47,51
W/O (All) Random	31,41	19,28	13,67	9,99	25,09

Tabelle 5.12: Evaluation verschiedener *Suffix-Select*-Varianten am CHILDES-Korpus. Die Tabelle zeigt die F-Scores der untersuchten Verfahren nach Vergleich mit den vom Berkeley Parser generierten Strukturen bei unterschiedlicher Beschränkung der maximalen Satzlänge. Um die Vergleichbarkeit der Ergebnisse zu ermöglichen, wurden jeweils 15.000 Sätze untersucht; lediglich bei Satzlänge 15 – 22 standen nur 6.100 Sätze zur Verfügung. Die letzte Spalte dient als Referenz und zeigt das Ergebnis nach Prozessierung der ersten 15.000 Sätze mit einer Länge zwischen 3 und 22.

Dennoch wird das hier verwendete Experiment auf einen Ausschnitt des (auch von Brodsky *et al.* (2007) analysierten) *Child Language Data Exchange System* (CHILDES)[172] angewendet, da – wie bereits in Abbildung 5.3 auf Seite 166 gezeigt – die in CHILDES enthaltenen Sätze größtenteils deutlich kürzer als die der bisher untersuchten Korpora sind, und da zudem jede Wortform durchschnittlich über 100 mal verwendet wird. Unabhängig von der Frage, ob ein Zusammenhang zwischen Alignment-Verfahren und Spracherwerbsprozessen angenommen werden kann, eignet sich das CHILDES-Korpus somit zur Überprüfung der u.a. in Cramer (2007) aufgestellten Hypothese, dass die Ergebnisse eines Alignment-Verfahrens stark von Satzlänge und lexikalischer Redundanz abhängig sind.

Das Experiment wurde mehrfach ausgeführt, wobei minimale und maximale Satzlänge variiert wurden. Tabelle 5.12 fasst den F-Score der unterschiedlichen Verfahren (nach Vergleich mit den vom Berkeley Parser erzeugten Annotationen) zusammen; für die vollständige Analyse sei auf die Tabellen A.22 bis A.26 in Anhang A.3.5 verwiesen. Es zeigt

[172] Die mehrsprachige CHILDES-Datenbank (online unter `http://childes.psy.cmu.edu/`) wird seit 1958 permanent erweitert und enthält u.a. Transkriptionen von Gesprächen zwischen Kindern und Bezugspersonen, die teilweise mit POS-Tags angereichert wurden (vgl. MacWhinney & Snow 1990). In der hier durchgeführten Untersuchung werden die Transkriptionen der US-amerikanischen Datenbank analysiert, wobei ausschließlich die Äußerungen, die von Bezugspersonen stammen und morphosyntaktisch annotiert wurden, verarbeitet werden.

sich, dass die untersuchten Verfahren auch bei Beschränkung der Satzlänge die von *Right* erzielten Ergebnisse nicht erreichen können. Werden nur Sätze mit drei bis sechs bzw. sieben bis zehn Wörtern untersucht, liegt der F-Score allerdings über bzw. nahe an den Ergebnissen, die auch in van Zaanen & Geertzen (2008) beobachtet wurden. Auch die in der letzten Spalte von Tabelle 5.12 aufgeführte Analyse kann als übereinstimmend mit den Ergebnissen aus van Zaanen & Geertzen (2008) betrachtet werden. Die durchschnittliche Satzlänge dieses Korpusausschnitts liegt mit 5,6 jedoch niedriger als im von van Zaanen & Geertzen (2008) untersuchten ATIS-Korpus.

5.4.4 Erweiterte Präprozessierung

In einem weiteren Experiment wurde untersucht, ob eine Reduktion der Anzahl unterschiedlicher Types die Ergebnisse verbessert. So verwenden Klein & Manning (2002, vgl. Abschnitt 2) und Bod (2006, vgl. Abschnitt 2) zur Evaluation ähnlicher Verfahren POS-Tags an Stelle von Wörtern, wodurch die Anzahl unterschiedlicher Types stark reduziert und folglich die Häufigkeit längerer N-Gramme erhöht wird. Zwar konnte dies innerhalb eines Tesla-Experimentes einfach umgesetzt werden, führte jedoch nicht zu besseren Ergebnissen, was sich mit den von Cramer (2007, S. 43f) beschriebenen Beobachtungen bezüglich ähnlicher Vorverarbeitung des dort untersuchten Korpus deckt.

Auch eine teilweise Ersetzung von Wörtern durch POS-Tags konnte die Ergebnisse nicht verbessern: Mit Hilfe der Komponenten *Gazetteer*, *Stanford Named Entity Detector* und *Order Based Sequencer* (vgl. Anhang B.7.1, B.7.2 und B.7.4) werden *Named Entities*[173] durch Kategoriebezeichnungen ersetzt sowie bspw. Präpositionen, Namen und temporale Ausdrücke durch POS-Tags ausgetauscht, während alle übrigen Wörter unverändert übernommen werden – Abbildung 5.12 zeigt den schematischen Versuchsaufbau, während die bereits auf Seite 146 dargestellte Abbildung 4.20 veranschaulicht, wie die Wortsequenzen der Korpora reorganisiert und neu typisiert werden.

Da jedoch keine Verbesserungen festgestellt werden konnten, wird hier auf eine ausführliche Ergebnispräsentation verzichtet – das Experiment dient somit lediglich zur Veranschaulichung der Möglichkeiten, die sich aus der Kombination unterschiedlicher Komponenten in Tesla ergeben (vgl. auch Anhang C.5).

[173] Dabei handelt es sich um teilweise aus mehreren Wörtern bestehende Begriffe (wie etwa *Universität zu Köln* oder *Nikola Tesla*), die unterschiedlichen Kategorien zugeordnet werden können. Hier wurden die in Faruqui & Padó (2010) beschriebenen Klassifizierer verwendet, um Orte, Personen und Organisationen zu detektieren.

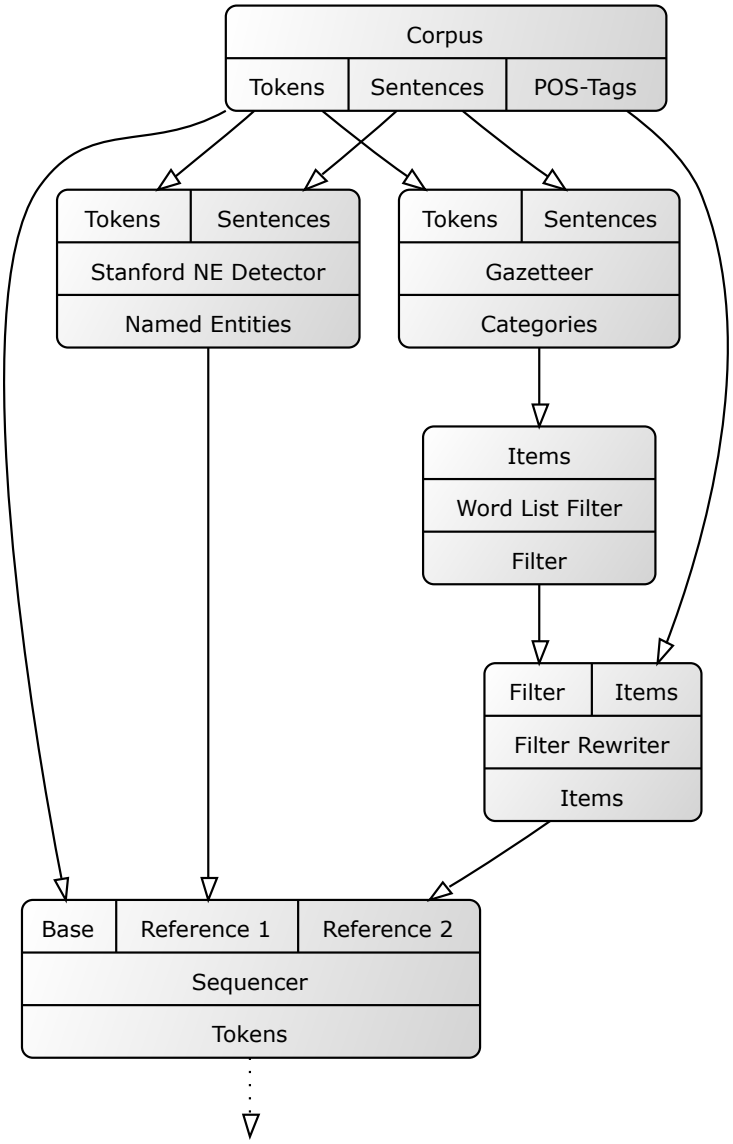

Abbildung 5.12: Schematische Darstellung der erweiterten Präprozessierung der untersuchten Korpora. Die *Gazetteer*-Komponente ist so konfiguriert, dass ausschließlich Präpositionen ausgezeichnet werden. Diese Annotationen werden für die Generierung eines Filters verwendet, um vom *Filter Rewriter* POS-Tags für die akzeptierten Annotationen produzieren zu lassen. Der *Order-Based Sequencer* erzeugt eine neue Symbol-Sequenz, in der POS-Tags, vom *Stanford NE Detector* erkannte *Named Entities* sowie Wörter des Ausgangstextes sequentiell angeordnet sind.

5.5 Ausblick: Alignment-basierte Kategorisierung von Wörtern

Wie die in den vorherigen Abschnitten analysierten Experimente zeigen, ist eine Extraktion syntaktischer Informationen mit den hier verwendeten Verfahren und Korpora nur stark eingeschränkt möglich. Dies kann u.a. darauf zurückgeführt werden, dass die untersuchten Korpora nicht genügend Beispiele enthalten, anhand derer die analysierten Verfahren korrekte Strukturen ermitteln und von falschen Annahmen unterscheiden können. Die Untersuchung des CHILDES-Korpus zeigte zwar, dass die Beschränkung auf kürzere Sätze mit einfacherer syntaktischer Struktur die Qualität der Ergebnisse verbessern kann, eine fehlerfreie Strukturierung kann jedoch, wie bereits in Abschnitt 5.1 erläutert, auf diese Weise nicht erzeugt werden. Dies schließt allerdings nicht aus, dass syntaktische oder paradigmatische Strukturen zumindest teilweise extrahiert werden können: So greifen bspw. Clark & Eyraud (2007) die Analyse von Gold (1967) auf und stellen einen Algorithmus vor, der *substituierbare kontextfreie Sprachen* erlernen kann.[174] Dabei handelt es sich um formale Sprachen, die strikt dem Konzept der Distributionsanalyse nach Harris (vgl. Abschnitt 2.1.1) folgen: Können zwei Zeichenketten X, Y im gleichen Kontext auftreten, so gilt in einer derartigen Sprache, dass diese Zeichenketten in sämtlichen Kontexten substituierbar sind (vgl. Clark & Eyraud 2007, Abschnitt 2) – unter dieser Voraussetzung können falsche Analysen (wie anhand des in Abschnitt 2.1.2 beschriebene Beispiels *Tesla worked/Tesla will work* gezeigt) vermieden werden. Eine substituierbare kontextfreie Sprache ist somit nicht nur zwangsläufig frei von lexikalischer Ambiguität, sondern zeichnet sich vor allem dadurch aus, dass Substitutionsregeln fehlerfrei aus positiven (Einzel-)Beispielen abgeleitet werden können. Es ist offensichtlich, dass es sich bei natürlichen Sprachen nicht um substituierbare kontextfreie Sprachen handelt, trotzdem bietet es sich an, diesen Gedanken an natürlicher Sprache zu untersuchen. Die in den Abschnitten 5.4.1 bis 5.4.2 vorgestellten Ergebnisse legen es dabei nahe, die Untersuchung nicht auf syntaktische Strukturen anzuwenden: das von *Select* produzierte Ausgangsmaterial enthält hierfür zu viele fehlerhafte Auszeichnungen. Stattdessen wird im Folgenden untersucht, ob sich mit Hilfe von Alignment semantische und morphosyntaktische Assoziationen zwischen Einzelwörtern detektieren lassen, ob also Wörter, die in identischen Kontexten verwendet werden können, auch in ihrer Bedeutung verwandt sind bzw. der-

[174] Die Autoren verwenden das von Gold definierte Lernbarkeitsmodell *Identifiable in the limit*: Einem Algorithmus werden sequentiell Sprachbeispiele übergeben, wobei dieser nach jedem Beispiel ein neues Sprachmodell generiert. Ist die Sprache erlernbar, so bleibt das Modell nach einer (unbestimmten, aber endlichen) Zeitspanne unverändert, und die Sprache wurde vom Algorithmus identifiziert (vgl. Gold 1967, S. 449).

selben Wortart angehören.

Wie bereits in Kapitel 1 erwähnt, ist die Evaluation eines Verfahrens zur Extraktion ähnlicher Bedeutungen ungleich komplexer als die eines Verfahrens zur Strukturauszeichnung. In diesem Abschnitt wird daher lediglich gezeigt, wie ein derartiges Experiment in Tesla umgesetzt werden kann. Aufgrund der in Kapitel 1 beschriebenen Probleme ist eine mit der Evaluation an einem Gold-Standard vergleichbare Auswertung der Ergebnisse nicht möglich, es wird jedoch gezeigt, wie in Tesla eine derartige Evaluationsschnittstelle mit Hilfe des TRS umgesetzt und exemplarisch auf Basis von WordNet implementiert werden kann.

In Kapitel 1 wurde der Vorschlag von Palmer *et al.* (2006) erwähnt, die Evaluation von Verfahren zur Aufdeckung semantischer Merkmale dadurch zu ermöglichen, dass eine gemeinsam definierte und akzeptierte Evaluationsform verwendet wird. Die hier verwendeten Rollen und Komponenten wurden hingegen nicht gemeinschaftlich definiert oder entwickelt, doch kann das TRS dazu genutzt werden, den dafür notwendigen Prozess des wissenschaftlichen Austauschs zu virtualisieren: Durch Distribution von Evaluations- und Vergleichskomponenten können neue Metriken vorgeschlagen und extern begutachtet werden; durch anschließende Reimplementation oder durch weitergehende Spezifikation der zugrundeliegenden Rolle ist zu jeder Zeit die Möglichkeit einer aktiven Beteiligung am Entwicklungsprozess gegeben. Im Unterschied zum Vorschlag von Palmer *et al.* (2006) ist es dabei nicht notwendig, dies im Rahmen einer Konferenz oder eines Workshops durchzuführen; zudem kann der Prozess jederzeit neu angestossen werden, um etwa auf erweiterte Anforderungen zu reagieren oder neue Evaluationsmaße einzubinden.

In Abschnitt 5.5.1 wird zunächst untersucht, wie groß die Gemeinsamkeiten von funktionalen Kategorien (vgl. Abschnitt 2.2) und grammatischen Kategorien sind, um in Abschnitt 5.5.2 die Eignung von Alignment-Verfahren zur Extraktion semantischer Ähnlichkeit zu analysieren. Beide Analysen sind dabei experimentell und weniger ausführlich als die zuvor zur Detektion syntaktischer Strukturen vorgestellten Ansätze – sie verdeutlichen jedoch, wie sich mit den in Abschnitt 4.1 beschriebenen Konzepten von Tesla computerlinguistische Fragestellungen experimentell beantworten lassen, und wie die maßgeblich durch das *Tesla Role System* realisierte Komponentendefinition dazu beiträgt, die Wiederverwertbarkeit von Komponenten zu ermöglichen. Um dies zu unterstreichen, wird in Abschnitt 5.5.3 mit dem *Hyperspace Analogue to Language*-Modell (*HAL*) nach Lund & Burgess (1996) die Integration eines weiteren Verfahrens zur Ermittlung semantischer Assoziationen zwischen Wörtern in Tesla vorgestellt.

5.5.1 Morphosyntaktische Analyse

Das im folgenden beschriebene Experiment veranschaulicht, wie die Wiederverwertbarkeit von Komponenten die Entwicklung und Evaluation von Verfahren vereinfacht: Die als *Substitution Rule Generator* bezeichnete Komponente nutzt die durch den (bereits in Abschnitt 5.3.1 beschriebenen und für die Extraktion syntaktischer Hypothesen verwendeten) *Hypothesis Generator* erzeugten Strukturhypothesen, um die Extraktion von Substitutionsregeln umzusetzen. Um eine Evaluation zu ermöglichen, ist die Komponente so konfiguriert, dass (im Kontrast zu den zuvor durchgeführten Experimenten) ausschließlich Hypothesen über Einzelwörter analysiert werden. Jede Hypothese, die zwei oder mehr einzelne Wörter miteinander in Relation stellt, wird in Regeln der Form $xay \Rightarrow xBy$ konvertiert, wobei x und y den Substitutionskontext, a ein einzelnes substituierbares Wort und B die Menge der einsetzbaren Wörter repräsentieren. So wird aus dem in Beispiel 5.3 dargestellten Auszug der Analyse des CHILDES-Korpus u.a. die Regel erzeugt, dass *lamp* im Kontext von *put the ... on the table* durch die Wörter *plate* und *trains* substituiert werden kann.

5.3 a. ... put the { lamp, plate, trains } on the table...

b. ... put the { farmer, butterfly, car } on top of...

c. ... that's your { lamp, sock, bad } ∅

Die vom *Substitutuion Rule Generator* erzeugten Regeln werden auf Basis der gemeinsamen Kontexte generiert, was zunächst dazu führt, dass ein Großteil der Regeln nur wenige Wörter miteinander assoziiert: So sind die in Beispiel 5.3.a und 5.3.b dargestellten Mengen vollständig, d.h. im untersuchten Korpusausschnitt wurden keine weiteren substituierbaren Wörter für den jeweiligen Kontext gefunden. Daher werden diese Regeln vom *Generalized Substitution Rule Generator* dahingehend weiterverarbeitet, dass anschließend zu jeder verwendeten Wortform genau eine Regel vorhanden ist, die sämtliche

austauschbaren Wörter referenziert: Für das Wort *lamp* in Beispiel 5.3 wird etwa die Regel generiert, dass es in jedem Kontext u.a. durch die Begriffe *plate, trains, sock* und *bad* substituierbar ist – dies entspricht dem Modell, das für eine substituierbare kontextfreie Sprache, wie in Clark & Eyraud (2007) definiert, verwendet werden könnte.

Beide Formen von Substitutionsregeln können hier als Mengen von Tokens bzw. Types betrachtet werden, was für die Evaluation ausgenutzt werden kann: Es wird untersucht, inwieweit die einzelnen Elemente einer Regel (bzw. Menge) hinsichtlich der in den Korpora verwendeten POS-Tags homogen sind, so dass für diese Untersuchung ein Gold-Standard zur Verfügung steht. Die Bewertung der Regeln kann mit Hilfe des *Purity*-Maßes berechnet werden, das hier aus Manning *et al.* (2008, Abschnitt 16.3) übernommen wird, wo es zur Evaluation von Clustering-Verfahren verwendet wird. Für jedes Cluster (bzw. jede Menge substituierbarer Wörter) wird untersucht, wie viele Elemente gleicher Kategorie im Cluster enthalten sind. Über alle Cluster wird anschließend die Summe der jeweils häufigsten Kategorie gebildet und durch die Anzahl aller analysierten Elemente geteilt, formal definiert als

$$purity(\Omega, \mathbb{C}) = \frac{1}{N} \sum_{i=1}^{k} \max_{j}(\omega_i \cap c_j) \tag{5.4}$$

Ω bezeichnet dabei die Menge aller Cluster $\omega_1 \ldots \omega_k$, $\mathbb{C}$ die Menge aller Klassen $c_1 \ldots c_j$. Im hier vorliegenden Anwendungsfall können die Substitutionsregeln als Cluster betrachtet werden; die Kategorie eines Elementes korrespondiert mit dem jeweils assoziierten POS-Tag.

Analog zu den in den vorherigen Abschnitten durchgeführten Experimenten werden auch hier zufallsbasierte Verfahren genutzt, um eine untere Schranke für zu erwartende Ergebnisse zu ermitteln. Die *Random Group Generator*-Komponente erzeugt zufällige Substitutionsregeln, wobei nicht nur deren Anzahl und Kardinalität aus den Annotationen einer Referenzkomponente ermittelt wird, sondern auch die Gesamtmenge der substituierbaren Elemente übernommen (und lediglich neu verteilt) wird. Im Gegensatz zu der zufallsbasierten strukturellen Auszeichnung, die in Abschnitt 5.2 erläutert wurde, nutzt die hier verwendete Zufallskomponente deutlich mehr Informationen des Referenzverfahrens, insbesondere ist keine Variation bezüglich der Gesamtmenge aller substituierbaren Wörter möglich. Daher wird noch eine weitere, als *Custom Choice Random Group Generator* bezeichnete Komponente verwendet – diese unterscheidet sich vom *Random Group Generator* dadurch, dass die assoziierten Elemente nicht unmittelbar von der Referenz-

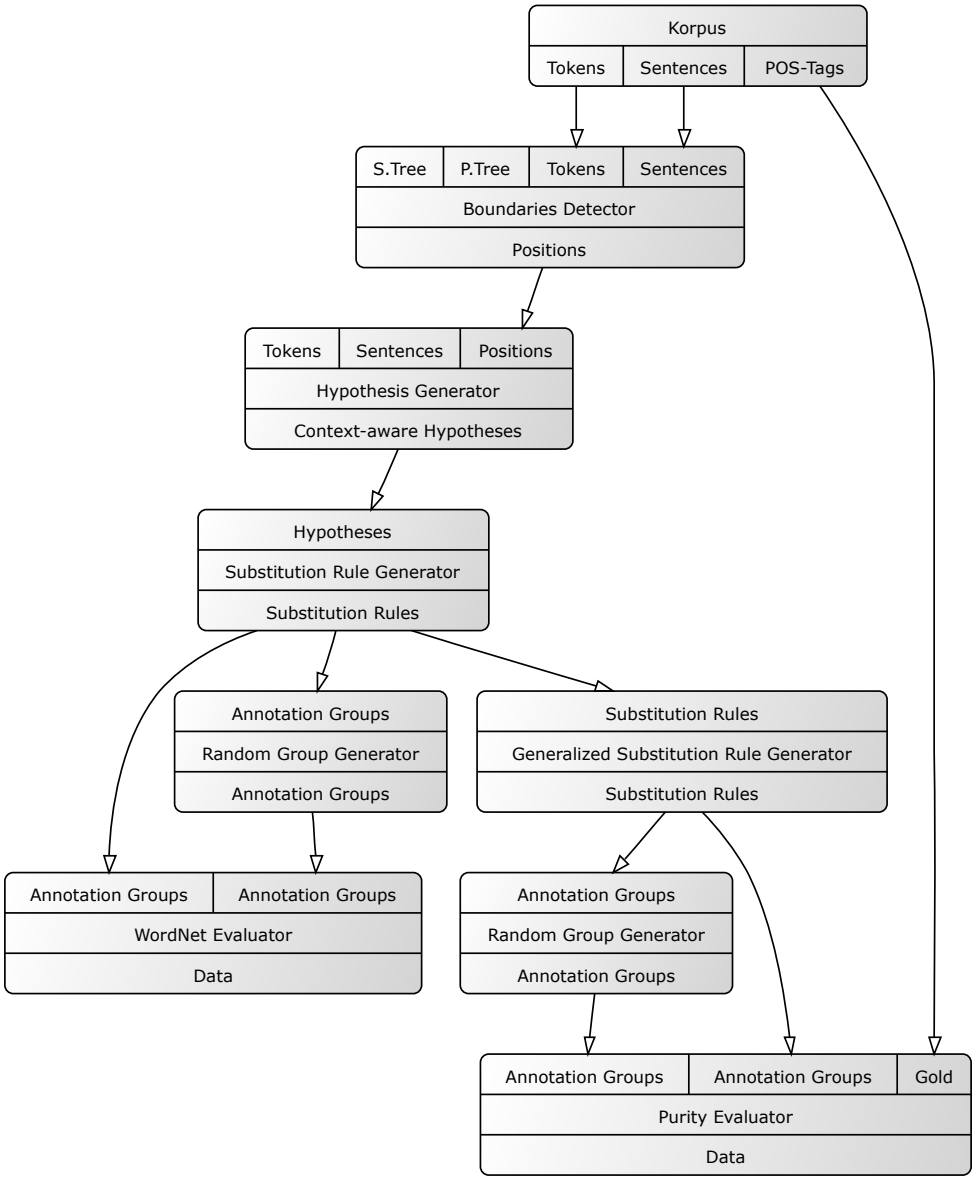

Abbildung 5.13: Bewertung von Substitutionsregeln. Neben der Evaluation kategorieller Ähnlichkeit ist hier bereits die in Abschnitt 5.5.2 beschriebene Komponente zur Berechnung konzeptueller Ähnlichkeit anhand von *WordNet* dargestellt.

Korpus	Verfahren	Regeln	Elemente	Types	Purity
TüBa-D/S	Generalized Rules	2.885	161.216	48	0,37
	Random	2.885	161.216	48	0,21
	Random (Custom)	2.885	161.216	48	0,35
TüBa-D/Z	Generalized Rules	4.739	114.403	48	0,68
	Random	4.739	114.403	48	0,53
	Random (Custom)	4.739	114.403	35	0,51
TüBa-E/S	Generalized Rules	1.401	84.879	34	0,32
	Random	1.401	84.879	34	0,22
	Random (Custom)	1.401	84.879	31	0,25
BNC-G	Generalized Rules	3.300	122.457	79	0,53
	Random	3.300	122.457	79	0,41
	Random (Custom)	3.300	122.457	58	0,24
CHILDES	Generalized Rules	3.385	337.929	67	0,59
	Random	3.385	337.929	67	0,51
	Random (Custom)	3.385	337.929	73	0,33

Tabelle 5.13: Morphosyntaktische Auswertung von Substitutionsregeln

komponente übernommen werden, sondern dass (auf Basis der Anzahl unterschiedlicher Types und deren Häufigkeitsverteilung, wie von der Referenz vorgegeben) eine eigene, zufallsbasierte Auswahl vorgenommen wird. Abbildung 5.13 zeigt den schematischen Versuchsaufbau; Tabelle 5.13 fasst die Ergebnisse nach Anwendung auf die bisher betrachteten Korpora zusammen.

Es zeigt sich, dass die Ergebnisse zwischen schriftsprachlichen und umgangssprachlichen Korpora erneut stark abweichen: Bei TüBa-D/S, TüBa-E/S und CHILDES liegt die Purity des Verfahrens nur leicht über dem besten Wert, den eines der *Random*-Verfahren liefert; insbesondere ist die Abweichung zum *Custom Random*-Verfahren bei den Tübinger Korpora gesprochener Sprache sehr gering. Die Abweichung bei TüBa-D/Z und BNC-G ist hingegen deutlich stärker – das beste Ergebnis mit einer Purity von 0,68 wird beim TüBa-D/Z erzielt.

Wie bereits eingangs erwähnt, repräsentieren generalisierte Substitutuionsregeln keine morphosyntaktische Ambiguität, was sich zwangsläufig negativ auf die Ergebnisse auswirkt, was allerdings auch zu neuen Fragestellungen führt: So könnte untersucht werden, ob die Qualität von Substitutionsregeln dadurch verbessert werden kann, dass nur die Elemente berücksichtigt werden, die in n unterschiedlichen Kontexten als substituierbar erkannt wurden, oder dass Elemente unberücksichtigt bleiben, wenn sie andernfalls in *zu*

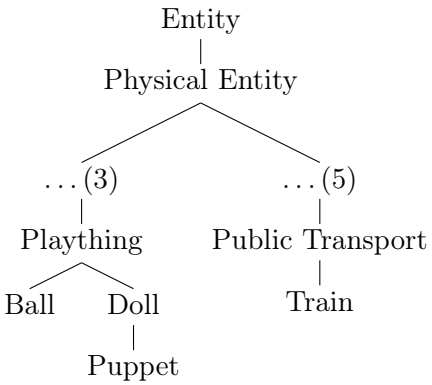

Abbildung 5.14: Ausschnitt der WordNet-Hierarchie. Dargestellt ist ein Teil der Hyponymie-Relation zu den Begriffen *Ball*, *Puppet* und *Train*. Ausgelassene Knoten sind durch Punkte (. . .) und die Länge des nicht dargestellten Pfades markiert.

vielen Regeln enthalten wären. Auch wäre es denkbar, einzelne Merkmale der im Kontext enthaltenen Wörter (wie etwa ihre Häufigkeit) zu berücksichtigen. Schließlich könnte auch die Form der Evaluation modifiziert werden: Die kategorielle Zuordnung des jeweiligen Gold-Standards wurde ohne weitere Modifikation übernommen, was dazu führt, dass bspw. Nomen und Eigennamen als unabhängige Wortarten interpretiert werden. Diesen Überlegungen wird jedoch nicht weiter nachgegangen – stattdessen wird im folgenden Abschnitt untersucht, inwieweit mit Hilfe von Alignment eine semantische Kategorisierung durchgeführt werden kann.

5.5.2 Semantische Analyse durch Alignment

Um eine semantische Evaluation zu ermöglichen, wird die WordNet-Datenbank[175] verwendet. WordNet enthält über 100.000 Synonym-Sets englischer Nomen, Verben, Adjektive und Adverbien, die (innerhalb der jeweiligen Wortart, sofern dort vorhanden) mit semantischen Relationen, wie Hyponymie oder Antonymie, verknüpft wurden. Synonym-Sets sind ebenfalls unter semantischen Gesichtspunkten organisiert und enthalten nicht nur lexikalische Synonyme (wie etwa *sound*, *gang*, *band* und *closed chain* als mögliche Synonyme von *ring*); vielmehr sind polyseme Begriffe mehrfach aufgeführt. So ist *ring* bspw. als *characteristic sound*, *association of criminals* und als *chain of atoms* definiert und entsprechend der Bedeutung in unterschiedlichen Taxonomien referenziert, was u.a. beim Zugriff auf das jeweilige Hyperonym deutlich wird.

Zur Bewertung der generierten Substitutionsregeln wird das von Wu & Palmer (1994, S. 136) als *conceptual similarity* bezeichnete Maß verwendet: Für zwei Begriffe C_1 und

[175] Online unter http://wordnet.princeton.edu.

C_2 wird zunächst das erste Hyperonym C_P ermittelt, das beide Begriffe umfasst. Die konzeptuelle Ähnlichkeit von C_1 und C_2 kann dann definiert werden als

$$ConSim(C_1, C_2) = \frac{2 \cdot len(ROOT, C_P)}{len(C_P, C_1) + len(C_P, C_2) + 2 \cdot len(ROOT, C_P)} \tag{5.5}$$

$len(X, Y)$ bezeichnet dabei die Länge des Pfades von X nach Y. Da analog zum morphosyntaktischen Ähnlichkeitsmaß auch hier der ermittelte Wert stets zwischen 0 und 1 liegt, kann die Bewertung einer aus mehreren Substituenten $b_1 \dots b_n$ bestehenden Regel durch $AvgConSim(a, B) = \sum_{i=1}^{n} \frac{ConSim(a,b_i)}{n}$ berechnet werden.

Abbildung 5.14 veranschaulicht die Berechnung der konzeptuellen Ähnlichkeit für die Wörter *puppet* und *ball*: Erstes gemeinsames Hyperonym beider Begriffe ist das Konzept *plaything*, das 5 Kanten vom Wurzelknoten *Entity* entfernt liegt. Die Distanz von *puppet* und *plaything* beträgt 2, zwischen *ball* und *plaything* liegt hingegen nur eine Kante, weshalb die konzeptuelle Distanz berechnet werden kann als $\frac{2 \cdot 5}{1+2+2 \cdot 5} \approx 0,77$.

Das von Wu & Palmer (1994) vorgeschlagene Ähnlichkeitsmaß berücksichtigt allerdings nicht, dass die Länge der Hyperonym-Pfade in WordNet nicht normalisiert ist: Da sich bspw. Lebewesen deutlich besser kategorisieren lassen als abstrakte Konzepte, und dies in WordNet auch entsprechend umgesetzt wurde, sind verschiedene ConSim-Ergebnisse nur schwer vergleichbar. Um dem entgegenzuwirken, wurde die ConSim-Bewertung entsprechend des Vorschlags von Wang & Hirst (2011, S. 1009) um den Gewichtungsfaktor α erweitert[176], so dass der Einfluss der getrennt verlaufenden Hyperonym-Pfade auf das Ergebnis der Bewertung mit steigendem Wert des Konfigurationsparameters α (> 1) abnimmt.

$$ConSim(C_1, C_2) = \frac{2 \cdot len(ROOT, C_P)^{\alpha}}{len(C_P, C_1) + len(C_P, C_2) + 2 \cdot len(ROOT, C_P)^{\alpha}} \tag{5.6}$$

Bei der Bewertung der erzeugten Substitutionsregeln an WordNet ergeben sich dennoch zahlreiche Probleme, die eine Interpretation der Ergebnisse erschweren: So sind bspw. nicht zwangsläufig alle Wörter, die in einer Regel enthalten sind, auch in WordNet repräsentiert.[177] Zudem müssen die in einer Regel vorkommenden Wörter nicht zwangsläufig zur

[176] Die Untersuchungen von Wang & Hirst (2011) gehen über diese Modifikation hinaus und beziehen zudem weitere Verfahren zur Berechnung der konzeptuellen Ähnlichkeit mit ein, die hier alterantiv zum ConSim-Maß verwendet werden könnten. Ziel dieses Abschnitts ist es jedoch nicht, eine umfassende Analyse WordNet-basierter Bewertungsverfahren durchzuführen – dies wäre vielmehr ein weiterer Untersuchungsgegenstand, der in Tesla auf ähnliche Weise wie das hier beschriebene Experiment betrachtet werden könnte.

[177] Insbesondere verwendet WordNet kein Vollform-Lexikon: Nomen sind bspw. ausschließlich im Singular

gleichen Wortart gehören – dies ist jedoch eine Voraussetzung für die Berechnung der konzeptuellen Ähnlichkeit, da in WordNet keine Wortart-übergreifenden Relationen modelliert wurden. Daher wird die Generierung von Substitutionsregeln auf Nomen beschränkt, indem die in den Korpora enthaltenen POS-Tags zur Filterung verwendet werden. Die Bewertung einer Regel erfolgt grundsätzlich wie oben definiert, d.h. durch Berechnung des Mittelwerts der Kontextähnlichkeiten aller Wortpaare. Da die untersuchten Korpora jedoch nicht semantisch annotiert sind, ist die (kontextabhängige) Wahl des korrekten Polysems aus einem Synonym-Set nicht fehlerfrei möglich.[178] Beim CHILDES-Korpus ergibt sich zusätzlich noch das Problem, dass die in WordNet modellierte Hierarchie die semantischen Besonderheiten von an Kinder gerichteter Sprache nicht abbildet. So könnte bspw. argumentiert werden, dass die Begriffe *sandbox* und *beach* aus der Perspektive eines Kindes konzeptuell ähnlicher als aus der Perspektive eines Erwachsenen sind, die entsprechende Assoziation in WordNet jedoch nicht vorhanden ist (siehe Tabelle 5.14). Enthält WordNet für einen der Begriffe mehrere Bedeutungen, wird in diesem Fall die beste Ähnlichkeitsbewertung aller möglichen Bedeutungspaare verwendet, auch wenn dies dazu führen kann, dass bspw. der Begriff *bed* innerhalb einer Regel gleichzeitig als physikalisches Objekt und als Gebiet (vgl. Tabelle 5.14) interpretiert wird.

Im Gegensatz zu den syntagmatischen Analysen in den vorherigen Abschnitten ist die Interpretation des ConSem-Wertes daher deutlich schwieriger, denn während Precision und Recall die prozentuale, objektive Übereinstimmung von zwei Mengen angeben, fließen in den ConSim-Wert WordNet-spezifische Überlegungen ein. Tabelle 5.14 zeigt exemplarisch einige Wortpaare, die vom *Paradigm Extractor* extrahiert und mit dem ConSim-Maß (mit Alpha-Werten 1 und 2) evaluiert wurden. Das dort ebenfalls aufgeführte, als *SharedPath* bezeichnete Maß kann als weitere Bewertungsfunktion betrachtet werden: Die zugrundeliegende Formel berücksichtigt im Unterschied zu ConSim die WordNet-Taxonomie lediglich bis zu einer konfigurierbaren Tiefe T, wobei der Wurzelknoten der Taxonomie ignoriert wird. Die durch

$$SharedPath(C_1, C_2) = \frac{2 \cdot len(ROOT, C_P^T) - 1}{len(ROOT, C_1^T) - 1 + len(ROOT, C_2^T) - 1} \tag{5.7}$$

aufgeführt, so dass Substitutionsregeln für Plural-Wortformen nicht evaluiert werden können, ohne dass diese zunächst auf die Singular-Form zurückgeführt werden. Auf diesen Schritt wurde jedoch verzichtet.

[178] Zwar könnte versucht werden, die korrekte Bedeutung eines Wortes mit Hilfe von Disambiguierungsverfahren (vgl. bspw. Palmer *et al.* 2006 für Beschreibung und Evaluation verschiedener Ansätze) zu ermitteln und als Grundlage für die Bewertung einer Regel zu verwenden – da es sich bei dem hier beschriebenen Experiment jedoch in erster Linie um einen exemplarischen Untersuchungsgegenstand handelt, wurde darauf verzichtet.

Wort 1	Wort 2	$CSim_1$	$CSim_2$	$S.Path_6$	$S.Path_7$	Konzept
Dresser	Counter	0,89	0,98	0,8	0,83	Table
Bit	While	0,83	0,96	0,8	0,83	Time
Bed	Cemetery	0,75	0,94	0,8	0,83	Area
Mouse	Kitty	0,74	0,97	0,8	0,83	Mammal
Elephant	Owl	0,7	0,95	0,8	0,83	Vertebrate
Foot	Knee	0,67	0,89	0,8	0,66	Body Part
Package	Card	0,66	0,86	0,8	0,83	Instrumentality
Cat	Potato	0,56	0,86	0,8	0,83	Organism
Sandbox	Beach	0,4	0,57	0,5	0,5	Object
Questionnaire	Survey	0,31	0,47	0,5	0,33	Communication
Grass	River	0,18	0,18	0,25	0,2	Physical Entity
Puppet	Paycheck	0,0	0,0	0,0	0,0	Entity

Tabelle 5.14: Auswahl substituierbarer Begriffe, die aus dem CHILDES-Korpus extrahiert wurden. *CSim* bezeichnet die Bewertung durch das in Formel 5.6 definierte *ConSim*-Maß; verwendet wurden die Alpha-Werte 1 und 2. In den mit *SPath* beschriebenen Spalten wird das *SharedPath*-Maß (Formel 5.7) verwendet, wobei die Werte 6 und 7 für den Parameter T gewählt wurden. Die Tabelle zeigt, dass eine Evaluation an WordNet nur eingeschränkt möglich ist: So erhalten bspw. die Wortpaare *Mouse/Kitty* und *Elephant/Owl* eine nahezu identische Bewertung, was darauf zurückzuführen ist, dass die vier Begriffe auf tiefster Ebene der in WordNet abgebildeten Hierarchie liegen. Das Wortpaar *Sandbox/Beach* demonstriert hingegen, dass die in WordNet abgebildeten Zusammenhänge nicht der Wahrnehmung entsprechen, und die hohe Ähnlichkeit von *Bed* und *Cemetery* ist darauf zurückzuführen, dass die Bedeutung von *Bed* hier falsch aufgelöst wurde (*a plot of ground in which plants are growing*).

Korpus	Verfahren	Regeln	Elemente	ConSim_2	SharedPath_6
TüBa-E/S	Rules	866	2.662	0,78	0,67
	Random	850	2.528	0,51	0,47
	Random (Custom)	608	1.748	0,45	0,43
BNC-G	Rules	1.208	4.644	0,64	0,57
	Random	1.261	4.564	0,45	0,49
	Random (Custom)	663	2.331	0,37	0,39
CHILDES	Rules	902	4.750	0,69	0,62
	Random	970	4.671	0,63	0,58
	Random (Custom)	507	2.183	0,43	0,41

Tabelle 5.15: Zusammenfassung der Berechnung der konzeptuellen Ähnlichkeit von Substitutionsregeln, die aus den Korpora TüBa-E/S, BNC-G und CHILDES extrahiert wurden. Die Bewertung wurde mit ConSim ($\alpha = 2$) und SharedPath ($maxLength = 6$) ermittelt. Zum Vergleich sind für jedes Korpus die Ergebnisse der *Random*-Verfahren angegeben – die deutlich geringere Anzahl ausgewerteter Regeln beim *Custom Random*-Verfahren ist darauf zurückzuführen, dass ein großer Teil der zufällig gewählten Nomen nicht in WordNet repräsentiert ist und daher nicht ausgewertet werden kann.

definierte Distanz ähnelt dem ConSim-Maß, C_i^T bezeichnet hier jedoch nicht zwangsläufig den Knoten C_i direkt, sondern vielmehr dessen Vorgänger mit Tiefe T (sofern vorhanden). Anhand des in Abbildung 5.14 abgebildeten Ausschnitts der WordNet-Taxonomie beträgt die SharedPath-Distanz (bei $T = 6$) der Begriffe *ball* und *puppet* $0,8$, denn C_P^6 bezeichnet den Knoten *plaything* auf Tiefe 5, so dass die Distanz berechnet werden kann durch $\frac{2 \cdot 4}{5+5}$.

Aufgrund der beschriebenen Probleme, die bei einer Bewertung der Ergebnisse durch Verwendung von WordNet auftreten können, wird hier – analog zum Vergleich von strukturellen Hypothesen mit den vom Berkeley Parser generierten Strukturen – der Begriff Evaluation vermieden.

Um die Ergebnisse trotz der beschriebenen Einschränkungen interpretieren zu können, werden die bereits in Abschnitt 5.5.1 verwendeten, zufallsbasierten Verfahren hier erneut zur Bestimmung einer unteren Schranke eingesetzt. Der Versuchsaufbau ähnelt somit dem im vorherigen Abschnitt beschriebenen Experiment (vgl. Abbildung 5.13); anders als dort werden hier jedoch kontextbezogene Substitutionsregeln analysiert.

Tabelle 5.15 fasst die Auswertung des Experiments für die untersuchten Korpora zusammen – da WordNet auf die englische Sprache beschränkt ist, werden hier lediglich die Korpora TüBa-E/S, BNC-G und CHILDES analysiert. Die Ergebnisse zeigen, dass die Ähnlichkeit der in einer Substitutionsregel assoziierten Begriffe bei beiden Bewertungsmaßen

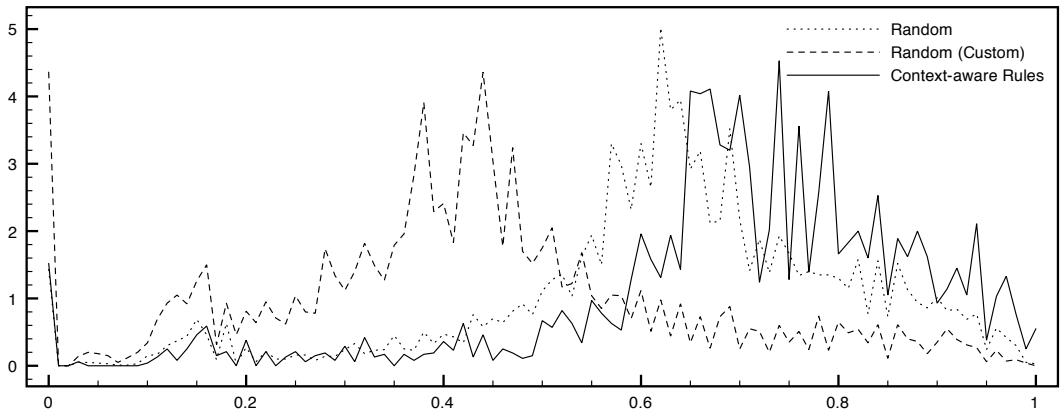

Abbildung 5.15: Verteilung der ConSim-Bewertung nach Analyse des CHILDES-Korpus. Auf der X-Achse ist die ConSim-Bewertung angegeben, auf der Y-Achse der prozentualle Anteil aller Elemente mit der jeweiligen Bewertung. Das Diagramm veranschaulicht die deutlich unterschiedliche Bewertungsverteilung von kontextabhängigen Regeln und zufällig erzeugten Mengen.

deutlich größer ist als die der vom *Custom Random*-Verfahren erzeugten Referenzmengen. Im Vergleich fällt auf, dass das *Random*-Verfahren am CHILDES-Korpus Ergebnisse liefert, die nahezu mit den Substitionsregeln identisch sind – dies deutet darauf hin, dass die Menge der miteinander assoziierten Nomen hinsichtlich ihrer Bedeutungsähnlichkeit bereits relativ homogen ist und nur wenige Teilbereiche der WordNet-Hierarchie umfasst.

Analog zum Vorgehen im vorherigen Abschnitt zeigt Abbildung 5.15 exemplarisch für das CHILDES-Korpus die Häufigkeitsverteilung der ConSim-Bewertung – auch hier sind deutliche Unterschiede zwischen Substitutionsregeln und den durch die zufallsbasierten Verfahren erzeugten Elementmengen erkennbar.

Die Ergebnisse legen nahe, weitere Untersuchungen zur Extraktion semantischer Assoziationen mittels Alignment durchzuführen und auch hier zu evaluieren, wie sich die Qualität der generierten Regeln weiter verbessern lässt. Da die hier und in Abschnitt 5.5.1 beschriebenen Experimente methodisch nah verwandt sind, ist es naheliegend, beide Untersuchungsgegenstände parallel zu analysieren und dabei weiterhin identische Komponenten einzusetzen. Denkbar wäre es jedoch auch, den hier verfolgen Ansatz mit anderen Verfahren zu verbinden, um bspw. Substitutionsregeln auch um solche Begriffe zu ergänzen, die nicht anhand identischer Kontexte ermittelt werden konnten, aber dennoch in ihrer Verwendung starke Übereinstimmungen aufweisen. Ein derartiges Verfahren wird (einschließlich der Integration in Tesla) im folgenden Abschnitt vorgestellt.

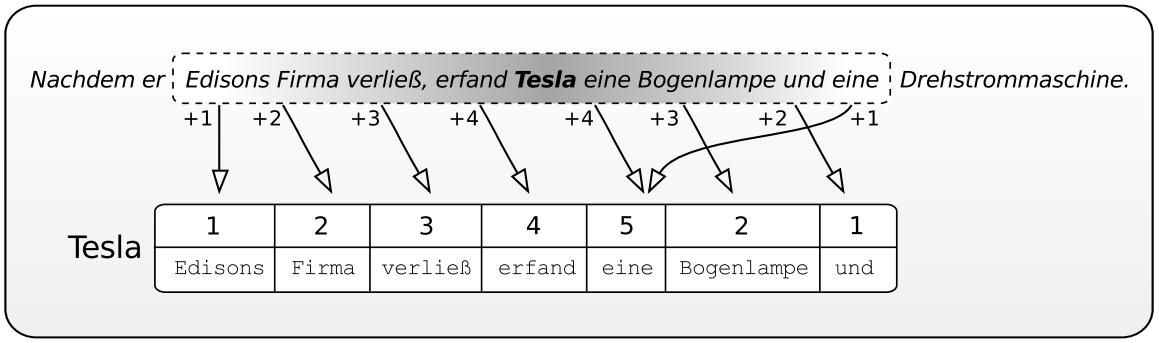

Abbildung 5.16: Schematische Darstellung der Erzeugung von Wortvektoren. Im Beispiel wird der Vektor zu *Tesla* aktualisiert, wobei ein Wortfenster der Breite 4 verwendet wird. Die vektorielle Repräsentation von *Tesla* wird an den Positionen, die durch die 8 Wörter im Kontext definiert werden, erhöht. Der addierte Wert entspricht dabei der Nähe des jeweiligen Wortes zu *Tesla*. Da das Wort *eine* zweifach im Kontext vorhanden ist, ergibt sich an dieser Stelle der Wert 4+1 = 5.

5.5.3 Ausblick: Ergänzende semantische Analyse mit *HAL*

Die in dem vorherigen Abschnitt durchgeführte Fokussierung auf semantische Ähnlichkeit legt es nahe, die bereits in Rolshoven & Schwiebert (2007, Abschnitt 3.4) beschriebenen Überlegungen aufzugreifen und neben dem symbolisch arbeitenden Alignment-Verfahren ein weiteres distributionsbasiertes Verfahren zu untersuchen, das unabhängig von den bisher verwendeten Komponenten arbeitet und einen alternativen, assoziativeren Ansatz verfolgt, Bedeutungsähnlichkeit zu ermitteln. Hierbei handelt es sich um das *Hyperspace Analogue to Language*-Modell (HAL) nach Lund & Burgess (1996), dessen Funktionsweise im Folgenden skizziert[179] wird. Auf eine Auswertung des sich daraus ergebenden Experimentes wird jedoch verzichtet – dieser Abschnitt dient vielmehr dazu, Teslas Potential für weitere strukturalistische Verfahren zu verdeutlichen, sowie mögliche Hypothesen zu umreißen, denen in separaten Untersuchungen nachgegangen werden könnte.

In HAL wird das zu analysierende Korpus in eine (zunächst mit 0 initialisierte) $n \times n$-Matrix überführt, wobei jede der n Wortformen eindeutig je einer Zeile und Spalte zugeordnet wird. Anschließend wird ein Wortfenster (mit konfigurierbarer Breite x) über den Text bewegt, d.h. zu jedem Wort w werden die x unmittelbar vor links bzw. rechts vom Wort vorkommenden Kookurrenzen $l_1 \ldots l_x$ und $r_1 \ldots r_x$ betrachtet (vgl. Abbildung 5.16). Wird w als Zeile in der Matrix interpretiert, bezeichnet jedes Wort im Kontextfenster eine Position im Vektor – der an einer solchen Position vorhandene Wert wird daraufhin (ab-

[179] Für eine ausführliche Beschreibung sei neben Lund & Burgess (1996) auch auf die Erklärungen in Manning & Schütze 1999, Abschnitt 8.5 verwiesen.

hängig von der Distanz der Wörter) erhöht.[180] Nach Abschluss des Prozesses enthält jede Zeile der Matrix eine vektorielle Repräsentation der jeweiligen Wortform (exemplarisch dargestellt in Abbildung 5.16), wobei die Distanz[181] zwischen zwei auf diese Art erzeugten Vektoren umso geringer ist, je häufiger die zugrundeliegenden Wörter in ähnlichen Kontexten verwendet wurden.

Die Umsetzung von HAL in Tesla profitiert wiederum von den Konzepten, die sich aus dem TRS ergeben: So konsumiert die Komponente *Word Vector Generator* bspw. die bereits beschriebenen Filter, um festzulegen, für welche Annotationen Vektoren erzeugt bzw. welche Annotationen innerhalb des Kontextes berücksichtigt werden, so dass etwa Nomen durch kookurrierende Verben beschrieben werden können. Da in der hier durchgeführten Analyse anhand der WordNet-Taxonomie ausschließlich Nomen betrachtet werden[182], wird die Komponente im hier vorgestellten Experiment mit Hilfe eines POS-basierten Filters entsprechend konfiguriert, wodurch gleichzeitig die Menge der zu analysierenden Vektoren stark reduziert werden kann. Ein weiterer, auf Basis einer Wortliste vom *Gazetteer* erzeugter Filter verhindert, dass hochfrequente Wörter bei der Generierung der Vektoren berücksichtigt werden.[183]

Das TRS ermöglicht es, die generierten Vektoren in Form einer Rolle als IO-Schnittstelle zwischen Komponenten zu verwenden, so dass die Interpretation bzw. das Clustering der Vektoren in separaten Komponenten umgesetzt werden kann und sich die Wiederverwertbarkeit dieser Komponenten erhöht (vgl. auch Abschnitt 4.1.4.2). Dies gilt auch für die Weiterverarbeitung der generierten Cluster: Diese werden von einer weiteren Kom-

[180] In Lund & Burgess (1996) werden linker und rechter Kontext eines Wortes getrennt repräsentiert – da hier jedoch lediglich der Kontrast gegenüber Alignment-basierten Verfahren dargestellt werden soll, wird auf diese Erweiterung verzichtet.

[181] Üblicherweise wird hier die Kosinus-Distanz verwendet, so dass die Ähnlichkeit zweier Vektoren v, w anhand des Winkels zueinander berechnet wird:

$$cos(v, w) = \frac{\sum_{i=1}^{n} v_i \cdot w_i}{\sqrt{\sum_{i=1}^{n} v_i^2} \cdot \sqrt{\sum_{i=1}^{n} w_i^2}} \tag{5.8}$$

Dadurch wird gewährleistet, dass die Länge der Vektoren keinen Einfluss auf die Ähnlichkeit ausübt – so kann die auf Wortebene stark abweichende Häufigkeit der Wortvorkommen ausgeglichen werden (vgl. auch Manning & Schütze 1999, Abschnitt 15.2.1).

[182] Neben Nomen sind in WordNet auch Verben hierarchisch organisiert; die Tiefe der Hierarchie weicht jedoch stark von der Tiefe der Taxonomie der Nomen ab, so dass beschlossen wurde, die Analyse auf Nomen zu beschränken.

[183] Dies geschieht unter der Annahme, dass hochfrequente Wörter mit großer Wahrscheinlichkeit in den Kontexten zweier unterschiedlicher Wortformen zu finden sind, so dass ihr Einfluss auf die Berechnung der Distanz der entsprechenden Vektoren vernachlässigt werden kann – da so auch die Dimension der Vektoren reduziert werden kann, müssen während des Clusterings deutlich weniger Rechenoperationen ausgeführt werden.

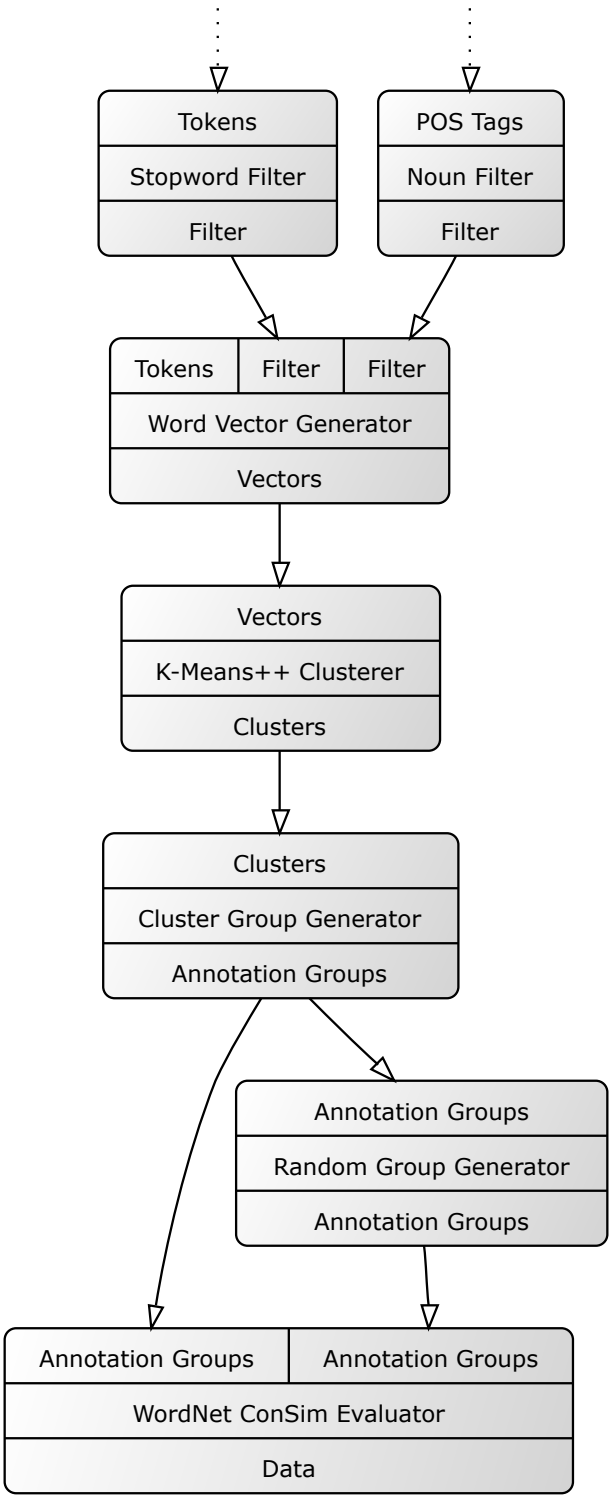

Abbildung 5.17: Schematischer Versuchsaufbau zur Bewertung der konzeptuellen Ähnlichkeit in Clustern, die mit Hilfe des HAL-Verfahrens erzeugt wurden, an WordNet.

ponente interpretiert und zur Gruppierung der zugrundeliegenden Annotationen genutzt, wobei die produzierte Rolle eine Superrolle der vom *Substitution Role Generator* erzeugten Rolle darstellt. Beide Komponenten sind somit bedingt austauschbar, was u.a. dazu führt, dass die generierten Annotationen sowohl von der WordNet-Komponente als auch vom *Random Group Generator* verarbeitet werden können – Abbildung 5.17 veranschaulicht den Versuchsaufbau.

Allerdings bedeutet die funktionale Austauschbarkeit nicht, dass die von den Komponenten erzeugten Daten vergleichbar sind: Zwar kann, wie bereits erwähnt, auch HAL auf Harris' Überlegungen zur Distributionsanalyse zurückgeführt werden, da auch hier die Vermutung, dass die Bedeutung eines Wortes abhängig von den Verwendungskontexten ist, zugrundegelegt wird. Doch während die bisher verwendeten Alignment-Verfahren ausschließlich lokale Kontextähnlichkeit ausnutzen und zwei oder mehr Symbolfolgen nur dann miteinander assoziieren, wenn diese in exakt übereinstimmenden Kontexten auftreten, ist das HAL-Modell weniger strikt und assoziiert Wörter auch dann miteinander, wenn diese lediglich in ähnlichen Kontexten verwendet werden. Insbesondere aber werden durch Alignment-Verfahren Tokens assoziiert, während HAL Types zueinander in Relation stellt und damit implizit eine Gleichsetzung von Signifikant und Signifikat durchführt (siehe Abschnitt 2.1, vgl. auch Rolshoven & Schwiebert 2007, Abschnitt 3.4). Zudem weist HAL im hier beschriebenen Experiment alle Nomen einem Cluster zu, während der *Paradigm Generator* sich auf die Teilmenge der Nomen beschränkt, die identische Verwendungskontexte vorweisen. Diese Unterschiede verhindern den direkten Vergleich beider Ansätze; von einer eigenständigen Evaluation von HAL wird abgesehen, da es u.a. zunächst notwendig wäre, die gesuchte Anzahl von Clustern für jedes Korpus zu bestimmen, um anschließend zu untersuchen, wie sich bpsw. eine Modifikation der Größe des Kontextfensters oder der verwendeten Filter(-Einstellungen) auf die Ergebnisse auswirkt.[184]

In weiteren Experimenten könnte jedoch untersucht werden, inwieweit sich unterschiedliche Verfahren zur Ermittlung semantischer Assoziationen kombinieren lassen: Wie in Abschnitt 5.5.2 gezeigt werden konnte, ist die konzeptuelle Ähnlichkeit innerhalb der durch Alignment erzeugten Substitutionsregeln teilweise sehr hoch – allerdings finden sich auch zahlreiche gegenteilige Beispiele; zudem enthalten die kontextabhängigen Substitutionsregeln häufig nur wenige Begriffe. Hier könnte der Type-bezogene Ansatz von HAL bspw. genutzt werden, um aus wenigen Elementen bestehende Substitutionsregeln zu vereinigen oder um Substitutionsregeln, die große Mengen von Elementen enthalten, durch Cluste-

[184] Als Ausgangsbasis derartiger Untersuchungen kann jedoch das hier skizzierte Experiment verwendet werden (siehe Anhang C.6).

ring in mehrere Regeln aufzuspalten, die hinsichtlich der konzeptuellen Ähnlichkeit ihrer Elemente eine größere Homogenität vorweisen.

Auch eine Ergänzung in umgekehrter Richtung ist denkbar: Wie oben beschrieben, findet in HAL keine Disambiguierung statt, so dass ein Vektor sämtliche Bedeutungen des zugrundeliegenden Symbols repräsentiert. Hier könnte der lokale, auf Kontext-Identität beruhende Ansatz eines Alignment-Verfahrens u.U. dazu genutzt werden, dass polyseme Types erkannt und in zwei oder mehrere Repräsentationen überführt werden können.

5.6 Zusammenfassung

Durch die Anwendung der Verfahren auf unterschiedliche Korpora können sowohl die in van Zaanen (2002) und van Zaanen & Geertzen (2008) vorgestellten positiven Ergebnisse als auch die insgesamt eher negativen Daten aus Cramer (2007) bestätigt werden, da die Qualität der Hypothesen, die durch Alignment-Verfahren generiert werden, stark von der syntaktischen Komplexität der verwendeten Korpora abhängig ist. Während die Analyse des CHILDES-Korpus Ergebnisse liefert, die teilweise sogar besser als die in van Zaanen & Geertzen (2008) beschriebenen Daten nach Anwendung auf das ATIS-Korpus sind, zeigt die Analyse sowohl deutscher als auch englischsprachiger Zeitungstexte, dass die Verfahren hier kaum verwertbare Daten produzieren. Dies wirft die Frage auf, ob es möglich ist, den Prozess der Strukturdetektion so zu modifizieren, dass zunächst anhand einfacher Beispiele eine grundlegende *Grammatik* erlernt wird, auf die anschließend bei der Verarbeitung komplexer Texte zurückgegriffen werden kann. Ein solches Vorgehen würde jedoch zunächst eine weitere Verbesserung der Verfahren erfordern, um den Anteil falscher Hypothesen zu verringern.

Die Analyse verschiedener *Align*-Verfahren in Abschnitt 5.4.1 hat gezeigt, dass die (sich in erster Linie aus hochfrequenten Wörtern ergebende) hohe Produktivität der Verfahren dazu führt, dass sich die Qualität der generierten Hypothesen nur unwesentlich von zufällig erzeugten Hypothesen unterscheidet. In Abschnitt 5.4.2 wurde beobachtet, dass das untersuchte *Select*-Verfahren einen deutlich höheren F-Score erreicht als ein Verfahren, dass zufällig überlappungsfreie Strukturen erzeugt – es wurde jedoch auch gezeigt, dass die Anwendung auf Hypothesen, die durch *Align*-Verfahren generiert wurden, nur geringfügig bessere Ergebnisse lieferte als die Anwendung auf zufällig generierte Hypothesen. Die Schlussfolgerungen, die sich aus beiden Untersuchungen ergaben, wurden in Form einer dritten Experimentreihe evaluiert, in der untersucht wurde, wie sich die Veränderung der minimalen Kontextbreite auf die Hypothesenmenge auswirkt, und wie der *Select*-Prozess

durch Analyse der Belege, die zur Bildung einer Hypothese führten, modifiziert werden kann. Dabei zeigte sich, dass die Ergebnisse für die untersuchten Korpora teilweise stark verbessert werden konnten.

Wie in Abschnitt 5.3 und 5.4.1 gezeigt werden konnte, unterscheiden sich die Hypothesenmengen, die von den untersuchten ABL-Align-Verfahren *All* und *WagnerFischer* erzeugt werden, nur unwesentlich. Gleiches gilt in Bezug auf das im Rahmen dieser Arbeit implementierte, auf N-Gramm-Bäumen basierende Verfahren – auch die in Geertzen & Zaanen (2004) beschriebenen Alignment-Varianten auf Basis von Suffixbäumen produzieren den Autoren zufolge ähnliche Ergebnisse. Dies ermöglicht es nicht nur, zukünftige Untersuchungen auf die deutlich performanteren Ansätze zu beschränken, sondern ist auch die Voraussetzung für einen praktischen Einsatz des Verfahrens.

Die erzielten Resultate legen zwar keine unmittelbare praktische Verwendung von Alignment-Verfahren zur unüberwachten Aufdeckung syntaktischer Strukturen nahe, dies war jedoch auch nicht das Ziel der hier durchgeführten Untersuchungen: Vielmehr sollte analysiert werden, wie die Qualität der Ergebnisse bei Anwendung auf unterschiedliche Korpora variiert, und ob bzw. wie sich die Verfahren optimieren lassen. Auch wenn nur wenige Optimierungsansätze untersucht wurden, konnte dieses Ziel erreicht werden: Precision und Recall lagen nach Ausführung der *Suffix Select*-Komponenten und Erhöhung der minimalen Kontextbreite selbst bei Anwendung auf die Zeitungs-Korpora deutlich über den Ergebnissen, die durch das zufallsbasierte Verfahren erreicht wurden, auch wenn in keinem Experiment die von *Right* vorgegebene Schranke überschritten werden konnte. Zwar genügt dies nicht, um die in Abschnitt 2.1.3 zusammengefasste Kritik am Strukturalismus nach Harris zu widerlegen, doch kann der dort vorgebrachte Einwand Chomskys (vgl. Seite 25) zumindest teilweise angezweifelt werden: Die untersuchten Korpora (bzw. Korpus-Ausschnitte) enthielten lediglich 15.000 bis 40.000 Sätze, dennoch konnten Ergebnisse generiert werden, die eine weitere wissenschaftliche Auseinandersetzung mit Verfahren, die auf Harris Arbeiten basieren, sinnvoll erscheinen lassen. Die Untersuchungen zeigten jedoch, dass das den hier verwendeten Verfahren zugrundeliegende Repräsentationsmodell unzureichend ist, um anhand komplexer Einzelbeispiele ein abstraktes, allgemein gültiges Sprachmodell zu generieren, so dass ihre erfolgreiche Anwendung außerhalb syntaktisch einfacher und inhaltlich kompakter *Sublanguages* unwahrscheinlich ist.

Wie bereits in der Einleitung zu diesem Kapitel erwähnt, kann keiner der untersuchten Ansätze den F-Score von *Right* erreichen. Zwei Gründe für die vergleichsweise guten Ergebnisse dieser Strategie wurden bereits in Abschnitt 5.2 diskutiert: Im Gegensatz zu *Right* verwenden die Alignment-Verfahren keine (expliziten oder impliziten) Annahmen

über die Struktur der zu untersuchenden Daten, sondern operieren ausschließlich auf Basis von Symbolvergleichen auf Wortebene. Da dabei weder morphosyntaktische Eigenschaften berücksichtigt werden noch eine morphologische Analyse angewendet wird[185], können die in Abschnitt 2.1 beschriebenen Analysemethoden nach Harris zwangsläufig nicht vollständig abgebildet werden.

Mit ABL wurde – trotz der hier vorgestellten Erweiterungen – lediglich ein einzelner Ansatz zur unüberwachten Extraktion syntaktischer Strukturen untersucht; alternative Ansätze wie etwa das in Solan *et al.* (2003b) beschriebene ADIOS oder das in Bod (2006) vorgestellte *Data-Oriented Parsing* (DOP) wurden nicht evaluiert. Der letztgenannte Algorithmus verfolgt im Gegensatz zu ABL einen Top-Down-Ansatz: Statt (durch Alignment) auf individuelle Strukturgrenzen zu schließen, erzeugt der Algorithmus zunächst sämtliche möglichen Repräsentationen der syntaktischen Struktur eines Satzes, um anschließend aus diesen die wahrscheinlichste Struktur auszuwählen. Bod (2006) berichtet von hervorragenden Ergebnissen bei Anwendung des Verfahrens auf einen Ausschnitt des *Wall Street Journals*, weist jedoch auch darauf hin, dass die Qualität der Ergebnisse stark von der Länge der untersuchten Sätze (und der damit exponentiell steigenden Menge potentieller Baumstrukturen, vgl. Abschnitt 5.2) abhängt.[186] Diesen Ansatz mit ABL zu vergleichen erscheint daher sinnvoll, zumal – sollten sich die Ergebnisse aus Bod (2006) reproduzieren lassen – sich daraus u.U. eine Verbesserung der oben erwähnten Bewertungfunktionen konstruieren ließe.

Eine Ausweitung der untersuchten Verfahren durch Berücksichtigung von ADIOS erscheint ebenfalls sinnvoll. Im Unterschied zu ABL und DOP werden Korpora hier als Graphen interpretiert, wobei jeder Satz als Pfad von Wort-Knoten abgebildet wird (vgl. Solan *et al.* 2003b). Häufig vorkommende Wortfolgen erscheinen im Graphen als stark frequentierte Teilpfade, anhand derer auf Muster geschlossen werden kann, aus denen sich wiederum Hypothesen generieren lassen. Aus technischer Sicht weist dieser Ansatz starke Ähnlichkeit zur Analyse von N-Gramm-Bäumen auf – ohne genauere Analyse kann jedoch nur darüber spekuliert werden, ob bzw. wie sich die Ergebnisse von ADIOS von den hier ermittelten Daten unterscheiden.

[185] Dies ist nicht notwendigerweise ein Widerspruch zum Verzicht auf die Verwendung linguistischen oder sprachabhängigen Vorwissens: In beiden Fällen könnte ausschließlich mit statistischen Methoden vorgegangen werden, wie in Benden (2004) zur Detektion von Morphemgrenzen beschrieben. Der dort vorgestellte Ansatz wurde bereits durch Hermes (2011, vgl. Abschnitt 6.2) in Form einer Teslakomponente umgesetzt, so dass ein Großteil der Komponenten, die für diesbezügliche Experimente benötigt werden, bereits vorhanden ist.

[186] Bei Beschränkung der maximalen Satzlänge auf 10 Wörter erreicht das Verfahren einen F-Score von fast 83. Wird die maximale Satzlänge auf 40 Wörter erweitert, sinkt der F-Score hingegen auf 66,4.

Die in Abschnitt 5.5 beschriebene Verwendung von Alignment-Verfahren für die Kategorisierung von Wörtern konnte zeigen, dass die Anwendungsmöglichkeiten über die Detektion von Strukturhypothesen hinausgehen: Im Gegensatz zu den in Abschnitt 5.4.1 und 5.4.2 durchgeführten Experimenten wurden hier keine weiteren Optimierungen durchgeführt – es ist daher zu vermuten, dass die Qualität der Ergebnisse weiter verbessert werden kann. Die durchgeführten Experimente veranschaulichen zudem, dass Alignment-Verfahren deutlich flexibler einsetzbar sind als etwa die zur syntagmatischen Strukturierung verwendete *Right*-Strategie.

Neben der Analyse und Evaluation verschiedener Alignment-Verfahren war auch die Entwicklung und Anwendung geeigneter Evaluationsformen selbst ein weiterer Schwerpunkt dieses Kapitels. In diesem Zusammenhang sind die unterschiedlichen zufallsbasierten Vergleichsverfahren zu erwähnen, die sich automatisch an individuelle Merkmale der untersuchten Verfahren anpassen und so eine empirisch ermittelte untere Schranke bereitstellen, die eine Interpretation der Ergebnisse vereinfacht. Randomisierte Verfahren konnten nicht nur zur Beurteilung von Precision und Recall bei Verfahren zur Detektion syntaktischer Strukturgrenzen, sondern auch für die Bewertung morphosyntaktischer und semantischer Kategorisierung eingesetzt werden. Es soll erneut darauf hingewiesen werden, dass die Vergleichsverfahren nicht völlig unabhängig von den zu untersuchenden Verfahren arbeiten (und bspw. die durchschnittliche Anzahl von Strukturhypothesen übernehmen, um eine vergleichbare Menge von Auszeichnungen zu generieren) und daher zwangsläufig bessere Ergebnisse liefern als ein ausschließlich zufallsbasiert arbeitendes Verfahren. Im Rahmen der hier durchgeführten Analysen wäre die Interpretation der Ergebnisse ohne ein sich automatisch anpassendes *Random*-Verfahren jedoch nur stark eingeschränkt möglich, wie u.a. anhand der in Abschnitt 5.4.2 durchgeführten Experimente deutlich wird, in denen *Select* sowohl auf von Alignment-Verfahren generierte Hypothesenmengen als auch auf zufällig erzeugte, quantitativ vergleichbare Mengen angewendet wurde: Hier konnte verdeutlicht werden, dass Alignment-Verfahren, die nicht hinsichtlich der minimalen Kontextbreite beschränkt werden, sich nur unwesentlich von zufallsbasierten Verfahren abgrenzen. Gerade dann, wenn die Resultate der untersuchten Verfahren teilweise nur gering von den Vergleichswerten abweichen, kann ein adaptives Verfahren helfen, die Signifikanz einer Abweichung zu erkennen – so wäre es u.U. sogar sinnvoll, die Vergleichsverfahren stärker an die analysierten Verfahren anzupassen.

Die Referenzen, die für die Auswertung der Experimente genutzt wurden, veranschaulichen das bereits in Kapitel 1 erwähnte Problem, (Gold-) Standards oder Daten zu nutzen,

die eine Evaluation anhand möglichst objektiver Kriterien ermöglichen: Wie bereits in Abschnitt 5.1 beschrieben, sind die strukturellen Auszeichnungen in den manuell annotierten Korpora unvollständig, und der Vergleich mit den vom Berkeley Parser generierten Strukturen ist kein adäquater Ersatz, sondern lediglich ein Kompromiss, durch den eine automatisierte Auswertung der Ergebnisse ermöglicht werden kann. Gleiches gilt für die Bewertung semantischer Ähnlichkeit durch WordNet: Auch hier kann nicht von einem Gold-Standard (im eigentlichen Sinne) gesprochen werden, entsprechend sind die Ergebnisse ebenfalls kritisierbar. Dieses Problem wurde bereits in Kapitel 1 erwähnt, eine zufriedenstellende Lösung kann jedoch nicht angeboten werden: Selbst bei manueller Auszeichnung der Korpora wäre das zugrundeliegende Problem nicht vollständig gelöst, da die Evaluationmöglichkeiten so auf wenige Korpora eingeschränkt würden – die Analysen haben jedoch gezeigt, dass die Qualität der erzielten Ergebnisse stark von den verwendeten Korpora abhängt, so dass es unvermeidlich ist, auf algorithmisch erzeugte Referenzen (wie die durch einen Parser generierte Strukturauszeichnung oder die von einem POS-Tagger durchgeführte Kategorisierung) zurückzugreifen.

Losgelöst vom theoretischen Hintergrund und dem Ziel einer Strukturaufdeckung nach linguistischen Maßstäben kann allerdings die Frage diskutiert werden, ob die hier genutzte Form der Evaluierung grundsätzlich den untersuchten Verfahren genügt, bzw. ob sie der Art des extrahierten Wissens entspricht: Die durchgeführten Experimente belegen, dass sich durch Alignment-Verfahren Strukturen extrahieren lassen, die den manuell hinzugefügten ebenso wie den vom Berkeley Parser erzeugten Strukturen häufiger entsprechen als zufällig generierte Strukturen. Gleiches gilt, wie in Abschnitt 5.5 gezeigt, hinsichtlich morphosyntaktischer und semantischer Kategorisierung. Daraus lässt sich schließen, dass die Verfahren potentiell dazu in der Lage sind, sinnvolle Strukturierungen durchzuführen – diese müssen jedoch nicht zwangsläufig vollständig den Regeln einer linguistisch motivierten Strukturierung bzw. Kategorisierung folgen und könnten daher mit alternativen Evaluationsmethoden besser untersucht werden.

So evaluieren bspw. Horn *et al.* (2004) das bereits erwähnte ADIOS dadurch, dass das System die Grammatikalität von Sätzen beurteilen muss, während Stehouwer & van Zaanen (2010b) ein N-Gramm-Verfahren zur Detektion und Korrektur von Wortstellungsfehlern nutzen und dessen Korrekturqualität messen. Weitere Möglichkeiten der Evaluation werden in Clark & Lappin (2010) diskutiert. Vorteil einer alternativen Evaluation wäre zudem, dass ähnliche Verfahren besser verglichen werden könnten – wie in Abschnitt 5.1 diskutiert, ist bspw. die Entscheidung darüber, ob triviale Strukturen bei der Evaluation berücksichtigt werden, maßgeblich am Ergebnis beteiligt. Abschnitt 5.1

zeigte jedoch auch, dass Prä- und Postprozessierung der Daten ebenfalls äußerst relevant sind und beim Vergleich unterschiedlicher Verfahren standardisiert werden müssen: Wie bereits in Kapitel 1 erwähnt, erhält bspw. Cramer (2007) bei der Anwendung von ABL Ergebnisse, die sich nur gering vom Erwartungswert abheben. Das dort verwendete Korpus besteht aus ca. 7.000 Sätzen mit durchschnittlich je 20 Wörtern, so dass die Ergebnisse im Rahmen der hier untersuchten Korpora TüBa-D/Z und BNC-Guardian liegen sollten. Tatsächlich liegt jedoch der F-Score des *Right*-Verfahrens dort lediglich bei 4,8 – dies legt die Vermutung nahe, dass in den dort beschriebenen Versuchen ein Konfigurationsfehler enthalten sein könnte, durch den die Ergebnisse negativ beeinflusst werden.[187]

In diesem Kapitel wurde ein Beispiel für eine Form des experimentellen, computerlinguistischen Arbeitens gegeben: Zunächst wurde ein bereits bekanntes Verfahren auf bisher nicht untersuchte Korpora angewandt – die einzelnen Verarbeitungsschritte wurden dabei nicht nur detailliert analysiert sondern auch so dokumentiert, dass sie jederzeit reproduziert werden können. Aus Analyse und Evaluation ergaben sich neue Fragestellungen und Hypothesen, denen in weiteren Experimenten nachgegangen wurde - teilweise wurden dafür zusätzliche Komponenten entwickelt, teilweise wurden lediglich Parameter variiert oder Verarbeitungsketten neu geordnet. Das Resultat dieser Arbeiten bestand in einem verbesserten Verfahren zur Detektion syntagmatischer Strukturen, das schließlich hinsichtlich seiner Eignung zur Anwendung in weiteren computerlinguistischen Bereichen evaluiert wurde, wobei auch hier ein exaktes Protokoll (in Form von Experiment-Definitionen, siehe Anhang C) angefertigt wurde, durch das eine Reproduktion der Ergebnisse möglich ist.

Der dritte Schwerpunkt, der in diesem Kapitel gesetzt wurde, lag darauf, Tesla als computerlinguistisches Labor zu evaluieren – der beschriebene Arbeits- und Analyseprozess konnte nicht nur vollständig in Tesla umgesetzt werden, es wäre vielmehr äußerst schwierig gewesen, ihn ohne Tesla umzusetzen, ohne gleichzeitig Kompromisse eingehen zu müssen: Keines der in Kapitel 3 beschriebenen Frameworks bietet den notwendigen Funktionsumfang, mit dem die Anforderungen an Flexibilität, Erweiterbarkeit, Analysierbarkeit, Evaluation und Reproduktion, die in diesem Kapitel deutlich gemacht werden konnten,

[187] Da der von Cramer (2007) durchgeführte Versuch in Ermangelung der genauen Beschreibung des Versuchsaufbaus nicht reproduziert werden kann, ist dies lediglich eine Spekulation. Bei den in der vorliegenden Arbeit durchgeführten Experimenten wurde zwischenzeitlich jedoch ein ähnliches Ergebnis des Right-Verfahrens erzielt, das auf eine unvollständige Präprozessierung (Berücksichtigung von Interpunktionssymbolen) zurückzuführen war.

erfüllt werden können. An dieser Stelle kann daher bereits das Fazit gezogen werden, dass Tesla die Voraussetzungen an eine Laborumgebung für computer- und korpuslinguistische Forschung und Entwicklung erfüllt – zusammen mit weiteren Aspekten wird dies im folgenden Kapitel jedoch noch einmal separat diskutiert.

6 Fazit: Tesla als virtuelle Forschungs- und Entwicklungsumgebung

> I do not think there is any thrill that can go through the human heart like that felt by the inventor as he sees some creation of the brain unfolding to success... Such emotions make a man forget food, sleep, friends, love, everything.
>
> *(Nikola Tesla)*

Ein wesentliches Ziel dieser Arbeit war es, die Architektur eines computerlinguistischen Komponentenframeworks zu konzipieren, zu implementieren und anhand eines konkreten Anwendungsfalls zu evaluieren. Wie bereits in Abschnitt 5.6 skizziert, veranschaulichen die in Kapitel 5 durchgeführten Experimente, dass dies erreicht werden konnte. Im Folgenden wird anhand einzelner Aspekte des Frameworks noch einmal verdeutlicht, inwiefern die in Tesla umgesetzten Konzepte eine notwendige Voraussetzung für die in Kapitel 5 verfolgte experimentelle, empirische Forschung in Computer- und Korpuslinguistik sind, zudem wird in Ausblick gestellt, wie diese Konzepte weiterentwickelt werden können.

6.1 Tesla als Komponentensystem

In Kapitel 3 wurde an den Systemen GATE, UIMA und TextGrid bemängelt, dass diese nicht den in Abschnitt 2.3 aufgeführten Anforderungen an ein Komponentensystem zum experimentellen, laborientierten Arbeiten mit neuen computerlinguistischen Verfahren, Hypothesen und/oder Korpora entsprechen – insbesondere wurde dabei kritisiert, dass die Schnittstellen, die den Datenfluss zwischen verschiedenen Komponenten definieren, zu restriktiv sind und dem Paradigma der Objektorientierung dahingehend widersprechen, dass beispielsweise die Aggregation von Methoden und Datentypen in Form von Klassen oder Interfaces nur eingeschränkt möglich ist. Ferner wurde kritisiert, dass komplexe

Objekte wie Graphen oder Matrizen nur mit großem Aufwand als Austauschformat genutzt werden können, was der Modularität des Komponentenkonzepts gegenübersteht. Um diese Probleme zu umgehen, wurde das in Abschnitt 4.1.4 beschriebene Konzept des *Tesla Role System* entwickelt – in Kapitel 5 konnte dessen Leistungsfähigkeit demonstriert werden, zugleich wurde (u.a. anhand der Komponenten *N-Gram Tree Generator*, *Hypothesis Generator* und *Word Vector Generator*) deutlich, dass die Umsetzung komplexer Komponenten erst durch die objektorientierten, interfacebasierten Schnittstellen des TRS möglich wird.

Die Experimente veranschaulichen, wie das TRS die Wiederverwertbarkeit von Komponenten ermöglicht – dass diese dabei auch über einzelne Projekte hinausgehen kann, zeigt sich durch Hermes (2011, Abschnitt 6.3), wo der im Rahmen dieser Arbeit entwickelte *N-Gram Tree Generator* für eine Analyse möglicher Dechiffrierungen des Voynich-Manuskripts eingesetzt wird. Als weiteres Beispiel ist die in Abschnitt 5.5.1 verwendete Komponente zur Berechnung des *Purity*-Maßes zu nennen, die ursprünglich im Rahmen einer Hausarbeit über dokumentbasiertes Clustering entwickelt wurde[188], hier jedoch dafür eingesetzt werden konnte, die Evaluation von Substitutionsregeln anhand morphosyntaktischer Eigenschaften zu ermöglichen. Tesla wird zudem seit April 2010 im Rahmen einer Kooperation von Bioinformatik und Informationsverarbeitung eingesetzt, um u.a. in der Bioinformatik etablierte Verfahren auf korpuslinguistischen Problemstellungen zu übertragen und um mit computerlinguistischen Verfahren uneinheitliche Genbezeichnungen einander zuzuordnen[189]. Bereits in Kapitel 1 wurde festgehalten, dass eine Beschleunigung des Forschungsbetriebes im *Text Engineering* wünschenswert ist – die drei Beispiele zeigen, wie Tesla dazu beitragen kann.

Anhand der in Kapitel 5 verwendeten Komponenten zeigt sich weiterhin, wie das TRS Kompatibilität und Standardisierung fördert: Im Rahmen der Beschreibung des Komponentenmodells von GATE wurde die von Cunningham & Bontcheva (2006) geäußerte Hypothese festgehalten, dass sich unterschiedliche Ressourcen kaum standardisieren lassen, da sie inkompatible Abfragesprachen verwenden (vgl. Seite 48 dieser Arbeit). In Abschnitt 5.5 konnte dies anhand der Verwendung von WordNet zur Berechnung konzeptueller Ähnlichkeit wiederlegt werden: Entscheidend ist, dass ein Komponentensystem die Möglichkeit bieten muss, nicht nur Datenstrukturen zu modellieren, sondern auch die Art, in der auf diese Strukturen zugegriffen wird. Das TRS ermöglicht es, fein granulierte Schnittstellen zu entwerfen, für deren Implementation beliebige Verfahren

[188] Siehe http://www.spinfo.phil-fak.uni-koeln.de/geduldiga.html
[189] Siehe Projekt-Website unter http://www.spinfo.phil-fak.uni-koeln.de/tiw.html

und Ressourcen verwendet werden können.

Die hier zusammengefassten Aspekte zeigen, dass die Möglichkeiten, die sich aus dem TRS ergeben, maßgeblich dazu beitragen, den Anforderungen an ein virtuelles Labor gerecht zu werden: Die mit dem TRS ermöglichte Offenheit in Bezug auf Datenstrukturen führt zu hoher Modularität, aus der sich eine erhöhte Kompatibilität und damit eine verbesserte Wiederverwertbarkeit von Komponenten ergibt.

6.2 Tesla als Entwicklungsumgebung

Im Rahmen dieser Arbeit wurden zahlreiche Komponenten implementiert und/oder verwendet, die sich hinsichtlich der produzierten Daten sowie der dafür geeigneten Zugriffsmethoden teilweise deutlich unterschieden. Dafür war nicht nur das im vorherigen Abschnitt hervorgehobene TRS eine notwendige Voraussetzung, sondern auch der in Abschnitt 4.2.2 beschriebene Annotationsgraph, der ein hohes Maß an Abstraktion erlaubt und Entwicklern Wahlfreiheit in Bezug auf Persistenzmechanismen ermöglicht. So enthalten bspw. die vom *Hypothesis Generator* erzeugten Strukturhypothesen detaillierte Informationen zu sämtlichen Vorkommen identischer Kontexte, weshalb zahlreiche Optimierungen notwendig waren, um den benötigten Speicherbedarf zu minimieren – mit einem Framework, das die Wahl von Datenstrukturen eingrenzt, hätte dies nicht umgesetzt werden können.

Die Entscheidung, dass die von einer Komponente erzeugten Annotationen stets persistiert werden, kann als Nachteil des Systems interpretiert werden, da dies das Laufzeitverhalten verschlechtert und den Bedarf an Festplatten-Speicherplatz erhöht – für eine Laborumgebung ist dies jedoch zwingend notwendig, da sämtliche Daten für Analyse und Evaluation zur Verfügung stehen müssen. Die in Abschnitt 4.2.1.3 beschriebene *TunguskaDB* zeigt zudem, wie eine für linguistische Komponentensysteme entwickelte Datenbank eine effiziente Persistenzlösung darstellen kann, so dass dieser Kritikpunkt abgeschwächt wird. Wie in Abschnitt 4.1.2 beschrieben, werden die von einer Komponente erzeugten Daten zudem wiederverwendet, falls diese mit identischer Konfiguration in einem (an anderer Stelle variierten) Experiment ausgeführt werden soll – dies wirkt sich nicht nur dann positiv auf die Laufzeit aus, wenn, wie in Kapitel 5 der Fall, Experimente durchgeführt werden, bei denen lediglich einigen Komponenten neue Parameter zugewiesen wurden, sondern auch dann, wenn bspw. eine neue Komponente entwickelt wird und innerhalb eines Experiments getestet werden soll.

Architektur und Konzeption von Tesla unterscheiden sich in einem weiteren Punkt von den in Kapitel 3 beschriebenen Systemen: Die in Tesla verfolgte Strategie ist vergleichweise stark auf die Entwicklung von Software fokussiert. Dies zeigt sich auch darin, dass Teslas Client in die Entwicklungsumgebung Eclipse integriert wurde, und dass dort ebenfalls ein voll funktionsfähiger Server zur Verfügung gestellt wird (vgl. Abschnitt 4.1.7.2). Neben der Möglichkeit, den Aufruf einer Methode (nicht nur bei Komponenten, sondern auch bei DataObject- und AccessAdapter-Implementationen) in einem beliebigen Experiment zwecks Fehlersuche zu unterbrechen, kann bspw. über den *Components View* unmittelbar auf den Quellcode einer Komponente zugegriffen werden, um etwa die Funktionsweise einer Komponente besser nachvollziehen zu können. Wie in Abschnitt 4.1.7.2 beschrieben, ist Tesla ist nicht nur ein Komponentensystem, sondern auch eine Entwicklungsumgebung, und kann je nach Anwendungsfall entsprechend eingesetzt werden.

Die aufgeführten Punkte verdeutlichen die in Tesla realisierte Unterstützung von Entwicklern: Der Annotationsgraph erlaubt die Umsetzung und Kapselung komplexer Verfahren und spezialisierter Annotationen, die IDE-Funktionalität vereinfacht die Entwicklung neuer Komponenten und Rollen und die Persistenzstrategie ermöglicht die Verwendung, effizienter, flexibler und jederzeit analysierbarer Datenstrukturen.

6.3 Tesla als Forschungsumgebung

Die in Kapitel 5 durchgeführten Experimente verdeutlichen die Anforderungen, die aus Anwenderperspektive von einem virtuellen Labor erfüllt werden müssen: So musste nicht nur gewährleistet werden, dass die Präprozessierung der zu untersuchenden Daten stets auf identische Weise ausgeführt wird, vielmehr mussten auch unabhängige Verarbeitungsketten modelliert werden, in denen die präprozessierten Daten auf unterschiedliche Weise weiterverarbeitet werden, um schließlich wieder in einer Evaluationskomponente zusammenzufließen (vgl. bspw. Abbildung 5.5 auf Seite 168). Tesla ist nicht nur dazu in der Lage, derartige Experimente auszuführen, sondern stellt zudem auch eine graphische Oberfläche zur Verfügung, mit der Aufbau, Konfiguration und Verwaltung von Experimenten unterstützt wird.

In Kapitel 5 wurde bereits ausführlich gezeigt, wie sich automatische Evaluations- bzw. Auswertungsverfahren in Tesla umsetzen lassen; es ist jedoch ebenfalls notwendig, Daten manuell auswerten zu können oder in externen Anwendungen zu evaluieren oder aufzube-

reiten (etwa für eine graphische Darstellung). Mit den in Abschnitt 4.1.6 beschriebenen Export- und Visualisierungsoptionen des Tesla Clients wird dies ermöglicht: Dank der in Abschnitt 4.1.6 beschriebenen Mechanismen können beliebige Rollen flexibel serialisiert und transformiert werden.

Ein weiterer Aspekt, der von einer virtuellen Forschungsumgebung berücksichtigt werden muss, betrifft die Möglichkeit der Reproduktion von Ergebnissen. Wie bereits in Abschnitt 4.1.3 diskutiert, müssen dabei lizenzrechtliche Aspekte berücksichtigt werden – die in Abschnitt 4.1.3 beschriebene Technik zur eindeutigen Adressierung von Dokumenten durch Prüfsummen vereinfacht den Umgang mit urheberrechtlich geschütztem Material und stellt sicher, dass Experimente ohne weitere Modifikation reproduziert werden können, sofern die verwendeten Korpora zur Verfügung stehen. Prozessierte Daten, Datenfluss und Konfiguration der Komponenten sind durch ein Tesla-Experiment eindeutig und vollständig beschrieben, so dass dieses (in Verbindung mit den verwendeten Komponenten) als Ergänzung einer wissenschaftlichen Publikation verwendet werden kann, wovon sowohl in Hermes (2011) als auch in dieser Arbeit Gebrauch gemacht wird.

Dank der im Rahmen von Tesla umgesetzten, flexiblen Möglichkeit zur Definition von Abhängigkeiten zwischen Komponenten, den umfangreichen Funktionen zur Analyse und Evaluation untersuchter Verfahren sowie der Möglichkeit, Experimente zu reproduzieren, konnte die in Kapitel 5 beschriebene Form laborativer Forschung umgesetzt werden.

6.4 Kritik und Ausblick

Die Zusammenfassung einzelner Aspekte von Tesla hat gezeigt, dass ein wesentlicher Teil der erweiterten Möglichkeiten, die Tesla gegenüber den in Kapitel 3 beschriebenen Frameworks bietet, auf das *Tesla Role System* zurückzuführen ist. Das TRS erlaubt die Konzeption und die Entwicklung hoch spezialisierter Rollen und Komponenten, was sich u.a. an Komponentenbezeichnungen wie *Boundaries Detector* oder *Custom Choice Random Group Generator* zeigt – im Gegensatz zu etablierten Bezeichnungen wie *Tokenizer* oder *Sentence Splitter* werden Funktion und Verwendungszweck hier nicht unmittelbar deutlich. Sofern derartige Komponenten nicht ausführlich dokumentiert werden, kann dies die Anforderungen an Anwender (etwa gegenüber GATE) erhöhen. An GATE wurde kritisiert, dass u.U. der Quellcode einer Komponente analysiert werden muss, um herauszufinden, auf welche Attribute der *FeatureMap* zugegriffen wird (vgl. Seite 58 in Abschnitt

3.1.5) – analog dazu kann am TRS kritisiert werden, dass sich die Semantik eines DataObjects, wie etwa einer als *Context-aware Hypothesis* bezeichneten Strukturhypothese, die über zusätzliche Informationen, die zu ihrer Generierung geführt haben, verfügt, ebenfalls nicht unmittelbar erschließt.

Die beiden Beispiele unterscheiden sich jedoch hinsichtlich der Komplexität der zugrundeliegenden Annotationen: Der hohe Grad an Spezialisierung, der in einer Tesla-Rolle ausgedrückt werden kann, erschwert es u.U., die Bedeutung und die Funktionalität einer Rolle zu erfassen, und erfordert somit zwangsläufig mehr Einarbeitungszeit – in GATE ist es hingegen *per se* nicht möglich, eine vergleichbar komplexe Funktionalität umzusetzen.

Es ist somit unvermeidbar, dass Entwicklung für und Verwendung von Tesla teilweise höhere Anforderungen stellen, als etwa bei GATE der Fall – allerdings zeigte sich auch, dass die mit dem TRS verbundene Komplexität bei der Definition neuer Rollen bei trivialen Datenstrukturen, wie den durch einen Tokenizer produzierten Datentypen, unnötig hoch ist, und hier ein vereinfachtes Verfahren, wie in UIMA durch die *Common Analysis Structure* umgesetzt, wünschenswert wäre. Auch zeigte sich, dass die Spezialisierung von Rollen durch abgeleitete Java-Interfaces detaillierte Kenntnis der Rollenhierarchie erforderte und oftmals mit einem Refactoring dieser Hierarchie verbunden war. Dies ist einerseits zwar positiv, da es dazu führt, dass Dokumentation, Konsistenz und Austauschbarkeit im TRS verbessert werden, bringt andererseits jedoch auch den Nachteil mit sich, dass die Einarbeitungszeit in das System stark von der Komplexität des genutzten Rollensystems abhängt. Analog zu den in Abschnitt 4.1.7.2 beschriebenen Erweiterungen der Eclipse-IDE könnte die Unterstützung von Entwicklern und Anwendern daher weiter ausgebaut werden; die Integration eines Typsystems für einfache linguistische Datenstrukturen (wie dem in Abschnitt 3.2.1 beschriebenen *UIMA Type System*) könnte ebenfalls dazu beitragen, die Verwendung des TRS zu vereinfachen. Die dafür notwendige Flexibilität ist vorhanden: Mit Ausnahme der Systemrolle *Annotator* sind alle weiteren Rollen optional, so dass sich das TRS vollständig an eigene Anforderungen und Anwendungskontexte anpassen lässt (vgl. Abschnitt 4.1.4.1).

Im Rahmen der Weiterentwicklung von Tesla wäre zu untersuchen, wie Tesla bspw. mit UIMA interagieren kann. So könnte überprüft werden, wie weit Teile von UIMA in das System übernommen werden können, um die Integration von Komponenten, die nicht von den Vorteilen des TRS profitieren, zu vereinfachen, oder um ein Mapping zwischen Teslas Annotationsmodell und JCas umzusetzen.

Auch in Bezug auf TextGrid könnte eine Erweiterung sinnvoll sein: In Abschnitt 3.3 u.a. die elaborierte Nutzer- und Rechteverwaltung von TextGrid vorgestellt, gleichzeitig

wurde jedoch auch bemerkt, dass die bisher in das System integrierten Werkzeuge (aus computerlinguistischer Sicht) nur grundlegende Aufgaben erfüllen. Eine Schnittstelle zwischen Tesla und TextGrid würde beiden Systemen zugute kommen: Wenn Experimente in Tesla unmittelbar auf TextGrid-Korpora angewandt werden könnten und umgekehrt die Ergebnisse solcher Experimente (etwa Form von Web Services) an TextGrid zurückgegeben werden könnten, würde dies den bereits in Kapitel 1 erwähnten Austausch zwischen computerlinguistischen Forschern und geisteswissenschaftlichen Anwendern vereinfachen und den Wissenstransfer optimieren und beschleunigen.

Unter softwaretechnologischen und -architektonischen Gesichtspunkten wäre es u.a. sinnvoll, die Komponentenschnittstelle in Tesla so zu modifizieren, dass eine parallele Prozessierung der Daten vereinfacht wird. Hier wäre eine Adaption des in Dean & Ghemawat (2004) beschriebenen *MapReduce*-Algorithmus denkbar, der ein einfaches Programmiermodell mit hoher Skalierbarkeit und Ausfallsicherheit kombiniert, und der erfolgreich für diverse Projekte der Entwickler eingesetzt wird (vgl. ebd.). Zu Beginn der Entwicklung von Tesla lagen noch keine frei verfügbaren Implementierungen des Konzepts vor, inzwischen könnte jedoch bspw. das Framework *Hadoop*[190] der *Apache Foundation* integriert werden.

Schließlich ist auch eine Erweiterung der laborativen und kooperativen Aspekte wünschenswert: So wäre zu überlegen, wie ein Anschluss an Community Plattformen realisiert werden kann (vgl. Abschnitt 4.1.1), um den Austausch von Komponenten, Experimenten und Rollen (-systemen) zu vereinfachen. Die mehrschichtige Client-Server-Architektur, mit der Tesla realisiert wurde, ermöglicht jedoch bereits jetzt ein kollaboratives Arbeiten innerhalb einer Forschergruppe.

Ziel dieser Arbeit war es, ein virtuelles Labor zu entwickeln, das für computer- und korpuslinguistische Forschung eingesetzt werden kann. Wie anhand der in Kapitel 5 durchgeführten Experimente praktisch demonstriert, konnte dieses Ziel umgesetzt werden. In diesem Kapitel wurde zusammengefasst, welche Aspekte Tesla zu einem virtuellen Labor machen. Nicht jeder Aspekt ist dabei ein Alleinstellungsmerkmal – es ist vielmehr die Kombination der Aspekte, die Tesla von den in Kapitel 3 diskutierten Systemen unterscheidet.

Wie bereits in Abschnitt 4.4 zusammengefasst, hat jedes der Frameworks individuelle Schwerpunkte und damit verbunden unterschiedliche Vor- und Nachteile: UIMA ist in produktiv eingesetzten Umgebungen das Mittel der Wahl, während TextGrid im

[190] Siehe http://hadoop.apache.org/.

Vergleich zu Tesla weniger spezialisiert ist und daher auch dort verwendet werden kann, wo fein granulierte, experimentelle Analysen von Texten und Verfahren nicht zwingend erforderlich sind. Die Funktionalität von GATE ist hingegen – wie die in Kapitel 5 durchgeführten Experimente und die Übersicht über die für Tesla verfügbaren Komponenten in Anhang B zeigen – mit Ausnahme der Möglichkeit, Annotationen auch manuell zu editieren, auch in Tesla enthalten. Wie jedoch zu Beginn dieses Abschnitts verdeutlicht wurde, ist das in GATE verwendete Annotationsmodell im Vergleich zum TRS weniger komplex, so dass (zumindest einfache) Komponenten dort mit weniger Lernaufwand umgesetzt werden können.

Jedes der untersuchten Frameworks hat einen geeigneten Anwendungszweck, umgekehrt existiert für jeden Anwendungszweck ein am besten geeignetes System – im Fall experimenteller computer- und korpuslinguistischer Forschung und Entwicklung handelt es sich dabei um das *Text Engineering Software Laboratory*.

A Evaluation

Im Folgenden sind die Ergebnisse der in Kapitel 5 beschriebenen Experimente tabellarisch zusammengefasst. Zwecks Vollständigkeit und zur besseren Vergleichbarkeit werden die dort bereits abgebildeten Tabellen hier erneut dargestellt. In Anhang A.1 sind alle Ergebnisse der in Abschnitt 5.4.1 beschriebenen Experimente zur Evaluation unterschiedlicher Alignment-Verfahren aufgeführt, Anhang A.2 enthält die vollständige Auswertung des in Abschnitt 5.4.2 beschriebenen *Select*-Verfahren. In Anhang A.3 sind schließlich alle Ergebnisse der in 5.4.3 beschriebenen Experimente dargestellt.

Um die Ergebnisse reproduzieren zu können, wurden die Experimentdefinitionen unter `http://www.spinfo.phil-fak.uni-koeln.de/sschwieb_thesis.html` abgelegt – dort stehen ebenfalls Tesla-Installationen zum Download für Windows, Mac OS X und Linux bereit, die alle hier verwendeten Komponenten enthalten. Nach erstem Start des Tesla Clients und erstmaligem Ausführen des Servers[191] können die Experimente in den Workspace des Clients kopiert, dort geöffnet und anschließend ausgeführt werden. Dies setzt jedoch voraus, dass die verwendeten Korpora installiert wurden – entsprechende Konfigurationsdateien sind zwar auf der oben genannten Webseite zu finden, die Korpora können jedoch aus rechtlichen Gründen nicht im Rahmen dieser Arbeit zugänglich gemacht werden. Unter `http://www.sfs.uni-tuebingen.de/corpora.shtml` kann allerdings der für Forschung und Lehre kostenfreie Zugriff auf die Korpora TüBa-D/S, TüBa-E/S und TüBa-D/Z beantragt werden, und über `http://childes.psy.cmu.edu/` kann das CHILDES-Korpus (ebenfalls kostenfrei) bezogen werden.

Für die erfolgreiche Ausführung der Experimente ist es zudem notwendig, den verfügbaren Arbeitsspeicher des Tesla-Servers zu erhöhen (im Rahmen dieser Arbeit wurden 4 GB RAM zugewiesen), je nach Leistungsfähigkeit des verwendeten PCs ist es zudem empfehlenswert, die Parallelprozessierung auf Komponentenebene zu deaktivieren[192], auch sei an dieser Stelle noch einmal betont, dass die Ausführung der Alignment-Verfahren *WF-B* und *All* mehrere Stunden Laufzeit in Anspruch nehmen kann.

[191] Siehe Tutorials unter `http://tesla.spinfo.uni-koeln.de/tutorials.html`.
[192] Dazu muss in der Experimentübersicht der Punkt *Execute Singleton* aktiviert werden.

A.1 Ergänzung zu Abschnitt 5.4.1

Im Folgenden sind sämtliche Ergebnisse der Evaluation von *Align*-Verfahren aus Abschnitt 5.4.1 aufgeführt (vgl. auch Anhang C.3).

A.1.1 Evaluation von Alignment-Verfahren am TüBa-D/S

	Hypothesen	Precision	Recall	F-Score
Gold	117.865	100	100	100
Gold Random	118.509	9,55	9,6	9,58
Berkeley	206.403	36,27	63,52	46,18
Left	210.105	4,11	7,33	5,27
Right	210.105	**22,14**	39,46	**28,36**
Random L/R/BP	210.105	9,43	16,81	12,08
WF-B	1.022.090	9,25	80,21	16,59
Random WF-B	1.019.749	8,43	72,89	15,1
All	1.211.620	8,69	89,39	15,85
Random All	1.212.222	8,11	83,45	14,79
Suffix	**1.253.541**	8,56	**91,02**	15,65
Suffix Random	1.252.679	8,04	85,43	14,69

Tabelle A.1: Evaluation von Alignment-Verfahren am TüBa-D/S.

	Hypothesen	Precision	Recall	F-Score
Berkeley	206.403	100	100	100
Left	210.105	6,89	7,01	6,95
Right	210.105	**55,24**	56,23	**55,73**
Random L/R/BP	210.105	15,16	15,46	15,33
WF-B	1.022.090	17,69	87,58	29,43
Random WF-B	1.019.749	14,19	70,11	23,6
All	1.211.620	15,96	**93,67**	27,27
Random All	1.212.222	13,82	81,17	23,62
Suffix	**1.253.541**	15,58	94,6	26,75
Suffix Random	1.252.679	13,74	83,4	23,6

Tabelle A.2: Auswertung anhand der vom Berkeley Parser generierten Strukturen.

A.1.2 Evaluation von Alignment-Verfahren am TüBa-E/S

	Hypothesen	Precision	Recall	F-Score
Gold	99.431	100	100	100
Gold Random	101.109	7,94	8,07	8,01
Berkeley	173.825	38,71	67,67	49,25
Left	174.417	3,21	5,64	4,09
Right	174.417	**31,2**	54,73	**39,75**
Random L/R/BP	174.417	8,02	14,08	10,22
WF-B	1.011.373	8,9	90,53	16,21
Random WF-B	1.012.329	7,43	75,61	13,53
All	1.182.044	8,06	95,74	14,86
Random All	1.182.868	7,29	86,81	13,45
Suffix	**1.219.879**	7,87	**96,57**	14,56
Suffix Random	1.208.232	7,25	88,86	13,4

Tabelle A.3: Evaluation der untersuchten Verfahren anhand der im TüBa-E/S annotierten Strukturen.

	Hypothesen	Precision	Recall	F-Score
Berkeley	173.825	100	100	100
Left	174.417	14,17	14,22	14,2
Right	174.417	**40,65**	40,79	**40,72**
Random LR/BP	174.417	16,51	16,57	16,54
WF-B	1.011.373	15,29	88,99	26,1
Random WF-B	1.012.329	13,45	78,33	22,96
All	1.182.044	13,97	95	24,36
Random All	1.182.868	13	88,56	22,67
Suffix	**1.219.879**	13,67	**95,94**	23,93
Suffix Random	1.208.232	12,9	90,45	22,58

Tabelle A.4: Vergleich der untersuchten Verfahren anhand der vom Berkeley Parser im TüBa-E/S detektierten Strukturen.

A.1.3 Evaluation von Alignment-Verfahren am TüBa-D/Z

	Hypothesen	Precision	Recall	F-Score
Gold	317.850	100	100	100
Gold Random	318.031	9,11	9,12	9,12
Berkeley	475.782	44,95	67,28	53,9
Left	475.948	7,29	10,91	8,74
Right	475.948	**14,39**	21,54	**17,25**
Random L/R/BP	475.948	9,11	13,64	10,92
WF-B	1.766.190	9,27	51,49	15,71
Random WF-B	1.766.182	7,98	44,35	13,53
All	2.374.955	8,81	65,82	15,54
Random All	2.375.019	7,6	56,8	13,41
Suffix	**2.469.203**	8,65	**67,23**	15,33
Suffix Random	2.468.898	7,55	58,63	13,37

Tabelle A.5: Evaluation der untersuchten Verfahren anhand der im TüBa-D/Z annotierten Strukturen.

	Hypothesen	Precision	Recall	F-Score
Berkeley	475.782	100	100	100
Left	475.948	6,59	6,6	6,6
Right	475.948	**38,65**	38,66	**38,65**
Random L/R/BP	475.948	11,88	11,89	11,89
WF-B	1.766.190	15,58	57,84	24,55
Random WF-B	1.766.182	10,86	40,3	17,1
All	2.374.955	14,12	**70,51**	23,53
Random All	2.375.019	10,51	52,47	17,51
Suffix	**2.469.203**	13,84	71,8	23,2
Suffix Random	2.468.898	10,48	54,38	17,57

Tabelle A.6: Vergleich der untersuchten Verfahren mit den vom Berkeley Parser im TüBa-D/Z detektierten Strukturen.

A.1.4 Auswertung von Alignment-Verfahren am BNC-G

	Hypothesen	Precision	Recall	F-Score
Berkeley	488.094	100	100	100
Left	488.217	11,21	11,21	11,21
Right	488.217	**35,2**	35,2	**35,2**
Random L/R/BP	488.217	11,92	11,92	11,92
WF-B	2.256.646	14,36	66,38	23,61
Random WF-B	2.254.207	10,16	46,92	16,7
All	3.145.796	12,39	79,88	21,46
Random All	3.143.923	9,65	62,14	16,7
Suffix	**3.320.833**	11,97	**81,43**	20,87
Suffix Random	3.320.392	9,56	65,06	16,68

Tabelle A.7: Vergleich der untersuchten Verfahren mit den vom Berkeley Parser im BNC-G detektierten Strukturen.

A.2 Ergänzung zu Abschnitt 5.4.2

Die folgenden Tabellen enthalten Zusammenfassungen der in Abschnitt 5.4.2 durchgeführten Analyse des *Terms Select*-Verfahrens. In jeder Spalte ist der höchste Wert, der für ein analysiertes Verfahren erreicht wurde, hervorgehoben (vgl. auch Anhang C.4).

A.2.1 Evaluation von Select-Verfahren am TüBa-D/S

	Hypothesen	Precision	Recall	F-Score
Gold	117.865	100	100	100
Berkeley	206.403	36,27	63,52	46,18
Right	210.105	22,14	39,46	28,36
Select WF-B	109.495	**23,88**	22,19	23
Select Random WF-B	104.404	22,04	19,52	20,71
Select All	128.651	23,41	25,55	24,43
Select Random All	134.772	21,83	24,95	23,28
Select Suffix	137.234	23,06	**26,85**	**24,81**
Select Suffix Random	142.007	21,72	26,17	23,73
Random Non-Cross.	137.259	12,78	14,88	13,75

Tabelle A.8: Evaluation nach der Select-Phase anhand der Strukturen des TüBa-D/S.

	Hypothesen	Precision	Recall	F-Score
Berkeley	206.403	100	100	100
Right	210.105	55,24	56,23	55,73
Select WF-B	109.495	**31,77**	16,86	22,03
Select Random WF-B	104.404	25,75	13,02	17,3
Select All	128.651	30,09	18,76	23,11
Select Random All	134.712	26,06	17,01	20,58
Select Suffix	137.234	29,69	**19,74**	**23,72**
Select Suffix Random	142.047	26,11	17,97	21,29
Random Non-Cross.	137.259	20,95	13,93	16,74

Tabelle A.9: Auswertung der Verfahren *WF-B*, *All* und *Suffix* nach der Select-Phase. Als Referenz dienen die vom Berkeley Parser erzeugten Strukturen im TüBa-D/S.

A.2.2 Evaluation von Select-Verfahren am TüBa-E/S

	Hypothesen	Precision	Recall	F-Score
Gold	99.431	100	100	100
Berkeley	173.825	38,71	67,67	49,25
Right	174.417	31,2	54,73	39,75
Select WF-B	94.099	**25,16**	23,82	24,47
Select Random WF-B	95.296	19,95	19,12	19,53
Select All	117.922	23,16	27,47	25,13
Select Random All	124.214	20,37	25,45	22,63
Select Suffix	126.589	23,06	**29,36**	**25,83**
Select Suffix Random	131.733	20,54	27,21	23,41
Random Non-Cross.	128.329	11,32	14,61	12,75

Tabelle A.10: Evaluation der Verfahren *WF-B*, *All* und *Suffix* nach der Select-Phase. Als Referenz dienen die manuell annotierten Strukturen im TüBa-E/S.

	Hypothesen	Precision	Recall	F-Score
Berkeley	173.825	100	100	100
Right	174.417	40,65	40,79	40,72
Select WF-B	94.099	**37,15**	20,11	26,1
Select Random WF-B	95.296	32,43	17,78	22,97
Select All	117.922	34,31	23,28	27,74
Select Random All	124.215	31,57	22,56	26,31
Select Suffix	126.589	34,02	**24,77**	**28,67**
Select Suffix Random	131.733	31,46	23,84	27,13
Random Non-Cross.	128.329	20,34	15,01	17,28

Tabelle A.11: Auswertung der Verfahren *WF-B*, *All* und *Suffix* nach der Select-Phase. Als Referenz dienen die vom Berkeley Parser erzeugten Strukturen im TüBa-E/S.

A.2.3 Evaluation von Select-Verfahren am TüBa-D/Z

	Hypothesen	Precision	Recall	F-Score
Gold	317.850	100	100	100
Berkeley	475.782	44,95	67,28	53,9
Right	475.948	14,39	21,54	17,25
Select Suffix	153.440	25,71	12,41	16,74
Select Suffix Random	115.132	19,82	7,18	10,54
Random Non-Cross.	151.927	11,37	5,44	7,36

Tabelle A.12: Evaluation der Verfahren *WF-B*, *All* und *Suffix* nach der Select-Phase. Als Referenz dienen die manuell annotierten Strukturen im TüBa-D/Z.

	Hypothesen	Precision	Recall	F-Score
Berkeley	475.782	100	100	100
Right	475.948	38,65	38,66	38,65
Select Suffix	153.440	30,12	9,71	14,69
Select Suffix Random	115.132	21,21	5,13	8,26
Random Non-Cross.	151.927	16,59	5,3	8,03

Tabelle A.13: Auswertung der Verfahren *WF-B*, *All* und *Suffix* nach der Select-Phase. Als Referenz dienen die vom Berkeley Parser erzeugten Strukturen im TüBa-D/Z.

A.2.4 Auswertung von Select-Verfahren am BNC-G

	Hypothesen	Precision	Recall	F-Score
Berkeley	488.084	100	100	100
Right	488.217	35,2	35,21	35,2
Select Suffix	132.771	35,03	9,53	14,98
Select Suffix Random	140.428	24,3	6,99	10,86
Random Non-Cross.	128.739	15,32	4,04	6,39

Tabelle A.14: Auswertung der Verfahren *WF-B*, *All* und *Suffix* nach der Select-Phase. Als Referenz dienen die vom Berkeley Parser erzeugten Strukturen im BNC-G.

A.3 Ergänzung zu Abschnitt 5.4.3

In diesem Abschnitt sind die Ergebnisse der Evaluation unterschiedlicher *Suffix Select*-Verfahren aufgeführt. Der Versuchsaufbau entspricht dabei dem in Abschnitt 5.4.3 beschriebenen Vorgehen (vgl. auch Anhang C.4).

A.3.1 Evaluation von *Suffix Select* am TüBa-D/S

	Hypothesen	Precision	Recall	F-Score
Gold	117.865	100	100	100
Right	210.105	22,14	39,46	28,36
Select Width	184.780	17,54	27,51	21,42
Select W/O (Best)	173.582	20,77	**30,59**	**24,74**
Select W/O (All)	178.557	19,74	29,9	23,78
Select (Terms)	75.770	**26,09**	16,77	20,42
Random N.C. (Best)	175.809	12,06	17,99	14,44
Random N.C. (Terms)	76.064	13,75	8,87	10,79

Tabelle A.15: Evaluation des *Suffix Select*-Verfahrens anhand der im TüBa-D/S ausgezeichneten Strukturen.

	Hypothesen	Precision	Recall	F-Score
Berkeley	206.403	100	100	100
Right	210.105	55,24	56,23	55,73
Select Width	184.780	37,74	33,79	35,66
Select W/O (Best)	173.582	**47,1**	**39,61**	**43,03**
Select W/O (All)	178.557	44,88	38,83	41,63
Select (Terms)	75.770	45,76	16,8	24,57
Random N.C. (Best)	175.809	19,51	16,62	17,95
Random N.C. (Terms)	76.064	22,75	8,38	12,25

Tabelle A.16: Auswertung des *Suffix Select*-Verfahrens anhand der vom Berkeley Parser im TüBa-D/S detektierten Strukturen.

A.3.2 Evaluation von *Suffix Select* am TüBa-E/S

	Hypothesen	Precision	Recall	F-Score
Gold	99.431	100	100	100
Berkeley	173.825	38,71	67,67	49,25
Right	174.417	31,2	54,73	39,75
Select Width	160.449	23,33	37,64	28,8
Select W/O (Best)	152.553	26,45	40,58	32,03
Select W/O (All)	156.013	26,56	**41,67**	**32,44**
Select (Terms)	55.806	**35,08**	19,69	25,22
Random N.C. (Best)	152.903	11,06	17,01	13,41
Random N.C. (Terms)	56.437	12,18	6,91	8,82

Tabelle A.17: Evaluation des *Suffix Select*-Verfahrens anhand der im TüBa-E/S ausgezeichneten Strukturen.

	Hypothesen	Precision	Recall	F-Score
Berkeley	173.825	100	100	100
Right	174.417	40,65	40,79	40,72
Select Width	160.449	35,31	32,6	33,9
Select W/O (Best)	152.553	37,63	33,03	35,18
Select W/O (All)	156.013	38,64	**34,68**	**36,55**
Select (Terms)	55.806	**49,03**	15,74	23,83
Random N.C. (Best)	152.903	19,59	17,23	18,34
Random N.C. (Terms)	56.437	22,39	7,27	10,98

Tabelle A.18: Auswertung des *Suffix Select*-Verfahrens anhand der vom Berkeley Parser im TüBa-E/S detektierten Strukturen.

A.3.3 Evaluation von *Suffix Select* am TüBa-D/Z

	Hypothesen	Precision	Recall	F-Score
Gold	317.850	100	100	100
Berkeley	475.782	44,95	67,28	53,9
Right	475.948	14,39	21,54	17,25
Select Width	263.720	12,91	**10,71**	**11,71**
Select W/O (Best)	235.504	11,13	8,25	9,48
Select W/O (All)	243.302	11,55	8,84	10,02
Select (Terms)	216.315	**14,31**	9,74	11,59
Random N.C. (Best)	236.022	10,65	7,91	9,07
Random N.C. (Terms)	218.400	10,57	7,26	8,61

Tabelle A.19: Evaluation des *Suffix Select*-Verfahrens anhand der im TüBa-D/Z ausgezeichneten Strukturen.

	Hypothesen	Precision	Recall	F-Score
Berkeley	475.782	100	100	100
Right	475.948	38,65	38,66	38,65
Select Width	263.720	25,68	14,24	18,32
Select W/O (Best)	235.504	**28,98**	**14,34**	**19,19**
Select W/O (All)	243.302	27	13,81	18,27
Select (Terms)	216.315	26,91	12,23	16,82
Random N.C. (Best)	236.022	14,79	7,34	9,81
Random N.C. (Terms)	218.400	14,81	6,8	9,32

Tabelle A.20: Auswertung des *Suffix Select*-Verfahrens anhand der vom Berkeley Parser im TüBa-D/Z detektierten Strukturen.

A.3.4 Auswertung von *Suffix Select* am BNC-G

	Hypothesen	Precision	Recall	F-Score
Berkeley	488.094	100	100	100
Right	488.217	35,2	35,21	35,2
Select Width	318.148	26,04	**16,97**	20,55
Select WO (Best)	289.637	28,48	16,9	**21,21**
Select W/O (All)	298.062	27,53	16,81	20,88
Select (Terms)	191.295	**31,79**	12,46	17,9
Random N.C. (Best)	291.541	14,29	8,53	10,69
Random N.C. (Terms)	193.698	14,07	5,58	7,99

Tabelle A.21: Auswertung des *Suffix Select*-Verfahrens anhand der vom Berkeley Parser im BNC-G detektierten Strukturen.

A.3.5 Auswertung von *Suffix Select* am CHILDES

	Hypothesen	Precision	Recall	F-Score
Berkeley	33.224	100	100	100
Right	33.331	61,84	62,04	61,94
Select Width	29.101	53,79	47,11	50,23
Select W/O (Best)	26.905	**58,43**	47,32	52,29
Select W/O (All)	27.704	57,82	**48,22**	**52,58**
Select (Terms)	26.098	55,61	43,69	48,93
Random N.C. (Best)	27.540	36,87	30,56	33,42
Random N.C. (Terms)	27.540	36,87	30,56	33,42

Tabelle A.22: Ergebnis des Vergleichs der in Abschnitt 5.4.2 beschriebenen *Select*-Verfahren mit den vom Berkeley Parser detektierten Strukturen im CHILDES-Korpus. Die Analyse wurde auf 15.000 Sätze beschränkt, die je **drei bis sechs** Wörter enthalten.

	Hypothesen	Precision	Recall	F-Score
Berkeley	89.673	100	100	100
Right	89.923	57,2	57,36	57,28
Select Width	76.672	45,09	38,55	41,56
Select W/O (Best)	69.919	**51,01**	39,77	44,69
Select W/O (All)	73.177	49,79	**40,63**	**44,75**
Select (Terms)	45.235	47,99	24,21	32,18
Random N.C. (Best)	70.427	23,52	18,47	20,69
Random N.C. (Terms)	45.000	23,38	11,73	15,63

Tabelle A.23: Ergebnis des Vergleichs der in Abschnitt 5.4.2 beschriebenen *Select*-Verfahren mit den vom Berkeley Parser detektierten Strukturen im CHILDES-Korpus. Die Analyse wurde auf 15.000 Sätze beschränkt, die je **sieben bis zehn** Wörter enthalten.

	Hypothesen	Precision	Recall	F-Score
Berkeley	150.382	100	100	100
Right	150.994	48,19	48,39	48,29
Select Width	126.231	37,78	31,71	34,48
Select W/O (Best)	116.934	42,57	33,1	**37,24**
Select W/O (All)	121.200	41,68	**33,59**	37,2
Select (Terms)	54.603	**42,66**	15,49	22,72
Random N.C. (Best)	116.051	17,71	13,66	15,42
Random N.C. (Terms)	60.000	16,62	6,63	9,48

Tabelle A.24: Ergebnis des Vergleichs der in Abschnitt 5.4.2 beschriebenen *Select*-Verfahren mit den vom Berkeley Parser detektierten Strukturen im CHILDES-Korpus. Die Analyse wurde auf 15.000 Sätze beschränkt, die je **11 bis 14** Wörter enthalten.

	Hypothesen	Precision	Recall	F-Score
Berkeley	91.560	100	100	100
Right	91.978	40,23	40,41	40,32
Select Width	71.085	32,61	25,32	28,51
Select W/O (Best)	66.479	35,79	25,99	30,11
Select W/O (All)	68.134	35,65	**26,53**	**30,42**
Select (Terms)	33.339	**35,81**	13,04	19,12
Random N.C. (Best)	67.649	14,45	10,68	12,28
Random N.C. (Terms)	33.000	13,28	4,79	7,04

Tabelle A.25: Ergebnis des Vergleichs der in Abschnitt 5.4.2 beschriebenen *Select*-Verfahren mit den vom Berkeley Parser detektierten Strukturen im CHILDES-Korpus. Die Analyse wurde auf Sätze beschränkt, die je **15 bis 22** Wörter enthalten. Da das untersuchte Korpus lediglich ca. 6.100 Sätze dieser Länge enthält, weicht die Zahl untersuchter Sätze von den in Tabelle A.22 bis A.24 aufgeführten Analysen ab.

	Hypothesen	Precision	Recall	F-Score
Berkeley	51.470	100	100	100
Right	51.632	57,41	57,59	57,5
Select Width	43.492	48,77	41,21	44,67
Select W/O (Best)	39.946	**53,28**	41,35	46,57
Select W/O (All)	41.427	52,43	**42,2**	**46,76**
Select (Terms)	31.958	52,13	32,37	39,94
Random N.C. (Best)	40.385	29,21	22,92	25,69
Random N.C. (Terms)	33.000	31,16	19,98	24,34

Tabelle A.26: Ergebnis des Vergleichs der in Abschnitt 5.4.2 beschriebenen *Select*-Verfahren mit den vom Berkeley Parser detektierten Strukturen im CHILDES-Korpus. Die Analyse wurde auf die ersten 15.000 Sätze, die **3 bis 22** Wörter enthalten, beschränkt.

B Komponenten

Als Anhang werden hier die Komponenten und Reader, auf die in dieser Arbeit referenziert wurde, nach Anwendungsfeld gegliedert vorgestellt. Sofern nicht anders angegeben, wurden die Komponenten vom Autor dieser Arbeit entwickelt und werden mit der Standard-Installation von Tesla verbreitet. Weitere der mehr als 50 zum Zeitpunkt der Fertigstellung dieser Arbeit verfügbaren Komponenten und Reader werden in Hermes (2011) vorgestellt, zudem sei auf die online verfügbare Dokumentation sämtlicher Komponenten unter http://tesla.spinfo.uni-koeln.de/modules.html verwiesen.

B.1 Reader

Das Konzept der *Reader* wurde in Abschnitt 4.1.3.1 und 4.1.5.3 diskutiert. Neben generischen Text-, XML- und PDF-Readern auf Basis von *Apache Tika*[193] wurden Reader für annotierte Korpora entwickelt, die im Folgenden kurz vorgestellt werden.

B.1.1 BNC Reader

Dieser Reader macht die Annotationen des *British National Corpus* innerhalb von Tesla zugänglich, indem die zugrundeliegenden XML-Strukturen in Tesla-Rollen konvertiert werden.

BNC Reader	
Konsumiert	Signale (Text)
Produziert	Paragraph Detector, Sentence Detector, Tokenizer, Lemmatizer, POS-Tagger
Datenbank	TunguskaDB
Autor	Sebastian Rose
Lizenz	Eclipse Public Licence

[193] Siehe http://tika.apache.org/

B.1.2 TüBa Corpus Reader

Der *TüBa Corpus Reader* konvertiert in XML kodierte Tübinger Baumdatenbanken in Teslas Rollensystem (siehe auch Abschnitt 5.1).

TüBa Corpus Reader	
Konsumiert	Signale (Text)
Produziert	Sentence Detector, Tokenizer, POS-Tagger, Constituent Detector
Datenbank	TunguskaDB
Lizenz	Eclipse Public Licence

B.1.3 Childes Reader

Das in der CHILDES-Datenbank genutzte, proprietäre Format kann mit diesem Reader in Teslas Rollensystem konvertiert werden (siehe Abschnitt 5.1 und 5.4.3).

Childes Corpus Reader	
Konsumiert	Signale (Text)
Produziert	Sentence Detector, Tokenizer, POS-Tagger
Datenbank	TunguskaDB
Lizenz	Eclipse Public Licence

B.2 Präprozessierung

Die in diesem Abschnitt aufgeführten Komponenten führen eine grundlegende Präprozessierung von Texten durch, in der Satz- und Wortgrenzen detektiert werden.

B.2.1 Simple Tokenizer

Dieser Tokenizer nutzt das durch die `java.text.BreakIterator` bereitgestellten Möglichkeiten zur Segmentierung von Texten in (u.a.) Sätze und Wörter auf Basis vorgegebener Spracheinstellungen.

Simple Tokenizer		
Konsumiert	Signale (Text)	
Produziert	Sentence Detector, Tokenizer	
Datenbank	TunguskaDB	
Konfiguration	Locale	Zu verwendende Spracheinstellung
	Tag Whitespaces	Definiert, ob Leerzeichen annotiert werden sollen.
	Ignore case on type id	Definiert, ob die Type-Id der Annotationen Groß- und Kleinschreibung unterscheiden soll.
Lizenz	Eclipse Public Licence	

B.2.2 SPre

Einbindung des in Hermes & Benden (2005) vorgestellten SPre-Tokenizers in Tesla. Der Tokenizer kann sehr flexibel konfiguriert werden, um an verschiedene Texteigenschaften angepasst zu werden: So kann bspw. die Definition von Sätzen und Wörtern nahezu beliebig modifiziert werden.

SPre		
Konsumiert	Signale (Text)	
Produziert	Sentence Detector, Tokenizer	
Datenbank	TunguskaDB	
Autoren	Jürgen Hermes und Christoph Benden	
Konfiguration	Parser Configurations	Parser Konfigurationen u.a. für Buchstaben-, Wort- und Satzebene
	Abbreviation list	Zu verwendende Akürzungsliste
Lizenz	Eclipse Public Licence	

B.3 Filter

Die in diesem Abschnitt beschriebenen Komponenten dienen dazu, einen Filter für Datenstrukturen zu erzeugen, der anschließend von Komponenten, die Filterung unterstützen, verwendet werden kann. Der von der Rolle *Filter* geforderte AccessAdapter definiert u.a.

die Methode `boolean accepts(Annotation annotation)`, welche eine solche Verwendung möglich macht.

B.3.1 Subclass Filter

Anhand der von einer Datenstruktur extendierten Klassen entscheidet diese Komponente, ob die Datenstruktur gefiltert wird oder nicht. Dadurch ist es bspw. möglich, die von einem Tokenizer produzierten Datenstrukturen so zu filtern, dass lediglich Wörter, nicht aber Satzzeichen akzeptiert werden (siehe Abschnitt 5.1).

Interface Filter		
Konsumiert	Annotatator	
Produziert	Filter	
Datenbank	TunguskaDB	
Konfiguration	Sub Classes	Liste der akzeptierten Datentypen
Lizenz	Eclipse Public Licence	

B.3.2 Frequency Filter

Diese Komponente erzeugt einen Filter auf Basis der Häufigkeit oder des Rangs von Annotationen und kann verwendet werden, um die am häufigsten oder am seltensten vorkommenden Annotationstypen zu filtern. Grundlage der Berechnung der Häufigkeit ist dabei die Type-Id, die von einer Annotation bereitgestellt wird (vgl. Abschnitt 4.2.2).

Frequency Filter		
Konsumiert	Annotation Statistics	
Produziert	Type Filter	
Datenbank	TunguskaDB	
Konfiguration	Threshold	Gibt an, wie viele Annotationen gefiltert werden sollen (abhängig von den weiteren Optionen).
	Filter most frequent annotations	Definiert, ob die häufigsten oder seltensten Annotationen berücksichtigt werden sollen.
	Invert matching	Definiert, ob die Matching-Strategie invertiert werden soll, so dass bspw. nicht die 100 häufigsten Annotationen, sondern alle anderen vom Filter akzeptiert werden.
	Use Rank	Gibt an, ob statt der absoluten Häufigkeit der Rang der Annotationen verwendet werden soll.
Lizenz	Eclipse Public Licence	

B.3.3 Sequence Length Filter

Dieser Filter kann genutzt werden, um Sequenzen in Abhängigkeit der Anzahl von Elementen, die sie enthalten, zu filtern, und so bspw. die Anwendung von Alignment-Methoden auf Sätze zu beschränken, die eine minimale und/oder maximale Anzahl von Wörtern enthalten (vgl. Abschnitt 5.1).

Sequence Length Filter	
Konsumiert	Anchored Elements (Sequences), Anchored Elements (Items)
Produziert	Filter
Datenbank	TunguskaDB
Lizenz	Eclipse Public Licence

B.3.4 Range Filter

Um abhängig von der Verankerung einer Annotation am Signal filtern zu können, wurde diese Komponente entwickelt. So können bspw. triviale Strukturen aus einer Menge von Annotationen entfernt werden (vgl. Abschnitt 5.1).

Range Filter		
Konsumiert	Anchored Elements (Items)	
Produziert	Filter	
Datenbank	TunguskaDB	
Konfiguration	Invert Matching	Gibt an, ob der Filter Annotationen, deren Verankerung am Signal mit einer der konsumierten Annotationen übereinstimmt, akzeptiert (*false*) oder zurückweist (*true*).
Lizenz	Eclipse Public Licence	

B.3.5 Filter Rewriter

Die oben beschriebenen Filter erzeugen keine am Signal verankerten Annotationen, sondern bieten lediglich die Möglichkeit, für eine gegebene Annotation zu überprüfen, ob sie vom Filter akzeptiert oder zurückgewiesen wurde. Der *Filter Rewriter* wurde daher entwickelt, um diese Informationen in eine sequentielle Repräsentation zu konvertieren: Zu jeder akzeptierten Annotation wird eine Verankerung am Signal erzeugt, die auf die ursprüngliche Annotation verweist. Dies ermöglicht es bspw. in Kombination mit dem in Anhang B.3.1 beschriebenen *Subclass Filter*, Wörter und Punktuationszeichen in der Menge der von einem Tokenizer produzierten Annotationen zu trennen (wie in Kapitel 5 benötigt).

Filter Rewriter	
Konsumiert	Filter (1 - n), Anchored Elements (Items)
Produziert	Linked Annotations
Datenbank	TunguskaDB
Lizenz	Eclipse Public Licence

B.4 Satzstruktur

B.4.1 Berkeley Parser

Dieser heuristische Parser unterstützt u.a. Englisch, Deutsch, Französisch, Arabisch, Bulgarisch und Chinesisch (weitere Sprachen können durch Training des Parsers unterstützt werden) und zeichnet sich gegenüber dem im folgenden Abschnitt beschriebenen *Stanford Parser* durch deutlich besseres Laufzeit- und Speicherverhalten aus (siehe auch Abschnitt 5.1 und 5.2).

Berkeley Parser		
Konsumiert	Sentence Detector, Tokenizer	
Produziert	Parser, Constituent Tagger	
Datenbank	TunguskaDB	
Konfiguration	Grammar file	Das zu verwendende Sprachmodell. Standardmäßig ist in der Komponente lediglich die Unterstützung für das Deutsche und Englische integriert, weitere Sprachmodelle müssen manuell hinzugefügt werden.
Lizenz	GNU General Public Licence	

B.4.2 Stanford Parser

Der *Stanford Parser* basiert auf dem im vorherigen Abschnitt beschriebenen *Berkeley Parser* (und steht folglich ebenfalls unter der GPL), ist jedoch um zahlreiche Erweiterungen ergänzt und in vielen Punkten modifiziert worden. Unterstützte Sprachen sind Englisch, Deutsch, Französisch, Arabisch und Chinesisch, weitere Sprachmodelle lassen sich durch Training des Parsers erzeugen.

Stanford Parser		
Konsumiert	Sentence Detector, Tokenizer	
Produziert	Parser, Constituent Tagger	
Datenbank	TunguskaDB	
Konfiguration	Language Configuration	Das zu verwendende Sprachmodell. Standardmäßig ist in der Komponente lediglich die Unterstützung für das Deutsche und Englische integriert, weitere Sprachmodelle müssen manuell hinzugefügt werden.
	Max sentence length	Maximale Satzlänge
Lizenz	GNU General Public Licence	

B.5 Alignment

B.5.1 ABL Align

Diese Komponente setzt die in Abschnitt 2.2 beschriebene *Align*-Phase von ABL4J um: Anhand von identischen und abweichenden Subsequenzen in den analysierten Daten werden Hypothesen über syntaktische Strukturen generiert (siehe auch Abschnitt 5.4.1). Eine ausführliche Beschreibung der Konfigurationsoptionen findet sich in der Dokumentation von ABL unter `http://ilk.uvt.nl/menno/files/software/abl/abl-1.0.tar.gz`.

ABL Align		
Konsumiert	Anchored Elements (Sequences), Anchored Elements (Tokens)	
Produziert	Multi Value Constituents	
Datenbank	TunguskaDB	
Konfiguration	Exclude Empty	Definiert, ob auch leere Konstituenten erzeugt werden sollen.
	Don't Merge	Wenn aktiviert, wird bereits detektierten Konstituenten bei erneuter Detektion in einem anderen Kontext ein neues nonterminales Symbol zugewiesen. Andernfalls wird das bereits vorhandene Symbol erneut verwendet, was eine deutliche Reduktion des benötigten Speicherplatzes bewirkt, jedoch auch zu einer möglicherweise unerwünschten Vor-Kategorisierung von Konstituenten führt.
	Seed	Initialisierungswert des Zufallszahlen-Generators
	Alignment Method	Definiert die zu verwendende Alignment-Methode (siehe auch Abschnitt 2.2.1 und 5.4.1).
	Part Type	Definiert, ob identische oder nicht identische Strukturen als Konstituenten markiert werden sollen.
	Exhaustive	Gibt an, ob alle Sequenzpaare miteinander verglichen werden sollen (true), oder lediglich diejenigen, die gleiche Symbole enthalten (false).
Lizenz	GNU Lesser General Public Licence	

B.5.2 ABL Cluster

Von *ABL Align* erzeugte Hypothesen werden von *ABL Cluster* auf Basis ähnlicher Kontexte gruppiert.

ABL Cluster		
Konsumiert	Multi Value Constituents	
Produziert	Multi Value Constituens	
Datenbank	TunguskaDB	
Konfiguration	Cluster Method	Definiert die verwendete Clustering-Strategie.
	Merge Method	Definiert die verwendete Merge-Strategie.
Lizenz	GNU Lesser General Public Licence	

B.5.3 ABL Select

Diese Komponente setzt die in Abschnitt 2.2 beschriebene *Select*-Phase von ABL um.

ABL Select		
Konsumiert	Multi Value Constituents	
Produziert	Multi Value Constituents	
Datenbank	TunguskaDB	
Konfiguration	Select Method	Definiert die zu verwendende Select-Methode (siehe auch Abschnitt 5.4.2).
Lizenz	GNU Lesser General Public Licence	

B.5.4 Random Align

Diese Komponente dient dazu, eine untere Schranke für Precision, Recall und F-Score eines Verfahrens zur Strukturdetektion zu ermitteln, indem anhand der Annotationen einer Referenzkomponente Strukturen mit zufälligen Anfangs- und Endpositionen generiert werden (vgl. Abschnitt 5.2).

Random Align		
Konsumiert	Constituents	
Produziert	Constituents	
Datenbank	TunguskaDB	
Konfiguration	Random Seed	Initialisierung des Zufallszahlen-Generators
Lizenz	Eclipse Public Licence	

B.5.5 Boundaries Detector

Markiert mit Hilfe von N-Gramm-Bäumen (siehe Anhang B.9.2) die Positionen mehrfach in den untersuchten Korpora vorkommender N-Gramme (vgl. Abschnitt 5.3.1).

Boundaries Detector		
Konsumiert	N-Gram Tree (Suffix u. Präfix), Anchored Elements (Sequenzen u. Tokens)	
Produziert	Boundaries	
Datenbank	TunguskaDB	
Konfiguration	Minimum Context Width (Left/Right)	Definiert minimale Länge der zu berücksichtigen N-Gramme
	Sequence Start/End Weight	Bestimmt, wie stark Sequenzanfang bzw. -ende gewichtet werden, um ggfs. als Ausname zu obigem Konfigurationsparameter berücksichtigt zu werden.
Lizenz	Eclipse Public Licence	

B.5.6 Hypothesis Generator

Generiert Hypothesen über Strukturgrenzen, wobei die erzeugten Annotationen Informationen über Länge und Anzahl der Belege, die für die Bildung einer Hypothese verwendet wurden, enthalten (vgl. Abschnitt 5.3.1).

Hypothesis Generator		
Konsumiert	Boundaries, Anchored Elements (Sequenzen u. Tokens)	
Produziert	Context-aware Hypotheses	
Datenbank	TunguskaDB	
Konfiguration	Minimum Context Width (Left & Right)	Definiert minimale Länge der zu berücksichtigen N-Gramme
	Sequence Start & End Weight	Bestimmt, wie stark Sequenzanfang bzw. -ende gewichtet werden, um ggfs. als Ausname zu obigem Konfigurationsparameter berücksichtigt zu werden.
Lizenz	Eclipse Public Licence	

B.6 Entwicklung und Evaluation

Die in diesem Abschnitt beschriebenen Komponenten generieren Daten, die weniger zur weiteren computerlinguistischen Verarbeitung als vielmehr zur Evaluation anderer Komponenten und zur Fehlersuche geeignet sind.

B.6.1 Component Tests

Diese Komponente kann eingesetzt werden, um andere Komponenten (bzw. deren Access-Adapter) zu testen (vgl. Abschnitt 4.2.3 auf Seite 148). Die Testergebnisse werden als Annotationen interpretiert und entsprechend im Annotationsgraph abgelegt; zudem wird eine Zusammenfassung der Ergebnisse generiert und dem Experimentprotokoll (vgl. Abschnitt 4.1.6) hinzugefügt. Abgesehen davon, dass es somit auch möglich ist, diese Komponente für einen Selbsttest zu verwenden, scheint eine Weiterverarbeitung der generierten Daten in diesem Fall wenig sinnvoll.

Component Tests		
Konsumiert	Annotator	
Produziert	Test Result	
Datenbank	Hibernate	
Konfiguration	Timeout	Gibt an, ob und nach welcher Zeit die Ausführung einer einzelnen Test-Methode abgebrochen werden soll.
Lizenz	Eclipse Public Licence	

B.6.2 Annotation Comparator

Für eine – einfache – Evaluation tokenbasierter Annotationen lässt sich der *Annotation Comparator* verwenden: Die Komponente berechnet *Precision*, *Recall* und *F-Score* anhand einer als *Gold-Standard* definierten Rolle G und einer oder mehreren zu evaluierenden Komponenten E. Die ermittelten Werte beruhen darauf, wie viele der von G generierten Annotationen auch von E erzeugt wurden und umgekehrt, entsprechend der Definition von Precision und Recall in Kapitel 1 (siehe auch Kapitel 5).

Annotation Comparator	
Konsumiert	Anchored Elements (Gold), Anchored Elements (Evaluation, 1 oder mehr)
Produziert	Tabular Summary
Datenbank	Hibernate
Lizenz	Eclipse Public Licence

B.6.3 WordNet Similarity Evaluator

Diese Komponente kann für die Evaluation *semantisch ähnlicher* Annotationsgruppen verwendet werden. Aufgrund der Verwendung von *WordNet* ist der Einsatz allerdings auf die Analyse von Mengen englischsprachiger Einzelwörter beschränkt (vgl. Abschnitt 5.5.2).

WordNet Similarity Evaluator		
Konsumiert	Annotation Groups (1 oder mehr)	
Produziert	Tabular Summary	
Datenbank	TunguskaDB	
Konfiguration	Evaluation Method	Eine Evaluationsmethode, die die Distanz zwischen zwei Begriffen berechnet (bspw. ConSim oder SharedPath, vgl. Abschnitt 5.5).
Lizenz	Eclipse Public Licence	

B.6.4 Random Group Generator

Um eine untere Schranke für Bewertungen von Annotationsmengen zu berechnen, kann diese Komponente eingesetzt werden: Anhand der von einer Referenzkomponente erzeugten Mengen werden neue Mengen erzeugt, die hinsichtlich der Anzahl und Kardinalität der Mengen ebenso wie den zugewiesenen Elementen identisch sind, bei denen die Elemente jedoch zufällig auf die Mengen verteilt wurden (vgl. Abschnitt 5.5.1 und 5.5.2).

Random Group Generator		
Konsumiert	Annotation Groups	
Produziert	Annotation Groups	
Datenbank	TunguskaDB	
Konfiguration	Random Seed	Initialisierung des Zufallszahlen-Generators
Lizenz	Eclipse Public Licence	

B.6.5 Custom Choice Random Group Generator

Wie der im vorherigen Abschnitt beschriebene *Random Group Generator* dient auch diese Komponente dazu, Bewertungen von Annotationsmengen zu erleichtern. Der Unterschied beider Komponenten liegt darin, dass der *Custom Choice Random Group Generator* nicht auf die Elemente der Referenzkomponente zurückgreift, sondern zufallsbasiert eine eigenständige Auswahl trifft (vgl. Abschnitt 5.5.1 und 5.5.2).

Random Group Generator		
Konsumiert	Annotation Groups, Anchored Elements	
Produziert	Annotation Groups	
Datenbank	TunguskaDB	
Konfiguration	Random Seed	Initialisierung des Zufallszahlen-Generators
Lizenz	Eclipse Public Licence	

B.6.6 Group Similarity Analyzer

Diese Komponente ermittelt mit Hilfe einer Referenz die Qualität von Kategorisierungs- und Clusteringverfahren (vgl. Abschnitt 5.5.1).

Group Similarity Analyzer	
Konsumiert	Annotation Groups, Annotations
Produziert	Tabular Summary
Datenbank	TunguskaDB
Autorin	Alena Geduldig
Lizenz	Eclipse Public Licence

B.6.7 Sequence Metrics

Zur Auswertung sequenzbasierter Verteilungen, wie etwa in Abbildung 5.3 dargestellt, kann diese Komponente genutzt werden.

Sequence Metrics	
Konsumiert	Annotations (Sequences), Annotations (Items), Annotations (Analyzed)
Produziert	Tabular Summary
Datenbank	TunguskaDB
Lizenz	Eclipse Public Licence

B.7 Tokenbasierte Auszeichnung

B.7.1 Gazetteer

Die Gazetteer-Komponente verwendet ein einfaches Vollform-Lexikon (in Form einer durch Zeilenumbrüche separierten Liste von Einträgen), um Vorkommen der dort definierten Wortsequenzen zu annotieren. Die Analyse des Korpus wird satzweise durchgeführt, um satzübergreifende Sequenzen zu verhindern. Die generierten Kategorie-Annotationen enthalten ein *Label*, welches den Typ der Kategorie (bspw. Vorname, Stadt, Wochentag) repräsentiert und beliebig modifiziert werden kann – dies bedeutet allerdings, dass es (im Gegensatz zu einer Interface-basierten Kategorisierung, wie sie bspw. für POS-Tags implementiert wurde, vgl. auch Anhang B.7.3) nur eingeschränkt möglich ist, die Bedeutung der Label algorithmisch zu interpretieren. Wird hingegen lediglich die Type-Id der generierten Annotationen benötigt (vgl. Abschnitt 4.2.2), wie in der in Abschnitt 5.4.4 beschriebenen Untersuchung, entfällt diese Einschränkung.

Gazetteer		
Konsumiert	Anchored Elements (Sentences), Anchored Elements (Tokens)	
Produziert	Multi Value Categorizer	
Datenbank	TunguskaDB	
Autor	Fabian Steeg	
Konfiguration	List (1 – 64)	Datenbasis der Komponente: Jeder Datensatz besteht aus einer Kategorie und einer Liste zulässiger Wort-Sequenzen.
	Ignore Case	Bei Aktivierung wird Groß-/Kleinschreibung beim Vergleich der Tokens mit der Datenbasis ignoriert.
Lizenz	Eclipse Public Licence	

B.7.2 Stanford Named Entity Extractor

Diese Komponente basiert auf einem an der Stanford University entwickelten Algorithmus zur Detektion von *Named Entities*[194]. Der Algorithmus arbeitet probabilistisch, unter Verwendung sprachabhängiger Trainingsdaten – standardmäßig bietet die Komponente

[194] Siehe http://nlp.stanford.edu/software/CRF-NER.shtml für eine ausführliche Beschreibung des Algorithmus

Modelle für die Erkennung von Personen-, Orts- und Organisationsnamen für englische und deutsche[195] Texte an, sie kann jedoch auch für die Verwendung anderer Modelle konfiguriert werden (vgl. auch Abschnitt 5.4.4).

Stanford Named Entity Extractor		
Konsumiert	Sentence Detector, Tokenizer	
Produziert	Named Entities	
Datenbank	Hibernate	
Konfiguration	Classifier	Sprachspezifisches Modell
	Tag Persons	Gibt an, ob Personennamen ausgezeichnet werden sollen.
	Tag Organisations	Gibt an, ob Organisationsnamen ausgezeichnet werden sollen.
	Tag Locations	Gibt an, ob Ortsnamen ausgezeichnet werden sollen.
	Tag Misc	Gibt an, ob Named Entities ausgezeichnet werden sollen, die nicht unter eine der obigen Kategorien fallen.
Lizenz	GNU General Public Licence	

B.7.3 Tree Tagger Wrapper

Der *Tree Tagger Wrapper* integriert den von Schmid entwickelten *Tree Tagger*[196] in Tesla. Der probabilistische Tagger arbeitet auf Basis von Entscheidungsbäumen, die anhand von Trainingsdaten erlernt wurden (vgl. Schmid 1994 für eine ausführliche Beschreibung der Funktionsweise), und ist für verschiedene Sprachen einsetzbar auch: Auf der Homepage des Projekts sind neben englischen und deutschen Trainingsdaten u.a. auch französische und bulgarische Daten zugänglich. Der Tree Tagger Wrapper konvertiert die generierten POS-Tags in die in Abschnitt 4.1.4.1 beschriebene Interfacehierarchie – zum Zeitpunkt der Fertigstellung dieser Arbeit werden jedoch nur das deutsche und das englische Tag-Set unterstützt. Die Lizenz des Taggers untersagt grundsätzlich eine Weiterverbreitung der

[195] Hier wurden die in Faruqui & Padó (2010) beschriebenen Daten verwendet, die unter http://nlpado.de/~sebastian/software/ner_german.shtml zum Download angeboten werden.

[196] Siehe http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/DecisionTreeTagger.html.

Anwendung, ihr Autor hat jedoch der Integration in Tesla unter der Bedingung, dass die Komponente ausschließlich für nicht-kommerzielle Zwecke eingesetzt wird, zugestimmt.

Tree Tagger Wrapper		
Konsumiert	Sentence Detector, Tokenizer	
Produziert	Tesla POS Tagger	
Datenbank	TunguskaDB	
Konfiguration	Tree Tagger binary directory	Programmverzeichnis
	Tree Tagger model file	Das zu verwendende Sprachmodell
Lizenz	Frei für nicht-kommerzielle Anwendung	

B.7.4 Order-based Sequencer

Diese Komponente kann dazu genutzt werden, die von verschiedenen Komponenten generierten Annotationen zu aggregieren und als eigene Annotationssequenz für die weitere Prozessierung zur Verfügung zu stellen. Damit ist es bspw. im Rahmen der Anwendung eines Alignment-Verfahrens möglich, Eigennamen durch vom Gazetteer erkannte Kategorien auszutauschen, und so die Ergebnisse des Verfahrens zu modifizieren (vgl. Abschnitt 5.4.4).

Order-based Sequencer		
Konsumiert	Anchored Elements (Base), Anchored Elements (Replacements, 1 - n)	
Produziert	Replacements	
Datenbank	TunguskaDB	
Konfiguration	Order	Reihenfolge der zu untersuchenden konsumierten Adapter
Lizenz	Eclipse Public Licence	

B.7.5 Geo Location Extender

Diese Komponente erweitert als Ortsbezeichnungen interpretierte *Named Entities* um zusätzliche Angaben wie Längen- und Breitengrad. Die benötigten Wörterbücher stammen

vom *GeoNames*-Projekt[197], werden täglich aktualisiert und können über die Konfigurationsschnittstelle der Komponente heruntergeladen werden[198]. Für einen möglichen Anwendungsfall, der über eine musterbasierte Informationsanreicherung hinaus geht, sei auf Abschnitt 4.1.4.2 dieser Arbeit verwiesen.

Geo Location Extender		
Konsumiert	Tokenizer, Named Entities, Filter (0 – n)	
Produziert	Locations	
Datenbank	Hibernate	
Konfiguration	Lexikon	Lexika, die für die Detektion und Erweiterung von Ortsangaben verwendet werden sollen
	Filters to match	Anzahl der Filter, die eine Ortsangabe akzeptieren müssen
Lizenz	Eclipse Public Licence	

B.8 Clustering

Die in diesem Abschnitt vorgestellten Komponenten stellen unterschiedliche Clusteringverfahren zur Verfügung, durch die Vektoren gruppiert werden können. Da sich die Komponenten sowohl in ihrer Konfiguration als auch in der Art der Zuordnung teilweise stark unterscheiden, wurden sie als individuelle Komponenten implementiert.

B.8.1 K-Means++ Clusterer

Diese Komponente verwendet ein modifiziertes K-Means-Clusteringverfahren zur Zuordnung von Vektoren: Im Gegensatz zum herkömmlichen K-Means-Algorithmus werden die initialen Clusterzentren nicht zufällig gewählt, sondern auf Basis der zu clusternden Vektoren berechnet, wodurch sich sowohl die Qualität der Ergebnisse als auch die benötigte Laufzeit stark verbessern kann.

[197] Siehe http://www.geonames.org/

[198] Wird kein lokaler Tesla-Server verwendet, müssen die Wörterbücher allerdings manuell in das entsprechende Verzeichnis auf dem Server kopiert werden.

K-Means++ Clusterer		
Konsumiert	Vectors	
Produziert	Clusters	
Datenbank	TunguskaDB	
Konfiguration	Random Seed	Initialisierung des Zufallszahlengenerators
	Number of Clusters	Anzahl der zu erzeugenden Cluster
	Maximum number of iterations	Anzahl der Iterationen, in denen Cluster-Zentren neu berechnet werden
Lizenz	Apache Licence	

B.8.2 Weka EM Clusterer

Das in Java entwickelte Data Mining Framework *Weka*[199] stellt unter anderem verschiedene Clustering-Verfahren zur Verfügung, wie den hier verwendeten *Expectation-Maximization*-Algorithmus. Im Unterschied zum K-Means-Clustering werden Vektoren hier nicht genau einem Cluster zugeordnet, sondern stattdessen (mit unterschiedlichen Zugehörigkeitsgraden) jedem Cluster.

Weka EM Clusterer		
Konsumiert	Vectors	
Produziert	Clusters	
Datenbank	db4o	
Konfiguration	Max Iterations	Maximale Anzahl der Wiederholungen
	Seed	Initiale Zufallszahl
	Number of Clusters	Anzahl der Cluster
Lizenz	GNU General Public Licence	

B.8.3 Word Vector Generator

Der *Word Vector Generator* erzeugt vektorielle Repräsentationen beliebiger Annotationen auf Basis ihres Kontextes (vgl. bspw. Manning & Schütze 1999, S. 296ff). Die Breite des

[199] Siehe http://www.cs.waikato.ac.nz/~ml/weka/index.html

zu analysierenden Kontextes ist einstellbar, ebenso können Filter verwendet werden, um die Annotationen, zu denen ein Vektor generiert werden soll, und die Annotationen, die in einem Vektor berücksichtigt werden sollen, einzuschränken.

Word Vector Generator		
Konsumiert	Sequence Annotator, Filter (0 - n)	
Produziert	Labeled Vectors	
Datenbank	db4o	
Konfiguration	Window Size	Größe des 'Fensters', das über das zugrundeliegende Signal verschoben wird
	Vector Implementation	Die zu verwendende Implementation des Interfaces `IIntegerVector`.
	Max Iterations	Maximale Anzahl der Neuberechnung und -Zuweisung von Clusterzentren und Vektoren
	Filters to match for vector entry	Die Anzahl der Filter, die eine Annotation akzeptieren müssen, damit sie im Kontext einer anderen Annotation berücksichtigt wird.
	Filters to match for vector generation	Die Anzahl der Filter, die eine Annotation akzeptieren müssen, damit ein Vektor für sie erzeugt wird.
Autoren	Sonja Subicin und Stephan Schwiebert	
Lizenz	Eclipse Public Licence	

B.9 Sonstige

B.9.1 TF/IDF

Diese Komponente berechnet *Term Frequency* (TF) und *Inverse Document Frequency* (IDF) beliebiger Annotationen anhand ihrer Type-Id und stellt so eine Möglichkeit bereit, die Relevanz annotierter Ausdrücke innerhalb eines Dokumentes zu berechnen. Die für die Berechnung von TF und IDF notwendigen Häufigkeiten der Vorkommen der Annotationen können zudem von anderen Komponenten, wie dem in Anhang B.3.2 beschriebenen *Frequency Filter*, genutzt werden.

TF/IDF	
Konsumiert	Annotator
Produziert	TF/IDF
Datenbank	db4o
Lizenz	Eclipse Public Licence

B.9.2 N-Gram Tree Generator

Der *N-Gram Tree Generator* erzeugt die in Abschnitt 5.3.1 beschriebenen N-Gramm-Bäume. Die maximale Länge eines N-Gramms ist ebenso einstellbar wie die *Richtung* eines N-Gramms (Suffix- oder Präfix-Baum).

NGram Tree Generator		
Konsumiert	Anchored Elements (Symbols), Anchored Elements (Sequences)	
Produziert	NGram Trees	
Datenbank	TunguskaDB	
Konfiguration	Reverse	Definiert, ob ein Suffix- oder Präfixbaum erzeugt werden soll.
	Max Length	Maximale Tiefe des Baums
Lizenz	Eclipse Public Licence	

B.9.3 Substitution Rule Generator

Diese Komponente generiert kontextbezogene Substitutionsregeln, d.h. Regeln der Form $x\alpha y \rightarrow xBy$, wobei x und y einen Kontext definieren, und B die Menge der Annotationen (oder Annotationssequenzen) umfasst, durch die α ersetzt werden kann (vgl. Abschnitt 5.5.1 und 5.5.2).

Substitution Rule Generator	
Konsumiert	Context-aware Hypotheses
Produziert	Substitution Rules
Datenbank	TunguskaDB
Lizenz	Eclipse Public Licence

B.9.4 Generalized Substitution Rule Generator

Diese Komponente erzeugt generalisierte Substitutionsregeln, indem für jeden Type alle konsumierten kontextbezogenen Regeln zusammengefasst werden (vgl. Abschnitt 5.5.1).

Substitution Rule Generator	
Konsumiert	Substitution Rules
Produziert	Substitution Rules
Datenbank	TunguskaDB
Lizenz	Eclipse Public Licence

C Experimente

Die in Kapitel 5 beschriebenen und ausgewerteten Experimente stehen unter http://www.spinfo.phil-fak.uni-koeln.de/sschwieb_thesis.html zum Download zur Verfügung. Im Folgenden wird daher nur knapp auf die jeweiligen Experimente eingegangen, um deutlich zu machen, ob bzw. welche Parameter modifiziert wurden, und welche Daten ausgewertet wurden.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:corpus_configuration id="BNC" xmlns:ns2="http://spinfo.uni-koeln.de/tesla">
    <providerClass>de.uni_koeln.spinfo.tesla.datasource.zip.ZipDocumentProvider</providerClass>
    <displayName>British National Corpus</displayName>
    <description>The British National Corpus in XML</description>
    <readerClass>de.uni_koeln.spinfo.tesla.component.reader.BNCReader</readerClass>
    <encoding>UTF-8</encoding>
    <indexed>true</indexed>
    <configurations>
        <entry>
            <key>path</key>
            <value>bnc.zip</value>
        </entry>
        <entry>
            <key>suffix</key>
            <value>.xml</value>
        </entry>
    </configurations>
    <metaData>
        <format>XML (TEI, Custom, UTF-8)</format>
        <source>British National Corpus, XML Edition</source>
        <language>English</language>
        <rights>BNC User Licence http://www.natcorp.ox.ac.uk/docs/licence.html</rights>
        <publisher>University of Oxford</publisher>
    </metaData>
</ns2:corpus_configuration>
```

Listing C.1: Definition einer Datenquelle in Tesla am Beispiel des BNC. Der mit dem Schlüssel `path` verknüpfte Eintrag verweist auf das verwendete Zip-Archiv, zulässig sind sowohl relative als auch absolute Pfadangaben. Relative Pfade werden dabei in Bezug auf das Unterverzeichnis *datasources* des Servers aufgelöst.

Da die verwendeten Korpora nicht zum Download angeboten werden dürfen, müssen diese separat bezogen und anschließend in Tesla integriert werden. Die verlinkten Tesla-Installationen sind hierfür bereits vorbereitet und enthalten die benötigten Datenquellen-Definitionen (vgl. Abschnitt 4.1.3), so dass die Korpora lediglich als .zip-Dateien kompri-

miert und im Unterverzeichnis *datasources* des Servers[200] abgelegt werden müssen. Die interne Struktur der Archive ist dabei irrelevant, da die enthaltenen Dokumente bei anschließendem Neustart des Servers indexiert und über Prüfsummen referenziert werden, wie in Abschnitt 4.1.3 beschrieben.

Die Definitionen der Datenquellen sind im Unterverzeichnis *configuration/datasources* des Servers zu finden; Listing C.1 zeigt exemplarisch den Aufbau einer solchen Definition anhand des BNC.

C.1 Statistische Auswertung der Korpora (Abschnitt 5.1)

Die in Abschnitt 5.1 beschriebenen Merkmale der untersuchten Korpora werden mit Hilfe von fünf Experimenten (zu finden im Unterverzeichnis *5.1*) extrahiert, die sich jedoch lediglich in den untersuchten Korpora unterscheiden. Tabelle 5.1 fasst Ergebnisse verschiedener Komponenten zusammen: Spalte eins bis drei (Anzahl der Sätze, Wörter und Types) werden von der Komponente *Gold Metrics* ermittelt und in der Zusammenfassung eines ausgeführten Experiments (vgl. Abschnitt 4.1.6) aufgeführt – Punktuationszeichen werden hier nicht berücksichtigt. Die Anzahl der Strukturen entspricht der Anzahl von Annotationen, die der TüBa-Reader für die Rolle *Constituent Tagger* produziert; gleiches gilt für die Anzahl gefilterter Strukturen, die der Anzahl von Annotationen entspricht, die von der hier als *All Gold Structures* bezeichneten Komponente generiert wird.

Grundlage der in Abschnitt 5.1 dargestellten Diagramme sind von *Gold Metrics* erzeugte CSV-Dateien, allerdings muss zur Reproduktion der Ergebnisse eine Konfigurationsänderung durchgeführt werden: Die als *Sentence Length Filter* bezeichnete Komponente darf ausschließlich Sätze der Länge 3 bis 30 akzeptieren, so dass diese Werte für Minimum und Maximum eingestellt werden müssen.

C.2 Ermittlung von Erwartungswerten (Abschnitt 5.2)

Das hier vorgestellte Experiment (zu finden im Unterverzeichnis *5.2*) ermittelt Erwartungswerte zur Bewertung von Verfahren zur Strukturaufdeckung und wurde verwendet, um die in Abschnitt 5.2 abgebildeten Tabellen zu erzeugen. Je fünf Instanzen des *Random Align*-Verfahrens erzeugen zufällige Strukturen anhand der Annotationen des Gold-

[200] Der in den Tesla-Client integrierte Server ist wiederum im Unterverzeichnis *plugins/de.uni_koeln.spinfo.tesla.server_<BUILD-ID>* zu finden. Der mit *<BUILD-ID>* bezeichnete Teil des Verzeichnisnamens repräsentiert das Datum, an dem der Server kompiliert wurde, und kann daher variieren.

Standards und des *Berkeley Parsers*; in den Tabellen sind die Mittelwerte abgebildet. Auf die tabellarische Zusammenfassung kann mit Hilfe des in 4.1.6 beschriebenen Export-Mechanismus zugegriffen werden.

C.3 Evaluation von Alignment-Verfahren (Abschnitt 5.4.1)

Vier der in Abschnitt 5.4.1 zur Evaluation von Alignment-Verfahren verwendeten Experimente unterscheiden sich lediglich bezüglich der analysierten Korpora. Die in den Experimenten genutzten Instanzen der *Range Comparator*-Komponente (vgl. Anhang B.6.2) erzeugen die in Abschnitt 5.4 und Anhang A.1 aufgeführten Tabellen. Nach Ausführung eines Experiments können diese mit Hilfe des in Abschnitt 4.1.6 beschriebenen Export-Mechanismus als CSV- und LaTeX-Dokumente exportiert werden.

Um die in den entsprechenden Tabellen aufgeführten Ergebnisse der *Random*-Verfahren zu reproduzieren, müssen die Experimente mehrfach ausgeführt und hinsichtlich der Konfiguration des *Random Seed*-Parameters der Zufallskomponenten variiert werden: Hier wurde stets der Mittelwert aus fünf Durchläufen verwendet, wobei zur Initialisierung des Zufallszahlen-Generators die Werte 0 bis 4 genutzt wurden. Das Experiment *Cross-Align TüBa-DS* verwendet fünf der *Range Comparator*-Komponenten, um die in Tabelle 5.7 dargestellte Gegenüberstellung der Verfahren zu berechnen. Alle Experimente sind im Unterverzeichnis *5.4.1* enthalten.

C.4 Evaluation von Select-Verfahren (Abschnitte 5.4.2 und 5.4.3)

Die Unterverzeichnisse *5.4.2* und *5.4.3* enthalten die Experimente zur Evaluation verschiedener Select-Verfahren – wie in Abschnitt 5.4.2 beschrieben wurde bei der Analyse von TüBa-D/Z und BNC-G auf die Anwendung der Alignment-Verfahren *WF-B* und *All* verzichtet. Analog zu den in Anhang C.3 beschriebenen Experimenten muss auch hier die Konfiguration der Random-Komponenten modifiziert werden, um die Ergebnisse exakt zu reproduzieren. Die in Tabelle 5.12 zusammengefasste Analyse von Select-Verfahren bei Beschränkung der minimalen und maximalen Satzlänge erfordert ebenfalls die mehrfache Ausführung eines Experimentes mit unterschiedlicher Konfiguration: Hier müssen die Parameter *Minimum* und *Maximum* der Komponente *Sentence Length Filter* im *CHILDES*-Experiment entsprechend angepasst werden.

Das Experiment zur Evaluation von *Suffix Select* am TüBa-D/S enthält zusätzlich noch eine weitere, dort als *Context Frequency Analyzer* bezeichnete Komponente – diese ermit-

telt die Häufigkeit der Wortpaare, die den unmittelbaren Kontext einer Hypothese definieren. Wird das Experiment ausgeführt, nachdem die minimale Kontextlänge der vom *Boundaries Detector* zu berücksichtigenden Positionen auf ein Wort festgelegt wurde, lassen sich die in einer Fußnote auf Seite 192 dargestellten Überlegungen zum Einfluss hochfrequenter Wörter auf die Hypothesenbildung empirisch untersuchen.

C.5 Evaluation variierender Präprozessierung (Abschnitt 5.4.4)

Das in Abschnitt 5.4.4 skizzierte Experiment ist in dem Unterverzeichnis *5.4.4* enthalten. Die voreingestellte Konfiguration entspricht Abbildung 5.12, kann jedoch in vielen Punkten variiert werden: Neben einer Modifikation der *Gazetteer*-Konfiguration ist es insbesondere durch Modifikation der vom *Order Based Sequencer* (vgl. Anhang B.7.4) konsumierten Rollen (etwa durch Entfernen des *Stanford Named Entities Detector* oder Änderung der Reihenfolge der konsumierten Rollen in der Konfiguration der Komponente) möglich, die Typisierung der Tokens zu beeinflussen.

C.6 Kategorisierung von Wörtern (Abschnitt 5.5)

Die in den Unterverzeichnissen *5.5.1* und *5.5.2* enthaltenen Experimente können genutzt werden, um die in Abschnitt 5.5.2 beschriebenen Ergebnisse zu reproduzieren. Das in Abschnitt 5.5.3 skizzierte Experiment zur Analyse semantischer Assoziationen mit *HAL* ist in Unterverzeichnis *5.5.3* zu finden.

Tabellenverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Literaturverzeichnis

Of all things I liked books best.

(Nikola Tesla)

Benden, C.: 2004, 'Automated Detection of Morphemes Using Distributional Measurements', in C. Weihs & W. Gaul (eds.), *Classification - the Ubiquitous Challenge. Proceedings of the 28th Annual Conference of the Gesellschaft für Klassifikation e.V.*, Springer, Berlin, Heidelberg, S. 490–497.

Bird, S., D. Day, J. Garofolo, J. Henderson, C. Laprun & M. Liberman: 2000, 'ATLAS: A Flexible and Extensible Architecture for Linguistic Annotation', in *Proceedings of the Second International Conference on Language Resources and Evaluation*, European Language Resources Association, Paris.

Bird, S. & M. Liberman: 2001, 'A Formal Framework for Linguistic Annotation', *Speech Communication* 33, S. 23–60.

Bloch, J.: 2008, *Effective Java*^{TM}, zweite Ausgabe, Addison-Wesley, Upper Saddle River, NJ, USA.

Bod, R.: 2006, 'An all-subtrees approach to unsupervised parsing', in *ACL 2006, 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, Band 44, Association for Computational Linguistics, Sydney, Australia, S. 865–872.

Brodsky, P., H. Waterfall & S. Edelman: 2007, 'Characterizing motherese: On the computational structure of child-directed language', in *Proceedings of the 29th Cognitive Science Society Conference*, Cognitive Science Society, Austin, Texas, USA, S. 833–38.

Burnard, L. & S. Bauman: 2008, *P5 Guidelines for Electronic Text Encoding and Interchange*, Text Encoding Initiative Consortium.

Buyko, E. & U. Hahn: 2008, 'Fully embedded type systems for the semantic annotation layer', in *ICGL 2008 - Proceedings of First International Conference on Global Interoperability for Language Resources*, Hong Kong, SAR, S. 26–33.

Chomsky, N.: 1957, *Syntactic Structures*, Mouton & Co., The Hague.

Clark, A. & R. Eyraud: 2007, 'Polynomial Identification in the Limit of Substitutable Context-free Languages', *Journal of Machine Learning Research* 8, S. 1725–1745.

Clark, A. & S. Lappin: 2010, 'Unsupervised learning and grammar induction', in A. Clark, C. Fox & S. Lappin (eds.), *The Handbook of Computational Linguistics and Natural Language Processing*, Wiley-Blackwell, Malden MA and Oxford, S. 197–220.

Clayberg, E. & D. Rubel: 2006, *Eclipse: Building Commercial-Quality Plug-ins*, zweite Ausgabe, Addison-Wesley, Boston.

Codd, E. F.: 1970, 'A relational model of data for large shared data banks', *Communications of the ACM* 13(6), S. 377–387.

Cramer, B.: 2007, 'Limitations of Current Grammar Induction Algorithms', in *Proceedings of the 45th Annual Meeting of the ACL: Student Research Workshop*, The Association for Computational Linguistics, Stroudsburg, PA, USA, S. 43–48.

Cunningham, H.: 2000, *Software architecture for language engineering*, Ph.D. thesis, University of Sheffield.

Cunningham, H. & K. Bontcheva: 2006, 'Computational Language Systems, Architectures', in K. Brown, A. H. Anderson, L. Bauer, M. Berns, G. Hirst & J. Miller (eds.), *The Encyclopedia of Language and Linguistics*, zweite Ausgabe, Elsevier, München.

Cunningham, H., D. Maynard, K. Bontcheva, V. Tablan, C. Ursu, M. Dimitrov, M. Dowman & N. Aswani: 2005, *Developing language processing components with GATE (a user guide)*.

Cunningham, H., W. Peters, C. McCauley, K. Bontcheva & Y. Wilks: 2000, 'Uniform Language Resource Access and Distribution', *Proceedings of the Workshop on Ontologies and Language Resources (OntoLex'2000), Sozopol* .

Dean, J. & S. Ghemawat: 2004, 'MapReduce: Simplified data processing on large clusters', in *OSDI'04: Proceedings on the 6th Symposium on Operating System Design and Implementation*, USENIX Association, Berkeley, CA, USA, S. 137 – 150.

Deerwester, S., S. T. Dumais, G. W. Furnas, T. K. Landauer & R. Harshman: 1990, 'Indexing by Latent Semantic Analysis', *Journal of the American Society for Information Science* 41, S. 391 – 407.

Duffner, R. & A. Näf: 2006, 'Digitale Textdatenbanken im Vergleich', *Linguistik online* 28, S. 7–22.

Edelman, S., Z. Solan, D. Horn & E. Ruppin: 2004, 'Bridging Computational, Formal and Psycholinguistic Approaches to Language', in K. Forbus, D. Gentner & T. Regier (eds.), *Proceedings of the 26th Conference of the Cognitive Science Society*, Chicago, Illionis, USA, S. 345–350.

Edlich, S., J. Paterson, H. Hörning & R. Hörning: 2006, *The definitive guide to db4o*, Apress, Berkely, CA, USA.

Edmonds, P. & A. Kilgarriff: 2002, 'Introduction to the special issue on evaluating word sense disambiguation systems', *Natural Language Engineering* 8, S. 279–291.

Egner, M. T., M. Lorch & E. Biddle: 2007, 'UIMA GRID: Distributed Large-scale Text Analysis', in *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, IEEE, S. 317–326.

Farach, M.: 1997, 'Optimal suffix tree construction with large alphabets', in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, S. 137–143.

Faruqui, M. & S. Padó: 2010, 'Training and Evaluating a German Named Entity Recognizer with Semantic Generalization', in M. Pinkal, I. Rehbein, S. S. im Walde & A. Storrer (eds.), *Semantic Approaches in Natural Language Processing*, Universitätsverlag des Saarlandes, [Saarbrücken, Germany, S. 129–134.

Ferrucci, D. & A. Lally: 2003, 'Accelerating Corporate Research in the Development, Application and Deployment of Human Language Technologies', in *Proceedings of the HLT-NAACL 2003 Workshop on Software Engineering and Architectures for Language Technology Systems*, Association for Computational Linguistics, Stroudsburg, PA, USA, S. 67–74.

Ferrucci, D. & A. Lally: 2004, 'UIMA: an architectural approach to unstructured information processing in the corporate research environment', *Natural Language Engineering* 10(3-4), S. 327–348.

Fodor, P., A. Lally & D. A. Ferrucci: 2008, 'The Prolog Interface to the Unstructured Information Management Architecture', *CoRR* abs/0809.0680.

Gamma, E., R. Helm, R. Johnson & J. Vlissides: 1995, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Geertzen, J. & M. V. Zaanen: 2004, 'Grammatical Inference Using Suffix Trees', in G. Paliouras & Y. Sakakibara (eds.), *ICGI '04: Proceedings of the 7th international colloquium on Grammatical Inference*, Springer, Berlin, Heidelberg, S. 163–174.

Ghemawat, S., H. Gobioff & S.-T. Leung: 2003, 'The Google file system', *ACM SIGOPS Operating Systems Review* 37, S. 29–43.

Gietz, P., A. Aschenbrenner, S. Büdenbender, F. Jannidis, M. Küster, C. Ludwig, W. Pempe, T. Vitt, W. Wegstein & A. Zielinski: 2006, 'TextGrid and eHumanities', in *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, IEEE Computer Society, Washington, DC, USA, S. 133–141.

Gillick, D.: 2009, 'Sentence Boundary Detection and the Problem with the U.S.', in *Proceedings of the North American Chapter of the Association for Computational Linguistics - Human Language Technologies*, S. 241–244.

Gold, E. M.: 1967, 'Language Identification in the Limit', *Information and Control* 10(5), S. 447–474.

Götz, T. & O. Suhre: 2004, 'Design and implementation of the UIMA Common Analysis System', *IBM Systems Journal* 43, S. 476–489.

Gusfield, D.: 1997, *Algorithms on strings, trees, and sequences*, Cambridge University Press, New York, NY, USA.

Habert, B. & P. Zweigenbaum: 2002, 'Contextual acquisition of information categories - What has been done and what can be done automatically?', in B. E. Nevin & S. M. Johnson (eds.), *The legacy of Zellig Harris: language and information into the 21st century*, Band 2, John Benjamins, S. 203–231.

Harris, Z. S.: 1957, 'Co-occurrence and transformation in linguistic structure', *Language* 33(3), S. 283–340.

Harris, Z. S.: 1959, 'The transformational model of language structure', *Anthropological Linguistics* 1(1), S. 27–29.

Harris, Z. S.: 1965, 'Transformational theory', *Language* 41(3), S. 363–401.

Harris, Z. S.: 1968, *Mathematical structures of language*, Wiley (Interscience), New York, NY, USA.

Harris, Z. S.: 1976, 'On a theory of language', *The Journal of Philosophy* 73, S. 253–276.

Harris, Z. S.: 1988, *Language and information*, Band 28, Columbia University Press, New York.

Harris, Z. S., M. Gottfried, T. Ryckman, A. Daladier & P. Mattick: 1989, *The Form of Information in Science: Analysis of an Immunology Sublanguage*, Band 104, Kluwer Academic Publishers, Dordrecht; Boston.

Hemphill, C. T., J. J. Godfrey & G. R. Doddington: 1990, 'The ATIS Spoken Language Systems Pilot Corpus', in *Proceedings of the DARPA Speech and Natural Language Workshop*, Association for Computational Linguistics, Stroudsburg, PA, USA, S. 96–101.

Hermes, J.: 2011, 'Textprozessierung - Design und Applikation', Voraussichtliche Veröffentlichung 2012.

Hermes, J. & C. Benden: 2005, 'Fusion von Annotation und Präprozessierung als Vorschlag zur Behandlung des Rohtextproblems', in B. Fisseni, H.-C. Schmitz, B. Schröder & P. Wagner (eds.), *Sprachtechnologie, mobile Kommunikation und linguistische Ressourcen. Beiträge zur GLDV-Tagung 2005 in Bonn*, Band 8, Lang, Frankfurt am Main, S. 78–90.

Hermes, J. & S. Schwiebert: 2010, 'Classification of Text Processing Components: The Tesla Role System', in A. Fink, B. Lausen, W. Seidel & A. Ultsch (eds.), *Advances in Data Analysis, Data Handling and Business Intelligence*, 4, Springer, Berlin, Heidelberg, S. 285–294.

Horn, D., Z. Solan, E. Ruppin & S. Edelman: 2004, 'Unsupervised language acquisition: syntax from plain corpus', Presented at Newcastle Workshop on Human Language.

Johnson, R.: 2003, *Expert one-on-one J2EE design and development*, Wrox Press Ltd., Birmingham, UK, UK.

Jungen, O. & H. Lohnstein: 2006, *Einführung in die Grammatiktheorie*, Wilhelm Fink Verlag, München.

KLEIN, D. & C. D. MANNING: 2002, 'Natural language grammar induction using a constituent-context model', in T. G. Dietterich, S. Becker & Z. Ghahramani (eds.), *Proceedings of the 2001 Neural Information Processing Systems Conference*, Band 14, MIT Press, Cambridge, MA, USA, S. 35–42.

KNUTH, D. E.: 1984, 'Literate programming', *The Computer Journal* 27(2), S. 97–111.

KUSTER, M. W., C. LUDWIG & A. ASCHENBRENNER: 2007, 'TextGrid as a Digital Ecosystem', in *2007 Inaugural IEEE-IES Digital EcoSystems and Technologies Conference*, IEEE, S. 506–511.

LISKOV, B. H. & J. M. WING: 2001, 'Behavioural subtyping using invariants and constraints', in *Formal methods for distributed processing*, Cambridge University Press, New York, NY, USA, S. 254–280.

LUND, K. & C. BURGESS: 1996, 'Producing high-dimensional semantic spaces from lexical co-occurrence', *Behavior Research Methods, Instruments & Computers* 28(2), S. 203–208.

LYONS, J.: 1971, *Einführung in die moderne Linguistik. 8. Auflage 1995*, Beck, München.

MACWHINNEY, B. J. & C. SNOW: 1990, 'The child language data exchange system: An update', *Journal of Child language* 17(2), S. 457–472.

MANNING, C. D., P. RAGHAVAN & H. SCHÜTZE: 2008, *Introduction to Information Retrieval*, Cambridge University Press, New York, NY, USA.

MANNING, C. D. & H. SCHÜTZE: 1999, *Foundations of Statistical Natural Language Processing*, MIT Press, Cambridge, MA, USA.

MASSOL, V. & T. HUSTED: 2003, *JUnit in Action*, Manning Publications Co., Greenwich, CT, USA.

MATTHEWS, P.: 2001, *A short history of structural linguistics*, Cambridge University Press, Cambridge, UK.

MCAFFER, J. & J.-M. LEMIEUX: 2005, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java™ Applications*, Addison-Wesley, Upper Saddle River, NJ.

MINTER, D. & J. LINWOOD: 2005, *Pro Hibernate 3*, Apress, Berkely, CA, USA.

Monson, C.: 2004, 'A framework for unsupervised natural language morphology induction', in *Proceedings of the ACL 2004 workshop on Student research*, Association for Computational Linguistics, Stroudsburg, PA, USA.

Özsu, M. T. & L. Liu (eds.): 2009, *Encyclopedia of Database Systems*, Springer.

Padó, S. & M. Lapata: 2007, 'Dependency-based Construction of Semantic Space Models', *Computational Linguistics* 33(2), S. 161–199.

Palmer, M., H. T. Ng & H. T. Dang: 2006, 'Evaluation of WSD Systems', in E. Agirre & P. Edmonds (eds.), *Word Sense Disambiguation. Algorithms and Applications*, Band 33, Springer Netherlands, S. 75–106.

Pedersen, T.: 2008, 'Empiricism is not a matter of faith', *Computational Linguistics* 34(3), S. 465–470.

Plötz, S.: 1972, *Transformationelle Analyse. Die Transformationstheorie von Zellig Harris und ihre Entwicklung*, Band 8, Athenäum, Frankfurt a. M., D.

Ramakrishnan, R. & J. Gehrke: 2000, *Database Management Systems*, zweite Ausgabe, McGraw-Hill, New York, NY, USA.

Rehm, G., A. Witt, H. Zinsmeister & J. Dellert: 2007, 'Corpus Masking: Legally Bypassing Licensing Restrictions for the Free Distribution of Text Collections', in *Proceedings of the Digital Humanities*, S. 166–170.

Rolshoven, J. & S. Schwiebert: 2007, 'Evidenzprozesse: Korpora, Kompression und Musterbildung', in C. M. Riehl & A. Rothe (eds.), *Was ist linguistische Evidenz?: Kolloquium des Zentrums Sprachenvielfalt und Mehrsprachigkeit, November 2006*, Shaker, Aachen, S. 91–108.

Sager, N. & R. Grishman: 1975, 'The restriction language for computer grammars of natural language', *Communications of the ACM* 18, S. 390–400.

Sager, N., M. Lyman, C. Bucknall, N. Nhan & L. J. Tick: 1994, 'Natural Language Processing and the Representation of clinical data', *Journal of the American Medical Informatics Association* 1(2), S. 142–160.

Sager, N. & N. T. Nhan: 2002, 'The computability of strings, transformations, and sublanguage', in B. E. Nevin & S. M. Johnson (eds.), *The legacy of Zellig Harris: language and information into the 21st century*, Band 2, John Benjamins, S. 79–120.

DE SAUSSURE, F.: 1931, *Grundfragen der allgemeinen Sprachwissenschaft. 2. Auflage 1967*, Walter De Gruyter Co.

SCHÄFER, U.: 2006, 'Middleware for Creating and Combining Multi-dimensional NLP Markup', in *Proceedings of the 5th Workshop on NLP and XML: Multi-Dimensional Markup in Natural Language Processing*, Association for Computational Linguistics, Stroudsburg, PA, USA, S. 81–84.

SCHMID, H.: 1994, 'Probabilistic part-of-speech tagging using decision trees', in *Proceedings of the International Conference on New Methods in Language Processing*.

SCHWIEBERT, S.: 2005, 'Entwurf eines agentengestützten Systems zur Paradigmenbildung', *Sprachtechnologie, mobile Kommunikation und linguistische Ressourcen. Beiträge zur GLDV-Tagung 2005 in Bonn* 8, S. 633–646.

SEARLE, J. R.: 1972, 'Chomsky's Revolution in Linguistics', *The New York Review of Books* 18(12).

SHANNON, B.: 2003, *Java^{TM} 2 Platform Enterprise Edition Specification, v1.4*, Sun Microsystems, Inc.

SMEETS, B. & S. LADD: 2007, *Building Spring 2 Enterprise Applications*, Apress, Berkely, CA, USA.

SOLAN, Z., D. HORN, E. RUPPIN & S. EDELMAN: 2003a, 'Unsupervised Efficient Learning and Representation of Language Structure', in *Proceedings of the 25th Conference of the Cognitive Science Society (CogSci 2003)*.

SOLAN, Z., E. RUPPIN, D. HORN & S. EDELMAN: 2003b, 'Automatic Acquisition and Efficient Representation of Syntactic Structures', in S. Backer, S. Thrun & K. Obermayer (eds.), *Proceedings of the 2002 Conference on Neural Information Processing Systems (NIPS'02)*, Band 15, MIT Press, Cambridge, MA, USA, S. 107–116.

SPYNS, P., N. T. NHÀN, E. BAERT, N. SAGER & G. D. MOOR: 1998, 'Medical Language Processing applied to extract clinical information from Dutch medical documents', in *MEDINFO 98: proceedings of the 9th World Congress on Medical Informatics, Part 1*, Band 52, IOS Press, S. 1–5.

Stehouwer, H. & M. van Zaanen: 2010a, 'Finding Patterns in Strings using Suffixarrays', in *Proceedings of the 2010 International Multiconference on Computer Science and Information Technology (IMCSIT)*, S. 505–511.

Stehouwer, H. & M. van Zaanen: 2010b, 'Using Suffix Arrays as Language Models: Scaling the n-gram', in *Proceedings of the 22st Benelux Conference on Artificial Intelligence (BNAIC-2010)*.

Szyperski, C., D. Gruntz & S. Murer: 1998, *Component Software. Beyond Object-Oriented Programming*, Addison-Wesley Professional, Boston, MA, USA.

Ukkonen, E.: 1995, 'On-line construction of suffix trees', *Algorithmica* 14(3), S. 249–260.

Vanhoutte, E.: 2004, 'An Introduction to the TEI and the TEI Consortium', *Literary and Linguistic Computing* 19(1), S. 9–16.

Verspoor, K., W. B. Jr., C. Roeder & L. Hunter: 2009, 'Abstracting the types away from a UIMA type system', in *From Form to Meaning: Processing Texts Automatically. Proceedings of the Biennial GSCL Conference 2009*, Gunter Narr Verlag, Tübingen, S. 249–256.

Wagner, R. A. & M. J. Fischer: 1974, 'The String-to-String Correction Problem', *Journal of the Association for Computing Machinery* 21(1), S. 168–173.

Wang, T. & G. Hirst: 2011, 'Refining the Notions of Depth and Density in WordNet-based Semantic Similarity Measures', in *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, S. 1003–1011.

Witten, I. H., E. Frank & M. A. Hall: 2005, *Data mining: practical machine learning tools and techniques*, Morgan Kaufmann, San Francisco, CA, USA.

Wu, Z. & M. Palmer: 1994, 'Verbs semantics and lexical selection', in *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, Association for Computational Linguistics, Stroudsburg, PA, USA, S. 133–138.

van Zaanen, M.: 2000a, 'ABL: Alignment-Based Learning', in *COLING 2000 - Proceedings of the 18th International Conference on Computational Linguistics*, Band 2, S. 961–967.

van Zaanen, M.: 2000b, 'Bootstrapping Structure using Similarity', in P. Monachesi (ed.), *Computational Linguistics in the Netherlands 1999. Selected Papers form the Tenth CLIN Meeting*, S. 235–245.

van Zaanen, M.: 2002, *Bootstrapping Structure into Language: Alignment-Based Learning*, Ph.D. thesis, School of Computing, University of Leeds, U.K., Leeds, UK.

van Zaanen, M.: 2003, 'Alignment-Based Learning versus Data-Oriented Parsing', in R. Bod, R. Scha & K. Sima'an (eds.), *Data-Oriented Parsing*, chap. 20, University of Chicago Press, S. 385–403.

van Zaanen, M. & J. Geertzen: 2008, 'Problems with evaluation of unsupervised empirical grammatical inference systems', in *ICGI '08: Proceedings of the 9th international colloquium on Grammatical Inference*, Band 5278, Springer, Berlin, Heidelberg, S. 301–303.